On the Implementation of Pfair-scheduled Multiprocessor Systems

by
Philip L. Holman

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2004

ABSTRACT

PHILIP L. HOLMAN: On the Implementation of Pfair-scheduled Multiprocessor Systems.
(Under the direction of James H. Anderson.)

The goal of this dissertation is to extend the Pfair scheduling approach in order to enable its efficient implementation on a real-time multiprocessor. At present, Pfair scheduling is the only known means for optimally scheduling recurrent real-time tasks on multiprocessors. In addition, there has been growing practical interest in such approaches due to their fairness guarantees. Unfortunately, prior work in this area has considered only the scheduling of independent tasks, which do not communicate with each other or share resources. The work presented herein focuses both on adding support for these actions and also on developing techniques for reducing various forms of implementation overhead, including that produced by task migrations and context switches. The thesis of this dissertation is:

tasks can be efficiently synchronized in Pfair-scheduled systems and overhead due to common system events, such as context switches and migrations, can be reduced.

This thesis is established through research in three areas. First, the pre-existing Pfair scheduling theory is extended to support the scheduling of groups of tasks as a single entity. Second, mechanisms for supporting both lock-based and lock-free synchronization are presented. Lock-based synchronization coordinates access to shared resources through the use of semaphores. On the other hand, lock-free operations are optimistically attempted and then retried if the operation fails. Last, the deployment of Pfair scheduling on a symmetric multiprocessor is considered.

To Traci.

ACKNOWLEDGMENTS

I have been fortunate to be mentored by people who have both inspired and challenged me. Foremost, I would like to thank Jim Anderson and Mark Edwards for their wisdom, advice, and friendship. I would also like to thank Sanjoy Baruah, Kevin Jeffay, Mark Moir, Srikanth Ramamurthy, Jan Prins, Ketan-Mayer Patel, Jasleen Kaur, Prasun Dewan, Steve Goddard, Prashant Shenoy, Mark Aulick, Chong-wei Xu, Arthur Woodrum, Wil Grant, and Marvin Payne. Special thanks to Sanjoy Baruah, Kevin Jeffay, Ketan-Mayer Patel, Jasleen Kaur, and Prashant Shenoy for their willingness to be on my committee. Finally, I would like to thank my fellow students Anand Srinivasan, Yong-jik Kim, Shelby Funk, Aaron Block, Uma Devi, Andrea Mantler, Dorian Miller, Swaha Das, Nathan Fisher, Vasile Bud, Abhishek Singh, and David Griggs for their support and friendship.

My work has been facilitated by the help and support of many others who deserve acknowledgment. First, thanks to the staff in the UNC Department of Computer Science, who have made my time here very enjoyable and also helped me countless times. In particular, I would like to thank Janet Jones, Madelyn Mann, Karen Thigpen, and Tammy Pike. Second, thanks to my family, particularly my parents, for their support and encouragement. Finally, I would like to thank my best friend and bride-to-be, Traci, who has endured my six-year absence while I sought this degree. She is my muse.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

BTB	Back to back
CBS	Constant-bandwidth server
DBA	Demand-based analysis
DFS	Deadline fair scheduling
DM	Directed migration
EDF	Earliest deadline first
EPDF	Earliest pseudo-deadline first
ER	Early release
FF	First fit
FIFO	First in, first out
FP-EDF	Fully preemptive EDF scheduling of jobs
FP-EDF-CS	Fully preemptive EDF scheduling of jobs with critical sections
FP-EDF-NP	Fully preemptive EDF scheduling of jobs with nonpreemptable code segments
GIS	Generalized intra-sporadic
IS	Intra-sporadic
JVM	Java virtual machine
LLF	Least laxity first
LS	Last scheduled
MESI	Modified-Exclusive-Shared-Invalid
MRAP	Most recent available processor
MRP	Most recent processor

NF	Next fit
PP	Previous processor
PS	Previous slot
PT	Previous task
PCP	Priority ceiling protocol
PIP	Priority inheritance protocol
QB-EDF	Quantum-based EDF scheduling of subtasks based on pseudo-jobs
QB-EPDF	Quantum-based EPDF scheduling of subtasks
QoS	Quality of service
QR	Quick release
RM	Rate monotonic
RP	Rollback protocol
RTJ	Real-time Java
SMP	Symmetric multiprocessor
SP	Skip protocol
SWSP	Static-weight server protocol
WM	Weight monotonic

LIST OF SYMBOLS

From Chapter 1:

J, K	Jobs	6
$J.a$	Arrival time of job J	6
$J.e$	Execution requirement of job J	6
$J.d$	Relative deadline of job J	6
$\mathbf{J}(a, e, d)$	A job that arrives at time a, and has an execution requirement of e and a relative deadline of d	6
$J.k$	Phase count of a multi-phase job J	7
$J^{[i]}$	The i^{th} phase of a multi-phase job J	7
$J^{[i]}.e$	The execution requirement of the i^{th} phase (an execution phase) of a multi-phase job J	7
$J^{[i]}.R$	The resource required by the i^{th} phase (an execution phase) of a multi-phase job J	7
$\mathbf{C}(R, e)$	An execution phase that requires resource R and has an execution requirement of e	7
$J^{[i]}.\theta$	The maximum suspension duration of the i^{th} phase (a suspension phase) of a multi-phase job J	7
$\mathbf{I}(\theta)$	A suspension phase with a maximum duration of θ	7
$J : P_1; \ldots; P_{J.k}$	A multi-phase job J consisting of phases $P_1, P_2, \ldots, P_{J.k}$ (in order of occurrence)	8
$\mathcal{I}$	The set of all suspension phases	9
$\mathcal{C}(\ell)$	The set of all execution phases that require resource ℓ	9
T, U	Tasks	9
$T.\phi$	Offset of a periodic or sporadic task T	9

$T.p$	Period of a periodic or sporadic task T	9
$T.e$	Per-job execution requirement of a periodic or sporadic task T	9
$T.d$	Per-job relative deadline of a periodic or sporadic task T	9
$\mathbf{P}(\phi, e, p, d)$	A periodic task with an offset of ϕ, a per-job execution requirement of e, a period of p, and a per-job relative deadline of d	9
$J.u$	Utilization of a periodic or sporadic task T	9
$T(i)$	The i^{th} job of a periodic of sporadic task T	10
$\mathbf{S}(\phi, e, p, d)$	A sporadic task with an offset of ϕ, a per-job execution requirement of e, a period of p, and a per-job relative deadline of d	10
$T : P_1; \ldots; P_{T.k}$	A multi-phase periodic or sporadic task T for which each job consists of phases $P_1, P_2, \ldots, P_{T.k}$ (in order of occurrence)	11
$T^{[i]}$	The i^{th} phase of any job of a multi-phase periodic of sporadic task T	11
$T.k$	Phase count of any job of a multi-phase periodic of sporadic task T	11
M	Processor count	14
$\lvert S \rvert$	Size of the set S	14
τ	The set of tasks	14
$\tau.u$	Cumulative utilization of all tasks in τ	14
Q	Quantum size	20

From Chapter 2:

$T.w$	Weight of a task T	34
$\mathbf{PF}(w)$	A Pfair task with weight w	35
$\mathit{fluid}(T, t_1, t_2)$	Share of processor time entitled to task T over the interval $[t_1, t_2)$	35
$\mathit{lag}(T, t)$	The lag of task T at time t	35

$\mathit{received}(T, t_1, t_2)$	Processor time allocated to task T over the interval $[t_1, t_2)$	36
T_i	The i^{th} subtask of a task T	38
$\omega(T_i)$	The window associated with the i^{th} subtask of a task T	38
$r(T_i)$	The pseudo-release of the i^{th} subtask of a task T	38
$d(T_i)$	The pseudo-deadline of the i^{th} subtask of a task T	38
$b(T_i)$	The successor bit (b-bit) of the i^{th} subtask of a task T	39
$\mathcal{E}(T)$	Per-cycle execution requirement of a task T	40
$\mathcal{P}(T)$	Cycle period of a task T	40
$\mathsf{maxsum}_k\ S$	The maximum value produced by summing $\min(k, \|S\|)$ elements from the multiset S	41
$D(T_i)$	The group deadline of the i^{th} subtask of a task T	44
$e(T_i)$	The eligibility time of the i^{th} subtask of a task T	46
$\mathbf{GIS}(w)$	A GISfair task with weight w	48
$\mathcal{S}, \mathcal{T}$	Supertasks	49
π	The set of supertasks	50

From Chapter 3:

$\xi_{\max}$	Maximum duration of the per-slot scheduler invocation on a single processor	56
$\xi_{\min}$	Minimum duration of the per-slot scheduler invocation on a single processor	56
$\jmath_+$	Maximum difference (caused by event jitter) between the actual occurrence time of an event and the expected occurrence time	56
$\jmath_-$	Maximum difference (caused by event jitter) between the expected occurrence time of an event and the actual occurrence time	56
φ	Maximum duration of any single event postponement	56

$\imath_{\max}$	Maximum amount of processor time consumed by immediate interrupt handlers within a slot on a single processor	57
$\imath_{\min}$	Minimum amount of processor time consumed by immediate interrupt handlers within a slot on a single processor	57
$S_{\max}$	Maximum runtime duration of a single slot	58
$S_{\min}$	Minimum runtime duration of a single slot	58
$Q_{\max}$	Maximum amount of processor time provided by a single quantum	58
$Q_{\min}$	Minimum amount of processor time provided by a single quantum	58
$T.c$	Tardiness threshold of task T	63
β_+	Upper lag scaler	65
β_-	Lower lag scaler	65
β	Sum of the lag scalers	65
ϵ_r	Pseudo-release extension	65
ϵ_d	Pseudo-deadline extension	65
ϵ	Sum of window extensions	67
$\mathcal{Z}$	Set of all integers	67
$\mathcal{B}(T)$	Cycle overlap of a task T	69
$span(T, n)$	Maximum n-span of a task T	86

From Chapter 4:

$T.J$	Number of consecutive cycles making up a pseudo-job for task T	103
$T.v$	Maximum duration for which task T can execute non-preemptably	104
$demand(S, t_1, t_2)$	The minimum amount of processor time needed by tasks in S to avoid a timing violation over the interval $[t_1, t_2)$	108

$\mathit{supply}(S, t_1, t_2)$	The amount of processor time provided to tasks in S over the interval $[t_1, t_2)$	108
$\mathit{demand}_M(S, t_1, t_2)$	The minimum amount of processor time needed by mandatory requests in S to avoid a timing violation over the interval $[t_1, t_2)$	111
$\mathit{demand}_C(S, t_1, t_2)$	The minimum amount of processor time needed by requests in S that are not mandatory to avoid a timing violation over the interval $[t_1, t_2)$	111
L_0	The shortest interval length over which a request of a component task can arrive and experience a timing violation	112
$\Gamma(T)$	The set of locks requested by task T	112
$\mathcal{S}_L$	The set of component tasks of $\mathcal{S}$ with miss thresholds at most L	121
Γ_L	The set of locks requested by mandatory jobs within an interval of length L	125
$\mu(\ell, L)$	The longest critical section that requires lock ℓ and does not belong to a task with mandatory jobs in an interval of length L	112
$\Delta(\mathcal{S}, L)$	The minimum value of $\mathcal{S}.w$ needed to avoid timing violations over intervals of length L	127
$\mathcal{L}$	The testing set of $\Delta(\mathcal{S}, L)$	127
$\mathcal{S}.w_{\text{opt}}$	The smallest value of $\mathcal{S}.w$ that satisfies all $\Delta(\mathcal{S}, L)$ restrictions, *i.e.*, the optimal reweighting solution	127
υ_L	The maximum circumstantial demand generated by non-preemptable code segments over an interval of length L	129
$\mathcal{N}$	Set of all natural numbers	130
$\phi(\mathcal{S}, L)$	The monotonic bounding function of $\Delta(\mathcal{S}, L)$	134
$\Psi(\mathcal{S})$	The characteristic function of $\phi(\mathcal{S}, L)$	136
$\mathcal{S}.w_\phi$	The limit of $\phi(\mathcal{S}, L)$ as $L \to \infty$	138

From Chapter 5:

Γ	The set of all shared resources (either locks and lock-free objects, depending on the context)	179
ℓ	Shared resource (either a lock or lock-free object, depending on the context)	179
$A(T, \ell)$	The maximum number of accesses made to the lock-free object ℓ by a task T within a single quantum	179
$A(\mathcal{S}, \ell)$	The maximum number of accesses made to the lock-free object ℓ from within the supertask $\mathcal{S}$ within a single quantum	179
$N(\ell)$	The maximum number of processors from which tasks may access the lock-free object ℓ in a single slot	179
$e_B(\ell)$	The base execution requirement of each access to object ℓ	179
$e_R(\ell)$	The retry cost of object ℓ	179
$e_B^{[m]}(\ell)$	The base execution requirement of each access to the lock-free object ℓ when tasks on at most m processors can access ℓ in each slot	179
$e_R^{[m]}(\ell)$	The retry cost of the lock-free object ℓ when tasks on at most m processors can access ℓ in each slot	179
$I(T, \ell)$	The maximum number of interferences suffered by a task T when accessing the lock-free object ℓ within a single slot when not using supertasks	180
$I(\mathcal{S}, \ell)$	The maximum number of interferences suffered by any single task $T \in \mathcal{S}$ when accessing the lock-free object ℓ within a single slot	180
$\mathcal{J}(T, \ell)$	The maximum number of accesses made to the lock-free object ℓ by a task T within a single job	182

From Chapter 6:

$T^{[i]}.B$	The length of the blocking zone associated with the request of a locking phase $T^{[i]}$	254

$T.\hat{e}(\ell)$	The duration of the longest critical section of T requiring ℓ	254
$\mathcal{S}.\hat{e}(\ell)$	The duration of the longest critical section of any task T in supertask $\mathcal{S}$ that requires ℓ	254
$Q^m(T,\ell)$	The maximum time needed to service competing requests of a request of task T across m quanta	254
$I(T,\ell)$	The maximum time needed to service all competing requests of a request of task T	254
$Q^m(\mathcal{S},\ell)$	The maximum time needed to service competing requests of a request of task T ($\in \mathcal{S}$) across m quanta when using supertasks	254
$I(\mathcal{S},\ell)$	The maximum time needed to service all competing requests of a request of task T ($\in \mathcal{S}$) when using supertasks	255
$\mathcal{U}(T,\ell)$	Task T's utilization of lock ℓ	274
$\mathcal{U}(\ell)$	The total utilization of lock ℓ	274

CHAPTER 1

Introduction

The focus of this dissertation is the design of a real-time multiprocessor operating system that uses Pfair scheduling algorithms. This chapter introduces real-time scheduling, articulates the goals of this dissertation, and summarizes results presented in later chapters. The next chapter discusses the Pfair scheduling approach in detail.

In Section 1.1, we discuss the research focus of this dissertation and the importance of real-time multiprocessor systems. Section 1.2 summarizes models used to represent real-time processes. In Section 1.3, we discuss issues that arise when sharing a processor among a collection of real-time processes. This is followed by a discussion of multiprocessor approaches in Section 1.4. We conclude the chapter by summarizing our goals (Section 1.5) and specific contributions (Section 1.6).

1.1 Research Focus

Program correctness is defined by two requirements. Programs must exhibit *logical correctness* and must eventually *terminate*. Informally, logical correctness requires that programs produce expected results. Real-time constraints impose a third requirement: *temporal correctness*. Temporal correctness strengthens these previous requirements by also imposing restrictions on the rate at which programs execute. For example, the most common real-time constraint is a *deadline*, which is an absolute time by which the associated activity must complete. Informally, temporal correctness requires that results must be produced on time.

A *real-time* system is one that imposes real-time constraints on at least one program. For example, real-time systems are often used to control mechanical systems. Such control systems typically consist of a uniprocessor and a static collection of programs, each of which must complete within a fixed amount of time whenever invoked. For instance, consider how paper currency (*e.g.*, dollar bills) are printed. The printing press actually produces large sheets that contain multiple bills laid out in a grid. These sheets are then fed into a second machine that uses a press to cut the sheets into individual bills. When automated, this system requires timing constraints to ensure that the movement of the mechanical parts is synchronized at all times. For example, if the insertion and removal of sheets into the cutting press is not synchronized with the lowering of the press, then bills will not be cut properly. Due to the cost and potential danger associated with such malfunctions, control systems often require off-line validation, *i.e.*, operation must be analyzed to show that timing constraints are always met under assumed operating conditions. Hence, *predictability* is paramount among all concerns. The need for validation techniques in control systems has given rise to a considerable body of research that focuses on the analysis of worst-case behavior.

Growing interest in providing *quality-of-service* (QoS) guarantees has expanded the scope of real-time research from simply focusing on the worst-case performance to also considering average and best-case performance. This interest stems largely from the widespread use of multimedia applications and from the commercialization of the Internet. For instance, many companies currently lease space on large web servers that is used by subscribers to host web sites. Ideally, these companies would like to impose strict limits on the quantity of resources (*e.g.*, network capacity, processor time, disk space, *etc.*) consumed by each subscriber. Although existing techniques partially address this need (*e.g.*, quota support in file systems), the sharing of processor capacity continues to be achieved through the use of unreliable *ad hoc* polices. Similarly, multimedia applications (*e.g.*, audio and video players) are common in modern computers. Optimum performance requires timely access to both storage media and processors. Delaying this access can produce artifacts in the playback, such as audible pops or clicks during a song. Hence, ensuring timely and predictable access to system resources, including processor time, is a growing concern in many areas.

Dissertation focus. The focus of this dissertation is multiprocessor real-time systems. Our interest in real-time techniques is based on their ability to provide predictable operation. We consider multiprocessors for several reasons, which are summarized below.

First, real-time uniprocessor systems have been and continue to be widely studied, and with much success. However, the use of real-time multiprocessor systems remains a largely unexplored area. Indeed, real-time multiprocessor systems are typically controlled by a partitioning approach, in which each process (or *task*) is bound to a single processor. This approach effectively treats a multiprocessor system as a collection of independent uniprocessor systems. Partitioning provides many practical benefits, including no migration[1] overhead and improved cache performance. In addition, uniprocessor techniques, which tend to be more efficient than their multiprocessor counterparts, can be applied on each processor. Unfortunately, a change in the set of runnable tasks may necessitate an on-line re-partitioning of the system. Since partitioning is a bin-packing problem, which is known to be NP-hard in

[1]For a task to execute on a processor, that processor must have access to the task's runtime state (or *context*). Here, *migration* refers to the act of moving a task's context from one processor to another.

the strong sense [25], timely re-partitioning requires the use of a suboptimal heuristic. Even with such a heuristic, the overhead may be unacceptable. (A partitioning algorithm has $\Omega(N)$ time complexity where N is the number of tasks.) Hence, alternative approaches are needed for such systems.

Second, microprocessor-based multiprocessors are often less expensive and provide more design flexibility than uniprocessors of equivalent power. For instance, in [29], Hennessy and Patterson note that multiprocessors built from mass-produced microprocessors are highly effective when used to share resources among multiple applications. In [69], Wolf echoes this sentiment when comparing the use of multiprocessors to that of custom logic in embedded systems. In both cases, the practicality of multiprocessor designs is attributed to the low hardware costs provided by mass production and to the attention given to optimizing the performance of mass-produced hardware, particularly processors.

Finally, some applications exceed the capabilities of a single processor. In such cases, the use of a multiprocessor is unavoidable. For instance, processing the information produced by a sensor network is quite time-consuming. In addition, such networks can be deployed in situations where malfunctions endanger lives, such as military actions. This suggests a need for off-line validation. Restricting attention to uniprocessor designs imposes limits on both the size of the network and the complexity of the processing. In practice, such limitations may be unacceptable. Exceeding these limits requires the use of either a multiprocessor or a more powerful uniprocessor. Unfortunately, uniprocessor performance improvements are increasingly the product of applying complex heuristics within the hardware, such as branch prediction and out-of-order execution of instructions. Although these techniques improve performance on average, their complexity makes runtime behavior more unpredictable, and hence necessitates the use of overly pessimistic assumptions when approximating worst-case performance. Indeed, Hennessy and Patterson observed that performance improvements obtained in this fashion appear to have already reached a point of diminishing returns [29]. Hence, it seems that the use of multiprocessors cannot be altogether avoided.

1.2 Real-time Process Models

In this and the next two sections, we present an overview of the issues involved when sharing processors among a set of real-time tasks. We initiate this discussion in this section by presenting common models for representing such tasks. These models drive both the discussion in this chapter and the work presented in later chapters.

We define a *task* to be any finite-duration code segment that requires only one processor and for which execution can be scheduled. Tasks can be either *persistent* or *temporary*: a persistent task may be invoked any number of times while a temporary task is invoked only a finite number of times. We refer to each invocation of a task as a *job*.

1.2.1 Representing Jobs

We now present notation that will be used to describe jobs. Two complications are common when scheduling jobs in real systems. These complications must be taken into account by the models described below.

First, a job may require exclusive access to a shared resource other than processor time (*e.g.*, a file, a data structure in shared memory, an external device, *etc.*) during its execution. We refer to a region of code with such a requirement as a *critical section*. For instance, an operation on a shared object that is represented by two words of memory may need to update both words. If exclusive access is not ensured while applying this operation, then a second operation may read the object while it is in an inconsistent state, *i.e.*, after performing the first write but before performing the second. Jobs (and tasks) are said to be *independent* when they have no critical sections.[2]

Second, a job may need to *suspend* its execution either for a fixed period of time or until some specified event occurs. Informally, a suspension is a period of time during which the task cannot make progress by executing and hence should not be granted processor time. For instance, a job that writes a buffer's contents to a disk may need to write multiple segments of the buffer in a series of operations. To accomplish this, the job would initiate the write of a segment and then request suspension until the disk controller signals completion of that write. Once this signal is received, the next write is initiated, and so on.

Simple jobs. An independent job J that never suspends can be fully characterized by the following three parameters.

- $J.a$, called J's *arrival* or *release* time, is the earliest time at which J can begin executing;
- $J.e$, called J's *execution requirement*, is the maximum processor time needed by J;
- $J.d$, called J's *relative deadline*, is the maximum allowed difference of J's completion time and its arrival time, *i.e.*, J must complete by its *absolute deadline* at time $J.a + J.d$.

The shorthand $J = \mathbf{J}(a, e, d)$ will be used to denote a job J with $J.a = a$, $J.e = e$, and $J.d = d$. A job with no timing constraint can be specified by letting $J.d = \infty$.

Figure 1.1 shows the execution of two such jobs. Job **J1** arrives at time 24, requires 7 units of processor time, and must complete within 18 time units (*i.e.*, by time 42). Job **J2**

[2]Dependencies among jobs also arise from *precedence constraints*, *i.e.*, the execution of one job may need to be postponed until another job has completed. Such a constraint typically arises from a producer-consumer relationship, in which the first job produces data required by the second job.

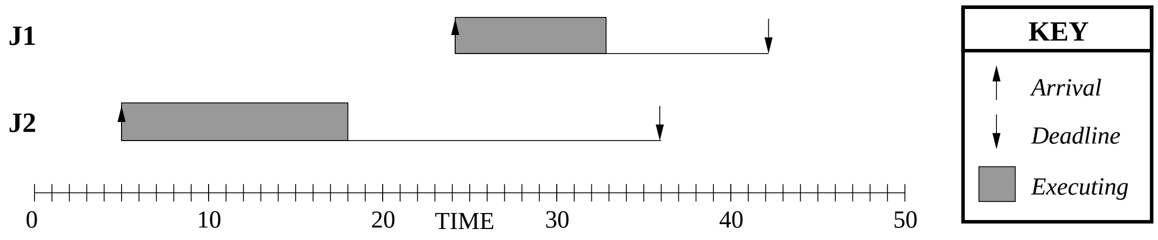


Figure 1.1: Two-job schedule with jobs $\mathbf{J1} = \mathbf{J}(24, 7, 18)$ and $\mathbf{J2} = \mathbf{J}(5, 13, 31)$. Neither job contains critical sections or suspends. Each job is executed immediately upon arrival.

arrives at time 5, requires 13 units of processor time, and must complete within 31 time units (*i.e.*, by time 36). The figure illustrates one possible uniprocessor schedule, in which each job is executed immediately upon arrival.

Complex jobs. A multi-phase representation will be used to describe jobs that either suspend or include critical sections. This representation decomposes each job J into a sequence of $J.k$ phases. (When the range of a phase index i is not explicitly given, the range $1 \leq i \leq J.k$ should be assumed.) Each phase $J^{[i]}$ is either an execution phase or a suspension phase.

An execution phase is described by two parameters. $J^{[i]}.e$ (respectively, $J^{[i]}.R$) denotes the *execution requirement* (respectively, *resource*) of phase $J^{[i]}$. $J^{[i]}.R$ indicates which (non-processor) shared resource[3] is required by $J^{[i]}$. If $J^{[i]}$ is *not* a critical section, then $J^{[i]}.R = \emptyset$. We let $J^{[i]} = \mathbf{C}(R, e)$ denote an execution phase $J^{[i]}$ with $J^{[i]}.R = R$ and $J^{[i]}.e = e$.

A suspension phase $J^{[i]}$ is characterized only by its maximum duration,[4] denoted $J^{[i]}.\theta$. Suspension phases are assumed to never occur consecutively. (Consecutive suspensions can be expressed as a single phase.) We let $J^{[i]} = \mathbf{I}(\theta)$ denote a suspension phase $J^{[i]}$ with $J^{[i]}.\theta = \theta$.

[3]The model considered here does not support nested critical sections. Adding support for such critical sections is a topic for future work.

[4]Knowing the minimum duration is often useful also; however, for our purposes, the maximum duration is sufficient. The reason for this will become more apparent in Chapter 3.

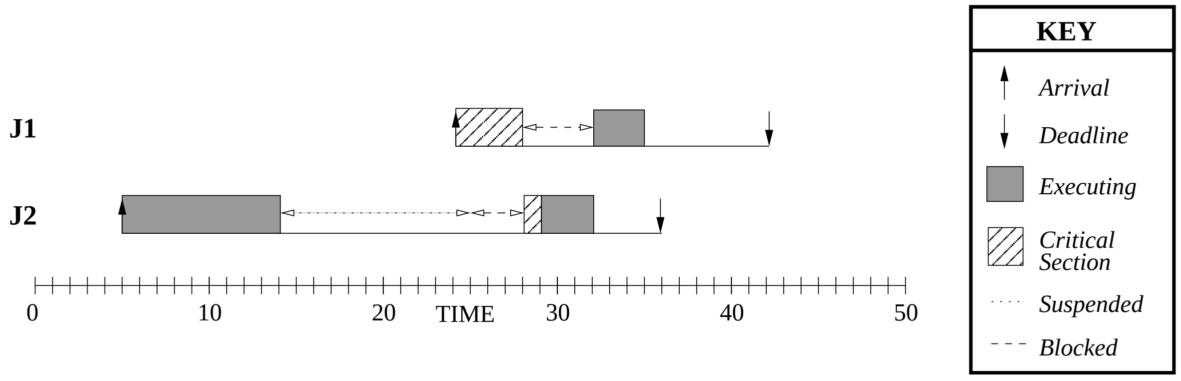


Figure 1.2: Two-job schedule with jobs $\mathbf{J1} = \mathbf{J}(24, 7, 18) : \mathbf{C}(\ell, 4)\,; \mathbf{C}(\emptyset, 3)$ and $\mathbf{J2} = \mathbf{J}(5, 13, 31) : \mathbf{C}(\emptyset, 9)\,; \mathbf{I}(11)\,; \mathbf{C}(\ell, 1)\,; \mathbf{C}(\emptyset, 3)$. Job **J2** is given priority over **J1**. However, due to **J2**'s suspension, **J1** is able to execute and gain exclusive access to the resource ℓ needed by **J2**, which results in 3 time units of blocking when **J2**'s suspension ends.

A multi-phase job will be represented by the combination of a simple job description, followed by a colon and then a semicolon-separated list of phase descriptions, given in increasing index order. For instance, a multi-phase job J that arrives at time 5, has a relative deadline of 10 time units, and consists of phases $\mathbf{C}(\emptyset, 5)$, $\mathbf{I}(1)$, and $\mathbf{C}(\ell, 1)$ (in order from earliest to latest) would be expressed as $J = \mathbf{J}(5, 6, 10) : \mathbf{C}(\emptyset, 5)\,; \mathbf{I}(1)\,; \mathbf{C}(\ell, 1)$. (Note that the execution requirement in the job description is the sum of the phase requirements.)

Figure 1.2 illustrates multi-phase jobs and both complicating behaviors (*i.e.*, critical sections and suspensions). In this example, job **J2** is given priority over **J1**. **J2** executes its first phase immediately upon arrival. It then initiates its suspension at time 14, which endures for 11 time units. In the meantime, **J1** arrives at time 24 and is permitted to immediately begin its first phase. Because **J1** is granted exclusive access to ℓ at time 24, **J2** cannot execute when its suspension ends at time 25 since its next phase also requires ℓ; **J1** is said to *block* **J2** at this time. Consequently, **J1** is allowed to continue executing. (Since this decision goes against the priority assignment, a *priority inversion* is said to occur over the time interval

[25, 28].) Once **J1**'s critical section (*i.e.*, first phase) ends, **J2** *preempts* **J1** and executes until completion. (To simplify the example, preemptions are assumed to incur no overhead.) After **J2** completes at time 32, **J1** then executes until completing at time 35.

To facilitate the statement of results, let $J^{[i]} \in \mathcal{I}$ denote that $J^{[i]}$ is a suspension phase. Similarly, let $J^{[i]} \in \mathcal{C}(\ell)$ denote that $J^{[i]}$ is an execution phase for which $J^{[i]}.R = \ell$.

1.2.2 Representing Tasks

One-shot model. The simplest form of temporary task is one that is invoked only once. Such tasks are called *one-shot* (and sometimes *aperiodic*) tasks. Since the job representations given above are sufficient to describe such a task, we introduce no additional notation.

Periodic model. The *periodic* task model [47] was proposed for control systems, in which tasks are periodically invoked to sample data from external devices. A periodic task T is specified by four parameters:

- $T.\phi$, called T's *offset*, is the arrival time of T's first job;
- $T.e$ is the per-job execution requirement of T;
- $T.p$, called T's *period*, is the *exact* separation between consecutive job arrivals;
- $T.d$ is the per-job relative deadline of T.

A periodic task T is said to be *synchronous* when $T.\phi = 0$, and *asynchronous* otherwise. We let $T = \mathbf{P}(\phi, e, p, d)$ denote a periodic task T with $T.\phi = \phi$, $T.e = e$, $T.p = p$, and $T.d = d$. In addition to the above parameters, a task is often characterized by its *utilization*, which is defined to be $T.u = \frac{T.e}{T.p}$.

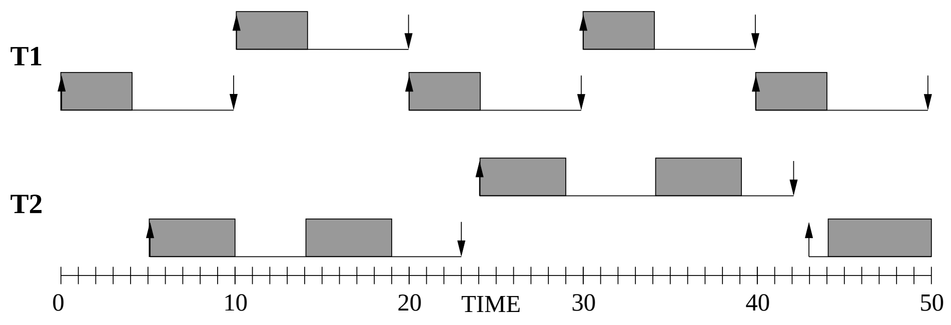


Figure 1.3: Two-task schedule with periodic tasks $\mathbf{T1} = \mathbf{P}(0, 4, 10, 10)$ and $\mathbf{T2} = \mathbf{P}(5, 10, 19, 18)$. The first five jobs of **T1** and the first three jobs of **T2** are shown. All jobs are independent and never suspend. Task **T1**'s jobs are given priority over **T2**'s jobs.

Figure 1.3 shows a schedule consisting of two periodic tasks: a synchronous task **T1** and an asynchronous task **T2**. Task **T1** is given priority over **T2** in this schedule. Notice that each task is invoked (*i.e.*, a job is *released*) periodically. Specifically, the i^{th} job of a periodic task T, denoted $T(i)$, is described by $T(i) = \mathbf{J}(T.\phi + (i-1)T.p, T.e, T.d)$. For example, the third job of $\mathbf{T1} = \mathbf{P}(0, 4, 10, 10)$ is given by $\mathbf{T1}(3) = \mathbf{J}(0 + (3-1) \cdot 10, 4, 10) = \mathbf{J}(20, 4, 10)$.

Sporadic model. The *sporadic* task model [49] was proposed to provide more flexibility than that provided by the periodic model. Specifically, the assumption that task invocations are evenly spaced may be unrealistic for some tasks. For instance, a single logical operation may be performed in stages by jobs of different tasks. Such jobs must access the data in proper sequence to ensure correctness of the operation, which gives rise to a producer-consumer relationship. Since the releases of jobs may need to be delayed until their data becomes available, the delay between consecutive releases may vary over time. To allow such variation, the sporadic model relaxes the periodic model by interpreting each task's period to be the *minimum* separation between consecutive invocations rather than the exact separation. We let $T = \mathbf{S}(\phi, e, p, d)$ denote a sporadic task T with $T.\phi = \phi$, $T.e = e$, $T.p = p$, and $T.d = d$.

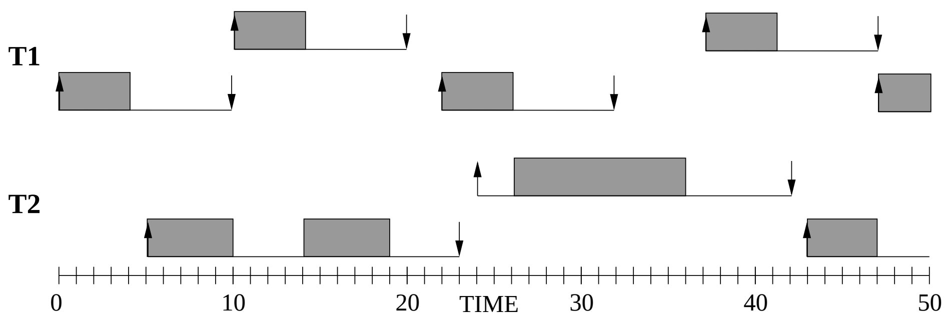


Figure 1.4: Two-task schedule with tasks $\mathbf{T1} = \mathbf{S}(0, 4, 10, 10)$ and $\mathbf{T2} = \mathbf{P}(5, 10, 19, 18)$. The first five jobs of **T1** and the first three jobs of **T2** are shown. All jobs are independent and never suspend. Task **T1**'s jobs are given priority over **T2**'s jobs.

Figure 1.4 is the result of changing task **T1** from a periodic to a sporadic task. Notice that the separation between the second and third job arrivals of **T1** exceeds the assumed minimum. This is also true of the separation between the third and fourth job arrivals.

Multi-phase tasks. Since all jobs of a task share a common source code, they must be identical. Hence, a multi-phase description that applies to one job must apply to all. To represent multi-phase tasks, the notation given previously for multi-phase jobs will be used with the following modification: a task description replaces the simple job description. For example, a task $T = \mathbf{P}(5, 6, 10, 10)$ in which each job consists of phases $\mathbf{C}(\emptyset, 5)$, $\mathbf{I}(1)$, and $\mathbf{C}(\ell, 1)$ (in order of occurrence) would be expressed as $T = \mathbf{P}(5, 6, 10, 10) : \mathbf{C}(\emptyset, 5)\,; \mathbf{I}(1)\,; \mathbf{C}(\ell, 1)$. Furthermore, since each job is identical, we let $T^{[i]}$ and $T.k$ denote the i^{th} phase and the phase count, respectively, of *any* job of T.

1.2.3 Selecting Parameters

A key problem when analyzing the behavior of real-time systems is the selection of parameters (*e.g.*, deadlines and execution requirements) that accurately describe the system. Techniques for achieving this goal are often complex and vary depending upon what assumptions are appropriate. A full discussion of these techniques is outside the scope of this dissertation. However, we give a brief overview of how each type of parameter is selected below.

Execution requirements. Two primary factors complicate the determination of execution requirements. First, execution times are affected by virtually every aspect of a system, including the structure of the source code, the processor architecture, the processor clock speed, the memory architecture, the operating-system structure, and how often tasks are preempted and migrated at runtime. Accurately determining the impact of each factor on a task's execution time is an incredibly complex problem. Second, the accuracy of tools designed to analyze source code is fundamentally limited. This follows from the fact that the *halting problem* has been shown to be unsolvable [68]. Informally, the goal of this problem is to produce an algorithm capable of determining whether an arbitrary algorithm, given as input, will always terminate. Since analyzing the execution time of a code segment (called *timing analysis*) also involves solving the halting problem, clearly no general solution exists.

Despite these complexities, the importance of timing analysis has motivated a considerable body of work. Research has shown that reasonably tight bounds can be obtained by restricting the form of the source code. For instance, forbidding the use of complex looping structures facilitates accurate analysis. Unfortunately, modern processor designs have made timing analysis more difficult by using complex techniques like caching and instruction pipelining.

Under pipelining, multiple instructions are processed simultaneously by different parts of the processor. On average, pipelining increases instruction throughput and hardware utilization. Unfortunately, it also reduces predictability and makes context switching more complex. Specifically, partially executed instructions in the pipeline must be nullified and the pipeline must be flushed before the new task can begin executing. In addition, the instruction pointer of the preempted task may need to be set back so that nullified instructions are re-issued when the task resumes execution. A more detailed explanation of pipelining and the complications it introduces can be found in [28]. Examples of how such complexities are factored into timing analysis can be found in [27, 36, 41, 43].

Deadlines and periods. Deadlines and periods are typically assigned values based upon the purpose of the task. These values are commonly influenced by external factors, such as the sampling rates of external sensors. For instance, consider a sensor with a sampling rate of 30 Hz (*i.e.*, 30 samples are produced per second) and a task that processes those samples. To avoid losing samples, the processing rate must equal or exceed the sampling rate. Hence, if each job processes 6 samples, then the task period should be at most 200 milliseconds. With exactly this period, the processing rate is $\frac{6 \text{ samples}}{0.2 \text{ seconds}} = 30$ Hz.

When sensors that support multiple sampling rates are used, it may be reasonable to consider reducing the sampling rates of sensors so that task periods and deadlines can be increased. This action will likely produce a decline in the quality of results, but can potentially reduce overhead as well. A simple example of such a sensor is a microphone that supports audio recording at variable rates. Audio clips recorded at less than ideal rates may still provide acceptable quality.

Suspensions. Techniques for bounding the duration of suspensions vary depending on the cause. In an earlier example, we considered a task that suspends while waiting for a write operation on a disk drive to complete. Such delays are called *I/O suspensions*. When accessing disks, delays are a function of the number of bytes being accessed and the time required to move the mechanical parts of the drive. Figure 1.2, discussed earlier, illustrates the second common cause of suspensions: resource contention. Such suspensions are called *blocking suspensions*. The duration of a blocking suspension is determined by the number of competing requests for the resource and the order in which these requests are granted.

1.3 Basics of Real-time Scheduling

When the number of tasks does not exceed the processor count, each task can simply be assigned a dedicated processor. However, when this is not the case, at least one processor must be shared among multiple tasks. In this section, we consider the design of a *scheduler*, which is the mechanism within an operating system that controls how processors are shared among tasks. The set of rules that govern this behavior is called the *scheduling policy* of the operating system. The goal of this section is to introduce concepts and notation that are common to all scheduling problems of relevance to this dissertation. In the next section, we discuss issues that are specific to multiprocessor scheduling.

1.3.1 Notation

We let M denote the processor count. Also, we use $|S|$ to denote the number of elements in the set S. We let τ denote the set of persistent tasks to be scheduled on the processors. (Temporary tasks are typically handled by using persistent server tasks, which reside in τ.) When considering periodic or sporadic tasks, the cumulative utilization of τ is given by $\tau.u = \sum_{T \in \tau} T.u$.

1.3.2 Task Prioritization

Scheduling consists of assigning priorities to all jobs that are eligible for execution and then selecting those with the highest priorities. Efficient prioritization is achieved by assigning an initial priority to each task and then updating its priority as needed on-line. To avoid re-sorting tasks unnecessarily, tasks are typically stored in a data structure that maintains either a partial or total ordering by priority (*e.g.*, a list or heap).

The choice of priority rules represents a trade-off between the effectiveness of scheduling and runtime overhead. On-line priority changes incur overhead. However, restricting the number of changes can prevent successful scheduling of some task sets. To characterize this trade-off, scheduling policies can be divided into three classes [16], which are described below.

Static-priority scheduling. Under *static-priority* scheduling, priorities never change once they are assigned. Hence, each job of a task inherits its initial priority. Static-priority schedulers are inherently *memoryless* in that the prioritization of tasks is independent of previous scheduling decisions. *Rate-monotonic* (RM) scheduling [47] is an example of such a policy for periodic and sporadic tasks. Under the RM prioritization, tasks with shorter periods are given higher priority. Figure 1.5 shows a sample schedule produced by applying the RM policy to a set of four tasks on a uniprocessor.

Task-level dynamic-priority scheduling. Under *task-level dynamic-priority* scheduling, each job can be assigned a unique fixed priority, *i.e.*, job priorities cannot change once assigned. (This class could also accurately be called *job-level static-priority* scheduling.) *First-in, first-out* (FIFO) scheduling is probably the simplest and most well-known task-level dynamic-priority policy. Under this policy, jobs are executed in arrival order. *Earliest-deadline-first* (EDF) scheduling[5] [47] is a task-level dynamic-priority policy often considered when scheduling periodic and sporadic tasks. Under the EDF prioritization, jobs with earlier absolute deadlines are given higher priority. Figure 1.6 shows the sample schedule produced

[5] Referred to as *deadline-driven* scheduling in [47].

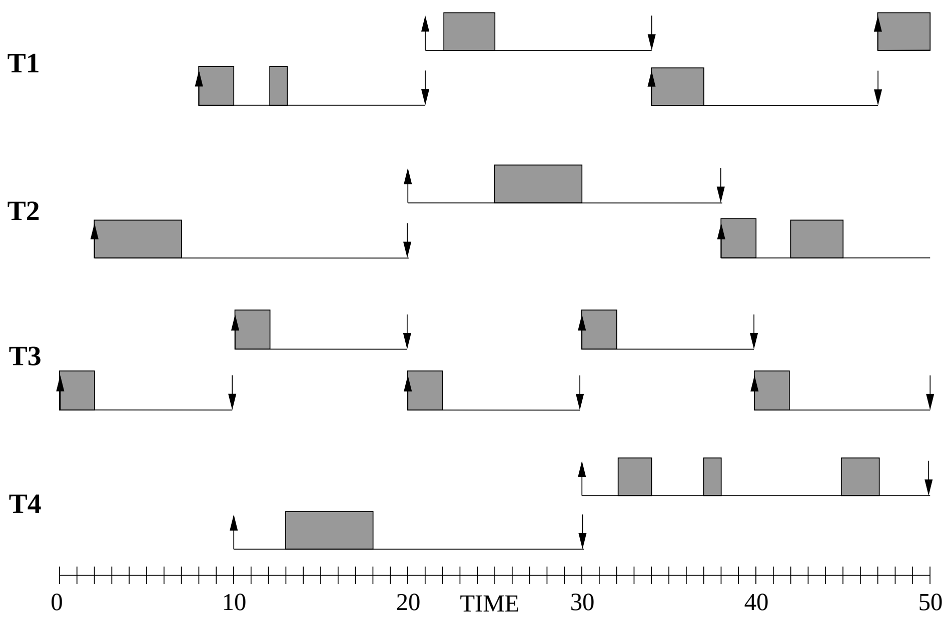


Figure 1.5: Four-task schedule with tasks $\mathbf{T1} = \mathbf{P}(8, 3, 13, 13)$, $\mathbf{T2} = \mathbf{P}(2, 5, 18, 18)$, $\mathbf{T3} = \mathbf{P}(0, 2, 10, 10)$ and $\mathbf{T4} = \mathbf{P}(10, 5, 20, 20)$. Jobs are prioritized according to the RM policy. The priority order (from highest to lowest) is as follows: **T3**, **T1**, **T2**, **T4**.

by applying the EDF policy to the same task set considered in Figure 1.5. These schedules differ only in the execution order of **T2**(3), **T3**(5), and **T4**(2).

Job-level dynamic-priority scheduling. Under *job-level dynamic-priority* scheduling, the frequency of priority changes is unrestricted. *Least-laxity-first* (LLF) scheduling[6] [42, 49] is an example of a job-level dynamic-priority policy for periodic and sporadic tasks. The LLF prioritization is based on the concept of *laxity*, which is defined to be maximum amount of time a job's execution can be delayed without causing it to miss its deadline. Letting t denote the current time and $\varepsilon(t)$ denote the *remaining* execution requirement of job J at time t, laxity can be formerly defined as $(J.a + J.d) - t - \varepsilon(t)$. Under LLF, jobs with lower laxities are given higher priority. Applying the LLF policy to the task set considered in

[6]Also called *minimum-laxity-first* and *least-slack-time* scheduling.

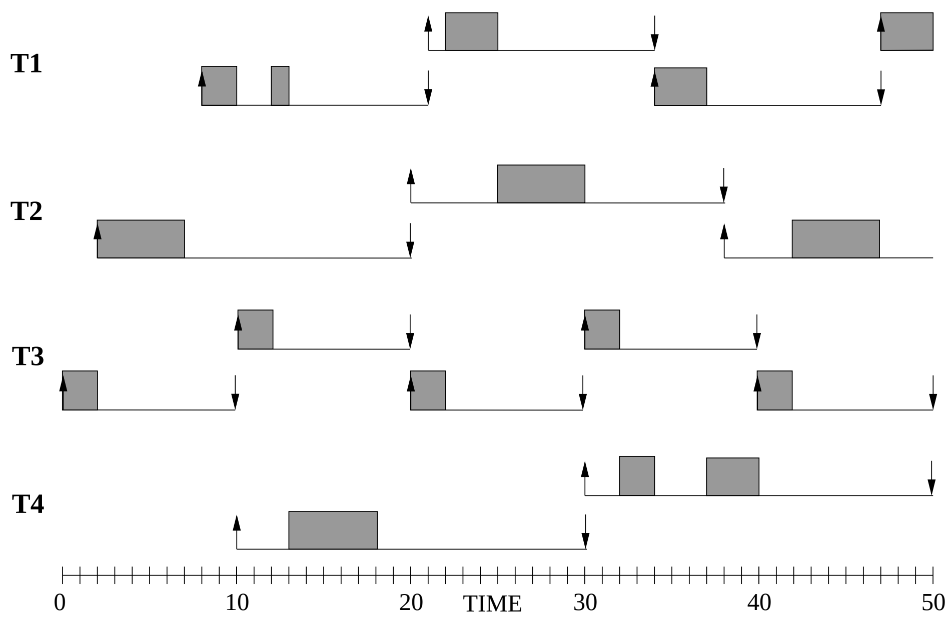


Figure 1.6: Four-task schedule with tasks $\mathbf{T1} = \mathbf{P}(8, 3, 13, 13)$, $\mathbf{T2} = \mathbf{P}(2, 5, 18, 18)$, $\mathbf{T3} = \mathbf{P}(0, 2, 10, 10)$ and $\mathbf{T4} = \mathbf{P}(10, 5, 20, 20)$. Jobs are prioritized according to the EDF policy.

previous examples produces a schedule identical to the EDF schedule shown in Figure 1.6. However, in general, policies like LLF can be problematic due to excessive context switching. Indeed, under a continuous time model, the LLF policy may require context switches to occur arbitrarily close together.

Benefits. We illustrate the benefit of this classification scheme by a simple example. In [47], Liu and Layland considered the problem of scheduling periodic tasks on a uniprocessor using the RM and EDF policies when each task's relative deadline equals its period. They proved that the RM policy was *optimal* among static-priority policies, *i.e.*, no other static-priority policy could guarantee such periodic constraints if the RM policy could not. Similarly, they proved that the EDF policy was optimal among *all* policies. Two implications follow from these results. First, job-level dynamic-priority scheduling provides no benefit in such situations. Since its use would needlessly incur additional overhead, attention should be restricted to static-priority and task-level dynamic-priority scheduling in this context. Second, the relative power of these two classes can be determined by studying the RM and EDF policies. For instance, Liu and Layland showed that any set of periodic tasks satisfying

$$\tau.u \leq 1 \tag{1.1}$$

can be successfully scheduled by EDF. Similarly, any set satisfying

$$\tau.u \leq |\tau| \cdot (2^{1/|\tau|} - 1) \tag{1.2}$$

can be successfully scheduled by RM. Since the utilization bound in (1.2) approaches $\ln 2$ ($\approx$ 0.69) from above as $|\tau|$ increases, these results suggest that the use of static-priority policies sacrifices up to 30% of the processing capacity of a uniprocessor. Such *analytical loss* (*i.e.*, loss resulting from techniques that require the use of weaker bounds during analysis) contrasts the loss caused by the increased runtime overhead suffered under dynamic-priority policies.

1.3.3 Scheduling Model

The second aspect of a scheduling policy is deciding when to switch between tasks. We refer to this aspect as the *scheduling model*. All previous examples have assumed a *fully preemptive* model, *i.e.*, a preemption occurs whenever a job with higher priority than an executing job becomes ready. In practice, delaying preemptions (and hence permitting priority inversions) can improve cache performance and can even reduce resource-sharing overhead. For instance, preempting a task that is executing a critical section lengthens the amount of time that the shared resource is unavailable to other tasks. By avoiding such preemptions, the maximum waiting time that a task experiences before gaining access to a resource can be reduced. In this subsection, we discuss the two basic classes of scheduling models.

Event-driven scheduling. Under an event-driven model, a preemption may occur in response to any event that changes task states (*e.g.*, job arrivals, job completions, suspensions, priority changes, *etc.*). Fully preemptive scheduling is a common example. Under a fully preemptive model, a check is performed each time a state change occurs to determine whether a preemption is necessary. If so, the preemption occurs immediately. The RM, EDF, and LLF policies are typically considered under this model (*e.g.*, [47]). Whenever the model is not explicitly given in this dissertation, the fully preemptive model should be assumed.

Nonpreemptive scheduling is another common example of event-driven scheduling. Under nonpreemptive scheduling, a job cannot be preempted once it has been scheduled. Hence, a job will continue to execute until it either suspends or completes. Figure 1.7 shows the previous EDF-scheduling example repeated under a nonpreemptive model. Notice that no preemptions occur at any point within the schedule.

The primary advantages of event-driven scheduling are that it is both efficient and *work-conserving*. A scheduler is said to be work-conserving if no processor idles while an unscheduled ready task exists. Efficiency follows from the fact that the scheduler is invoked only when necessary. However, this property is also a disadvantage: the timing of scheduler invocations depends on the runtime behavior of tasks and hence may be difficult to predict. Though bounds on the total number of preemptions occurring within in interval can often be derived, it may be difficult to predict which tasks will be preempted and how often.

Time-driven scheduling. Under a time-driven model, preemptions occur only at predetermined times that are independent of runtime behavior. Hence, time-driven scheduling

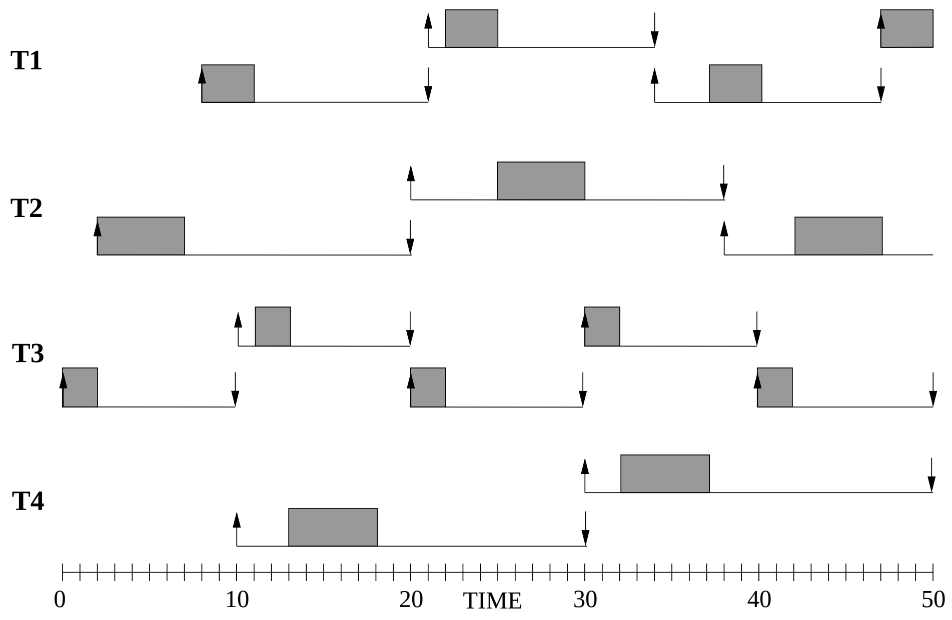


Figure 1.7: Four-task schedule with tasks $\mathbf{T1} = \mathbf{P}(8, 3, 13, 13)$, $\mathbf{T2} = \mathbf{P}(2, 5, 18, 18)$, $\mathbf{T3} = \mathbf{P}(0, 2, 10, 10)$ and $\mathbf{T4} = \mathbf{P}(10, 5, 20, 20)$. Jobs are prioritized according to the EDF policy under a nonpreemptive model.

is based on a *polling* design, *i.e.*, the system periodically checks the set of ready tasks to determine whether a preemption should occur.

The most common example of (on-line) time-driven scheduling is quantum-based scheduling. Under this approach, processor time is allocated in fixed-size chunks, called *quanta*. We denote the quantum size by Q. If a task suspends or completes before its quantum is exhausted, the processor idles for the remainder of the time, *i.e.*, unused processor time is not re-allocated.[7] This approach has the effect of dividing time into a sequence of fixed-length slots, one per quantum. In each slot, each processor can be assigned to at most one task. Figure 1.8 shows the result of applying a quantum-based model in which $Q = 3$ to the previous example. Notice that a second timeline has been provided to explicitly show how the

[7]General-purpose operating systems often employ a variant of quantum-based scheduling that does re-allocate unused processor time. Such a model is a hybrid between event- and time-driven scheduling since it contains elements of both.

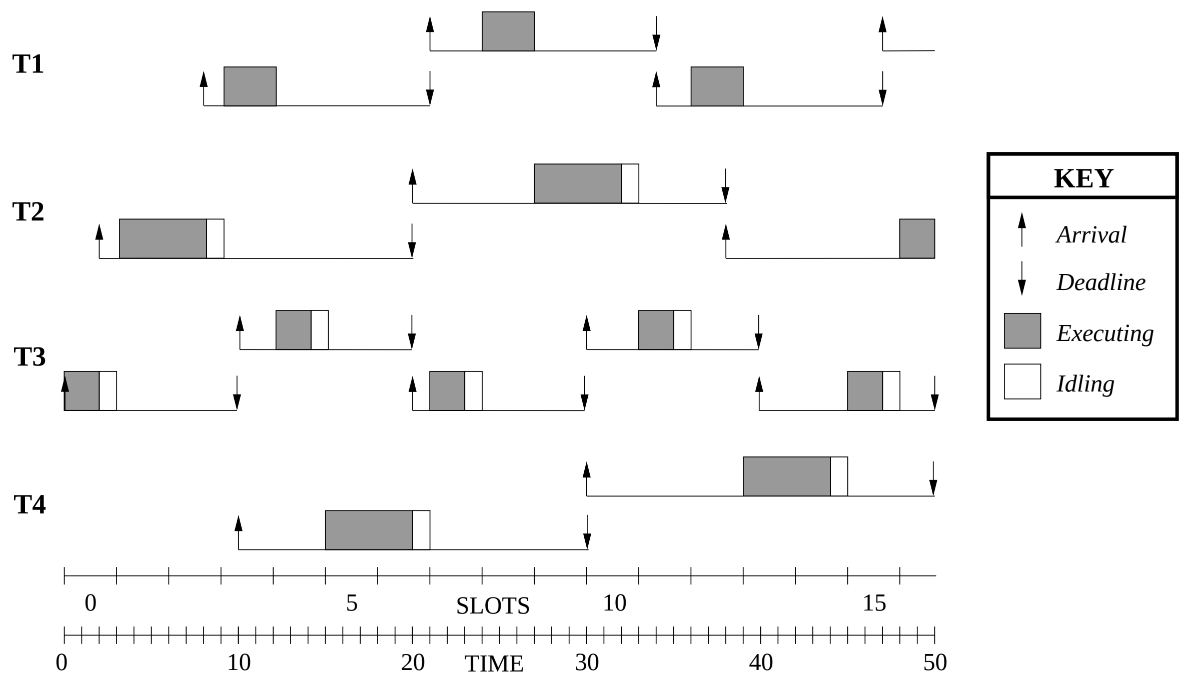


Figure 1.8: Four-task schedule with tasks $\mathbf{T1} = \mathbf{P}(8, 3, 13, 13)$, $\mathbf{T2} = \mathbf{P}(2, 5, 18, 18)$, $\mathbf{T3} = \mathbf{P}(0, 2, 10, 10)$ and $\mathbf{T4} = \mathbf{P}(10, 5, 20, 20)$. Jobs are prioritized according to the EDF policy under a quantum-based model in which $Q = 3$.

use of a quantum divides time into slots. Also notice that context switches occur only on slot boundaries, resulting in some processor idling.

The advantage of time-driven scheduling is the predictability of context switches, which facilitates analysis like that needed by real-time systems. The primary disadvantages are that processor time can be wasted by idling processors within quanta and by polling the state of the task set unnecessarily. Both of these problems are illustrated by Figure 1.8. Nine units of processor time are wasted due to idling in this example. In addition, the scheduler invocations occurring at times 6, 18, 30, and 42 are unnecessary.[8]

[8]Although job arrivals do occur at time 30, the new jobs have lower priority than the executing job. Hence, a scheduler invocation was unnecessary.

1.3.4 Validation

As explained earlier, the goal of real-time analysis is to show that all timing constraints will be met at runtime. We refer to the process of making this determination as *validation*. The following terminology is needed to discuss validation precisely:

- A task set is *schedulable* under a scheduling policy σ if the use of that policy results in a schedule in which all timing constraints are met.
- A task set is *feasible* under a set of scheduling policies Σ if there exists a policy $\sigma \in \Sigma$ such that the task set is schedulable under σ.
- A scheduling policy σ is *optimal* for the set of policies Σ if $\sigma \in \Sigma$ and any task set that is feasible under Σ is also schedulable under σ.

When Σ is not given in this dissertation, the set of all possible policies should be assumed.

Constraint strictness. Tasks can be classified according to the strictness of their timing constraints. Constraints that *must* be satisfied are referred to as *hard* constraints; tasks with such constraints are said to be *hard tasks*. A simple example of a hard task is the sample-processing task discussed in the example on page 13. If deadlines are missed, then samples can be lost, which may result in incorrect operation.

All weaker constraints are called *soft* constraints; tasks with these constraints are called *soft tasks*. This class actually represents a wide variety of constraints. One simple form of soft constraint that we consider throughout this dissertation is a *tardiness* bound. This bound specifies the maximum amount by which a task is permitted to miss its deadlines.

To motivate interest in soft constraints, again consider the sample-processing task considered on page 13. If sample processing times vary significantly, it may be difficult to ensure that all samples in a batch are processed before the next batch arrives. This guarantee can be provided at great expense by overprovisioning the processing task based on the worst-case processing time of any batch. An alternative solution is to split sample processing into two steps, each of which is handled by a different task. First, a hard task is used to obtain the samples from the external device. However, instead of processing them, this task simply stores them in a memory buffer for the second task. The second task then performs the actual processing later. As before, the period of the processing task must be selected so that the average processing rate is at least the sampling rate. However, the instantaneous rate can be permitted to fluctuate at the expense of additional buffering. Hence, it is reasonable to treat the second task as a soft task.

Validation tests. Timing guarantees are provided through schedulability and feasibility tests. As the names suggest, these tests determine whether a given task set is schedulable or feasible, respectively, under the conditions assumed by the test. Schedulability and feasibility tests fall into two basic categories: exact and sufficient. Under both categories, any system that is accepted by the test is guaranteed to meet its timing constraints. However, the categories differ in the meaning of a rejection. Under exact tests, rejection implies that a timing constraint can be violated. Under sufficient tests, rejection implies only that a timing constraint *may* be violated, *i.e.*, the test could not rule out the possibility of a violation. Hence, a trivial test that is always sufficient is one that always rejects.

The tests given in (1.1) and (1.2) are examples of exact and sufficient schedulability tests, respectively. Recall that these tests consider the use of the EDF and RM policies, respectively, under a fully preemptive model. By definition, schedulability tests for a given policy are sufficient feasibility tests for any class to which that policy belongs. If the policy is optimal for a class and its schedulability test is exact, then that test is an exact feasibility test for that class. For instance, (1.1) is an exact feasibility test for the class of task-level dynamic-priority policies.

1.4 Multiprocessor Real-time Scheduling

The previous sections summarized terminology and concepts common to processor scheduling. In this section, we focus on multiprocessor scheduling. After discussing the basic approaches (partitioning and global scheduling), we introduce Pfair scheduling, which is the focus of this dissertation. The next chapter discusses Pfair scheduling in detail.

1.4.1 Partitioning

Under *partitioning*, the task set is divided into up to M subsets, each of which is statically assigned to a unique processor. Each task can execute only on the processor to which it is assigned, *i.e.*, migration is forbidden. As a result, processors can be scheduled independently using uniprocessor policies. Partitioning effectively reduces the multiprocessor scheduling problem to multiple uniprocessor scheduling problems.

Advantages. The primary advantages of partitioning are improved runtime performance and lower runtime overhead. These benefits stem from several sources. First, uniprocessor techniques are often significantly simpler and more efficient than their multiprocessor counterparts. Hence, their use results in less runtime overhead. Second, partitioning implicitly respects *processor affinity* by preventing migration. Informally, processor affinity refers to the property that tasks tend to execute quicker when not migrated. This improvement is typically the result of better cache performance.

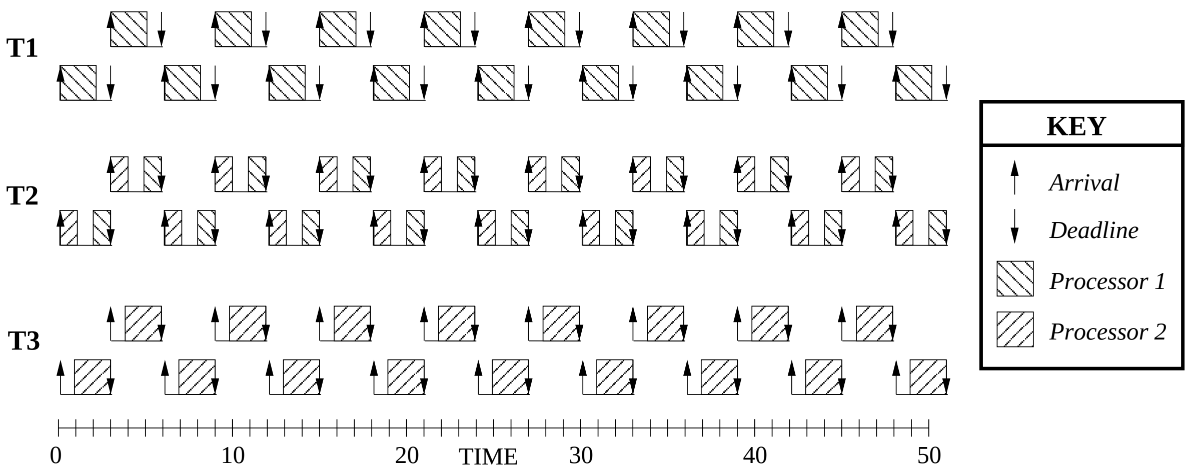


Figure 1.9: Three-task, two-processor schedule in which all tasks are described by $\mathbf{P}(0,2,3,3)$. Migration is required to meet all constraints.

Disadvantages. Partitioning has two primary flaws. The first flaw is its inherent suboptimalilty when scheduling periodic tasks. A well-known example is a two-processor system consisting of the three $\mathbf{P}(0,2,3,3)$ tasks. Since each task's utilization is $\frac{2}{3}$, no two tasks can be bound to the same processor without overutilizing it, *i.e.*, $\frac{2}{3}+\frac{2}{3}>1$. Hence, the partitioning approach cannot schedule this task set on 2 processors. However, as shown in Figure 1.9, all timing constraints can be met on 2 processors when migration is allowed. Therefore, this task system is feasible. The second flaw is that assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense [25]. Hence, assignment must be done either off-line or using a suboptimal heuristic.

1.4.2 Global Scheduling

Under *global scheduling*, all tasks are stored in a single set and all processors are scheduled by the same scheduler. Hence, the highest-priority task is always selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled.

Advantages. The primary advantage of global scheduling is that optimal scheduling is possible since migration is permitted. In addition, global management of data structures can facilitate recovery following a processor fault. The reason for this is that global scheduling requires the ability to move data between processors quickly and efficiently. The mechanism that provides this capability may also permit data to be retrieved from a faulty processor's local memory. Of course, fault recovery still involves many well-studied and difficult challenges, including reliable fault detection and the need for election protocols.[9] Further study is needed to determine what benefits global scheduling can provide in this area.

Disadvantages. Unfortunately, producing a conventional global scheduler suitable for real-time systems has proved to be a daunting task. Dhall and Liu demonstrated that fully preemptive global scheduling of periodic tasks using either EDF or RM priorities can result in deadline misses whenever $\tau.u > 1$ [23]. Furthermore, optimal scheduling requires job-level dynamic priorities. Recall that such priorities produce the most runtime overhead of the three classes discussed earlier. The need for these priorities is implied by the example considered in Figure 1.9. At time 0, $\mathbf{T2}(1)$ has priority over $\mathbf{T3}(1)$. However, at time 1, $\mathbf{T3}(1)$ must preempt $\mathbf{T2}(1)$, which implies a change in at least one of their priorities. Finally, under global scheduling, task information must be stored in shared data structures, which must be updated each time a task's state changes. Hence, this information has the potential to become a point of heavy contention in the operating system. In addition, accesses to this information must be synchronized to avoid corruption and hence may incur significant overhead.

[9]When the faulty processor is responsible for coordinating the operation of other processors, this duty must be re-assigned to one of the remaining processors. An election protocol identifies the remaining processors and determines which should assume these responsibilities.

1.4.3 Pfair Scheduling

Proportionate-fair (or *Pfair*) scheduling [11] is a global-scheduling approach proposed as a means for optimally scheduling periodic tasks on a multiprocessor when each task's deadline equals its period. Pfair scheduling is unconventional in that it focuses on ensuring an approximately steady rate of execution (relative to Q) for each task. Consequently, it is said to be a *rate-based* approach. In addition, Pfair scheduling uses a quantum-based model.

Under Pfair scheduling, each task T is assigned a weight $T.w$ in the range $(0, 1]$ that represents the fraction of a single processor entitled to that task. Quanta are then allocated to these tasks in proportion to their weights so that each task receives approximately its fair share (*i.e.*, a fraction $T.w$ of a single processor's time) at all times. Due to this notion of fairness, Pfair scheduling is said to be a *fair* policy.

When Q is sufficiently small, periodic tasks can be scheduled using a Pfair policy by letting $T.w = T.u$ for each task. Figure 1.10 shows the result of applying a Pfair policy to the previous example when $Q = 1$. (We explain how Pfair scheduling works in greater detail in the next chapter.) Notice that the Pfair policy distributes each task's allocation evenly over time, resulting in approximately uniform execution rates.

Although Pfair scheduling has been proved optimal [11], prior work has been limited to the consideration of independent tasks. In addition, some aspects of Pfair scheduling are unconventional. First, this approach uses the quantum-based model described earlier on page 20. As illustrated in Figure 1.8, this model has a disadvantage in that suspensions and completions can lead to processor idling. In Figure 1.10, idling is avoided by using very short slots. Unfortunately, scheduling and context-switching overheads impose a practical limit on how short slots can be. Second, executing tasks at uniform rates can produce a high

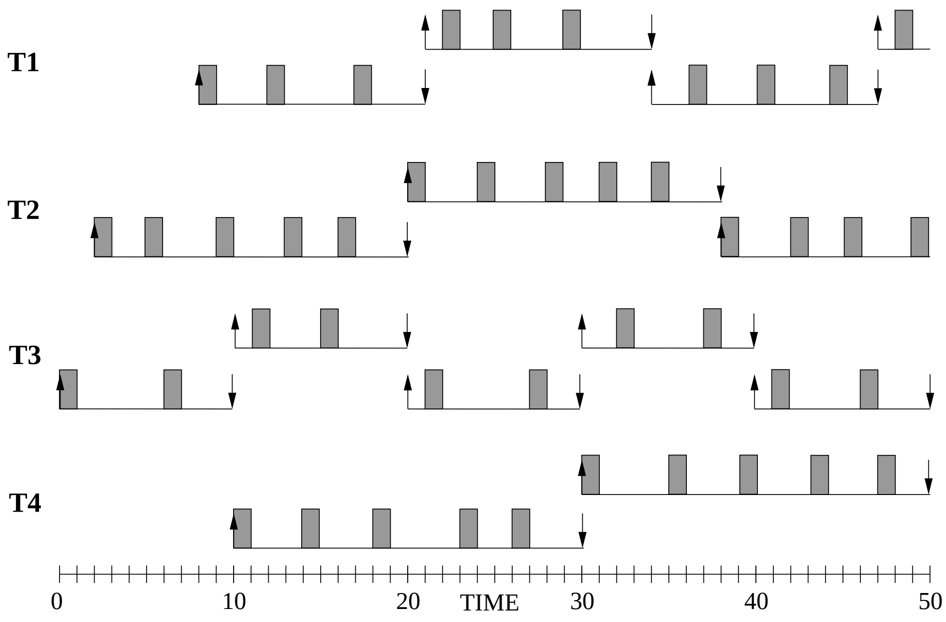


Figure 1.10: Four-task schedule with tasks $\mathbf{T1} = \mathbf{P}(8, 3, 13, 13)$, $\mathbf{T2} = \mathbf{P}(2, 5, 18, 18)$, $\mathbf{T3} = \mathbf{P}(0, 2, 10, 10)$ and $\mathbf{T4} = \mathbf{P}(10, 5, 20, 20)$. Tasks are prioritized by a Pfair policy using $Q = 1$.

frequency of context switches. For instance, compare the number of switches in Figure 1.10 to that in Figure 1.6. Third, migrations are unrestricted and may occur frequently under Pfair scheduling. Recall that optimal scheduling requires migration, as evidenced by Figure 1.9. However, since migration induces additional overhead, it is still desirable to avoid it whenever possible. The above problems combined have led to questions regarding the practicality of Pfair scheduling. Our work addresses many of these concerns, as explained below.

1.5 Research Goals

The goal of this dissertation is to enable the use of Pfair scheduling in real systems by developing techniques for supporting task synchronization and for reducing various overheads, such as context-switching and migration overheads. Our interest in Pfair scheduling from several observations, the most compelling of which are given below.

First, Pfair scheduling is a relatively new approach and, unlike partitioning, has the potential to improve substantially in the future. Many problems currently posed by Pfair scheduling are likely not inherent to that approach, but rather are side effects of the lack of development. Indeed, the unconventional aspects of Pfair scheduling (explained above) suggest that unconventional, but effective, solutions to well-understood problems may be possible. Hence, the study of this approach has the potential to not only produce an efficient real-time multiprocessor design, but also to provide new insights into the fundamental trade-offs present in all multiprocessor operating systems. On the other hand, partitioning has been well-studied, which suggests that most (if not all) of the known limitations are fundamental. It is unlikely that further attention will yield substantial improvements.

Second, growing interest in web-based and multimedia applications is increasing the demand for rate-based scheduling and QoS guarantees. Recall, for instance, the web server example discussed at the start of this chapter. Research into providing such guarantees under Pfair scheduling has already produced promising results [2, 61, 62, 63], and suggests that further attention is warranted.

Finally, although research into other global-scheduling approaches has produced interesting results, none of these alternatives approach optimality. Much like EDF scheduling on a uniprocessor, Pfair scheduling currently provides much better analytical bounds than rival approaches, but at the cost of more runtime overhead. However, it is presently unclear whether this overhead negates the benefits of Pfair scheduling. Until these overheads can be quantified for Pfair scheduling and its alternatives, there is no reliable way to determine which approach will produce the best results in practice.

Thesis statement. The thesis of this dissertation is:

tasks can be efficiently synchronized in Pfair-scheduled systems and overhead due to common system events, such as context switches and migrations, can be reduced.

Although proving the above thesis will not definitively prove the practicality of Pfair scheduling, techniques that achieve the goals given in the statement are essential to demonstrating that novel solutions do exist to many of the problems posed by this approach.

1.6 Contributions

The specific contributions of this dissertation are summarized in the subsections that follow.

1.6.1 Hierarchal Scheduling

Our first contribution is in the area of hierarchal scheduling. Under this approach, a collection of tasks are grouped together and scheduled as a single entity. When the entity is scheduled, one of the tasks in the collection is selected to execute according to an internal policy.

Hierarchal scheduling provides two primary benefits under Pfair scheduling. First, by using an event-driven policy within the group (such as fully preemptive EDF), a processor assigned to the group will idle only when no tasks in the group are ready. Second, assigning tasks to the same group has the effect of serializing their execution. By preventing parallel execution, contention for shared resources can be reduced.

The key problem when using hierarchal scheduling is selecting a weight for the group such that all timing constraints of group members will be met. We solve this problem by presenting a flexible analytical framework for deriving weight restrictions based on the task requirements and the internal scheduling policy. We also present an efficient algorithm for computing a weight that satisfies these restrictions.

1.6.2 Task Synchronization

Our second major contribution is support for resource sharing under Pfair scheduling. We consider both lock-free and lock-based synchronization. Recall that a disadvantage of Pfair's quantum-based model is that suspensions can lead to processor idling. For this reason, lock-free techniques are better-suited to Pfair systems than lock-based synchronization, which introduces the potential for blocking suspensions. Lock-free algorithms optimistically attempt an operation; if interference occurs, then the operation is simply retried until successful. Unfortunately, efficient lock-free algorithms exist only for simple software-implemented objects, like buffers and queues. Hence, lock-based synchronization is still needed to synchronize access to hardware devices and complex objects.

For lock-free synchronization, we show how quantum-based scheduling enables tight bounding of the number of retries required by a lock-free operation. We also discuss techniques for reducing the worst-case execution times of lock-free algorithms, including the use of hierarchal scheduling to reduce the number of retries.

For lock-based synchronization, we explore several alternatives. First, we consider the use of *inheritance* to speed critical-section executions. Under this approach, a task can temporarily adopt the characteristics of other tasks that it blocks, such as priorities. Though inheritance is very effective under fully preemptive uniprocessor scheduling [54], we note several disadvantages to using it under Pfair scheduling. Second, we present an approach based on the observation that short critical sections will never be preempted if started sufficiently early within a quantum. Such a guarantee is quite powerful because preemption of a critical section can increase the length of time that a resource is unavailable by several orders of magnitude. Finally, to support longer critical sections, we present a simple server-based approach, under which a lock server executes critical sections remotely on behalf of the requesting task. We show how to account for resource-sharing contention under each of these last two approaches when determining the schedulability of a task set.

1.6.3 Staggered Scheduling

Our third major contribution is to consider the compatibility of Pfair scheduling with a symmetric multiprocessor (SMP) architecture. Recall from Figure 1.10 that context switches and migrations an occur frequently under Pfair scheduling. To keep these overheads reasonable, tightly coupled processing is needed, *i.e.*, it must be possible to move data between processors and between a processor and memory quickly. SMPs provide this by using a central shared memory, local processor caches, and a high-speed shared bus.

Unfortunately, we present evidence that Pfair scheduling actually *promotes* bus contention on SMPs and that this contention can greatly degrade the performance of the tasks. Specifically, tasks may need to reload data into local caches at the start of each quanta, resulting in a period of increased bus traffic. Under Pfair scheduling, quanta begin simultaneously on all processors. Hence, these bursts of traffic are forced to also occur simultaneously, resulting in heavy contention for the bus at the start of each quantum.

For such bus-based architectures, we propose an alternate *staggered* model for Pfair scheduling. In this model, the start of the first slot on each processor is slightly delayed relative to the starting point on the preceding processor. In addition to reducing bus contention, this model allows scheduler overhead to be distributed across processors without requiring the use of a parallel scheduling algorithm. Hence, scheduling overhead can also be reduced. To enable the use of this model, we present an efficient scheduling algorithm for it and discuss how existing techniques and results can be adapted for use under staggering.

1.7 Organization

The rest of the dissertation is organized as follows. Chapter 2 introduces notation and summarizes prior work on Pfair scheduling. Chapter 3 presents an overview of a Pfair system; simple results and results that follow easily from prior work are explained in this chapter. In Chapter 4, we present our work on supporting hierarchal scheduling under a global Pfair scheduler. In Chapters 5 and 6, lock-free and lock-based synchronization, respectively, are discussed. Chapter 7 then considers deploying Pfair scheduling on a symmetric multiprocessor. Chapter 8 concludes with a discussion of remaining challenges.

CHAPTER 2

Pfair Scheduling

In this chapter, Pfair scheduling and its variants are formally defined, and prior work is summarized. This chapter serves two purposes. First, it introduces basic capabilities of Pfair scheduling that will be act as a foundation for our work. Second, the summary of prior work is intended to motivate interest in this approach by demonstrating its depth.

We begin by introducing the Pfairness timing constraint in Section 2.1. Scheduling policies are then discussed in Section 2.2. Additional task models not discussed in these two sections are discussed in Section 2.3, followed by a brief summary of other related work in Section 2.4.

2.1 Basics of Pfair Scheduling

This section explains the concepts underlying Pfair scheduling.

2.1.1 The Pfairness Constraint

Under Pfair scheduling, each task T is characterized by a *weight* $T.w$ in the range $(0, 1]$. Conceptually, $T.w$ is the fraction of a single processor to which T is entitled.[1] T is said to be *light* if $T.w < \frac{1}{2}$, and *heavy* otherwise. Under the task model proposed by Baruah, Cohen,

[1]This approach assumes an *identical* multiprocessor, *i.e.*, processors share a common architecture and speed.

Plaxton, and Varvel [11], tasks are persistent, become ready at time 0, and are *not* allowed to suspend. We let $T = \mathbf{PF}(w)$ to denote a Pfair task with $T.w = w$.

Pfair scheduling uses a quantum-based model. As in the previous chapter, we let Q denote the quantum size. (In an ideal system like that considered in [11], $Q = 1$.) To simplify the presentation, the slot length will be used as the basic time unit for all time parameters. Formally, the interval of time in which the i^{th} quanta is allocated is called *slot* $i - 1$. Since scheduling begins simultaneously on all processors, slot i spans the time interval $[i, i+1)$ across processors. For instance, in Figure 2.1(b), task B is scheduled in slot 3, which corresponds to the time interval $[3, 4)$. (The rest of this figure is considered in detail below.) In each slot, each processor can be assigned to at most one task, and each task can be assigned at most one processor. Task migration is allowed. The sequence of scheduling decisions made in these slots defines a *schedule*.

In a *fluid* schedule, each task T receives exactly its designated share of a processor ($T.w$) at all times. Under ideal circumstances (*i.e.*, when $Q = 1$), each processor provides $t_2 - t_1$ units of processor time over the interval $[t_1, t_2)$. When per-slot overhead exists, only Q (< 1) units of processor time are made available to tasks in each slot. Ideally, this processor time would be provided at a uniform rate throughout each slot. Hence, each processor allocates $(t_2 - t_1) \cdot Q$ units of processor time to tasks over the interval $[t_1, t_2)$ in an ideal schedule, and T's share of this time is given by

$$\mathit{fluid}(T, t_1, t_2) = T.w \cdot (t_2 - t_1) \cdot Q. \tag{2.1}$$

Pfair scheduling tracks the allocation of processor time in a fluid schedule; deviation is formally expressed as $\mathit{lag}(T,t)$, which is defined below.

$$\mathit{lag}(T,t) = \mathit{fluid}(T,0,t) - \mathit{received}(T,0,t) \tag{2.2}$$

In the above equation, $\mathit{received}(T,t_1,t_2)$ denotes the amount of processor time received by T over $[t_1,t_2)$. Using this notion of lag, the *Pfairness* timing constraint for a task T can be formally defined as shown below.

$$\text{for all } t,\ |\,\mathit{lag}(T,t)\,| < Q \tag{2.3}$$

Informally, T's allocation must always be within one quantum of its fluid allocation. When tasks are present at time 0 and always remain ready (as assumed in [11]), the above constraint yields the property given below.

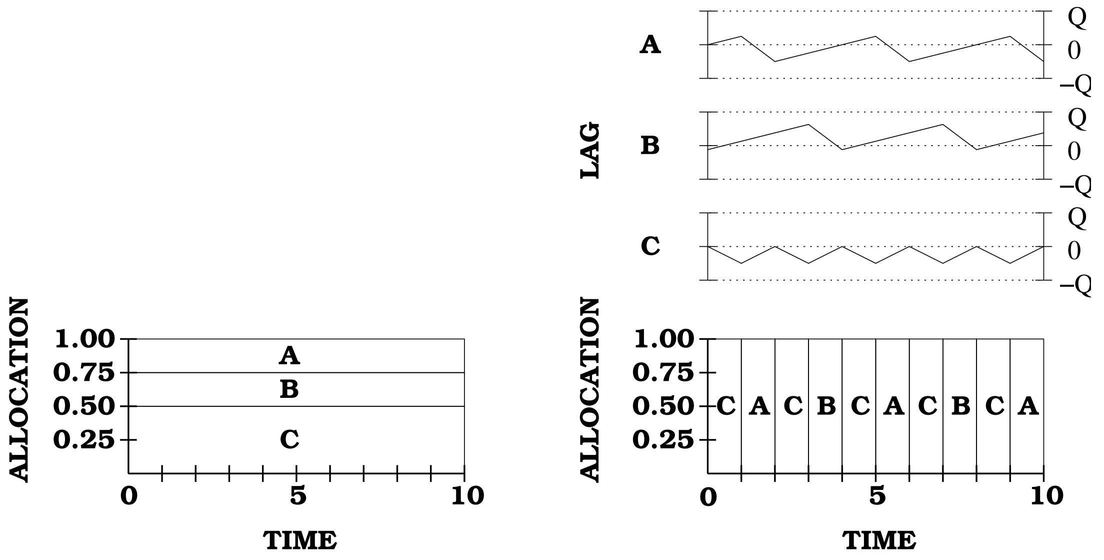
Pfairness (Pf): T receives either $\left\lfloor \frac{\mathit{fluid}(T,0,t)}{Q} \right\rfloor$ or $\left\lceil \frac{\mathit{fluid}(T,0,t)}{Q} \right\rceil$ quanta over $[0,t)$.

Figure 2.1(a) shows fluid and Pfair uniprocessor schedules for a task set containing three Pfair tasks: $A = \mathbf{PF}\left(\frac{1}{4}\right)$, $B = \mathbf{PF}\left(\frac{1}{4}\right)$, and $C = \mathbf{PF}\left(\frac{1}{2}\right)$. In Figure 2.1(b), changes in each task's lag are shown across the top of the schedule.

Relationship to the periodic constraint. As explained earlier, Pfair scheduling was initially proposed as a means to optimally schedule synchronous periodic task systems on a multiprocessor [11]. Baruah *et al.* observed the following property:[2]

Periodicity (P): When a and b are integers satisfying $b \geq a > 0$, the timing constraints of a periodic task $U = \mathbf{P}(0, aQ, b, b)$ can be satisfied by executing it

[2]Property P slightly generalizes the property discussed in [11], which only considered the $Q = 1$ case.



(a) Fluid scheduler **(b) Lag-based scheduler**

Figure 2.1: Sample schedules for $\tau = \{A, B, C\}$ where $A = \mathbf{PF}\left(\frac{1}{4}\right)$, $B = \mathbf{PF}\left(\frac{1}{4}\right)$, and $C = \mathbf{PF}\left(\frac{1}{2}\right)$. **(a)** Schedule produced by a fluid scheduler. **(b)** Schedule produced by a Pfair lag-based scheduler.

within the processor time allocated to a Pfair task $T = \mathbf{PF}(\frac{a}{b})$.

To prove Property P, assume T's timing constraints are satisfied. By (2.1), $\mathit{fluid}(T, 0, nb) = \left(\frac{a}{b}\right) \cdot (nb) \cdot Q = naQ$ for all non-negative integers n. Since all such points fall on slot boundaries, $Q \mid \mathit{received}(T, 0, nb)$ (*i.e.*, $\mathit{received}(T, 0, nb)$ is a multiple of Q) follows trivially since processor time is allocated in units of Q. Hence, $Q \mid \mathit{lag}(T, nb)$ holds also by (2.2). By (2.3), $Q \mid \mathit{lag}(T, nb) \Rightarrow \mathit{lag}(T, nb) = 0$, and hence, $\mathit{received}(T, 0, nb) = \mathit{fluid}(T, 0, nb) = naQ$. Therefore, $kaQ - (k-1)aQ = aQ$ units of processor time are allocated to T over the $[(k-1)b, kb)$, which is sufficient to service U's k^{th} job. Property P follows.

2.1.2 Feasibility

Baruah *et al.* [11] showed that a schedule satisfying (2.3) exists on M processors for any set τ of Pfair tasks if and only if the following condition holds.

$$\sum_{T \in \tau} T.w \leq M \tag{2.4}$$

2.1.3 Using Deadlines to Support Rates

Although the Pfairness constraint is rate-based rather than deadline-based, the use of a quantum-based model permits an equivalent deadline-based view. Specifically, tasks can be decomposed into quantum-length *subtasks*, each of which is constrained by (2.3) to execute within a specific interval of slots, called the subtask's *window*. Although this view of Pfair scheduling was suggested by the analysis presented in [11, 12], it was not formalized until [5].

Subtasks and windows. We let T_i denote the i^{th} subtask of task T. Also, we let $\omega(T_i)$ denote the window of subtask T_i. Figure 2.2(a) shows the window within which each subtask of the task $\mathbf{PF}\left(\frac{3}{10}\right)$ must execute. For example, $\omega(T_2) = [3, 7)$ in Figure 2.2(a).

$\omega(T_i)$ extends from T_i's *pseudo-release*, denoted $r(T_i)$, to its *pseudo-deadline*, denoted $d(T_i)$. (The "pseudo" prefix avoids confusion with job releases and deadlines. This prefix will be omitted when the proper interpretation is clearly implied.) In Figure 2.2(a), $r(T_2) = 3$ and $d(T_2) = 7$. These values can be computed using the following equations [5].

$$r(T_i) = \left\lfloor \frac{i-1}{T.w} \right\rfloor \tag{2.5}$$

$$d(T_i) = \left\lceil \frac{i}{T.w} \right\rceil \tag{2.6}$$

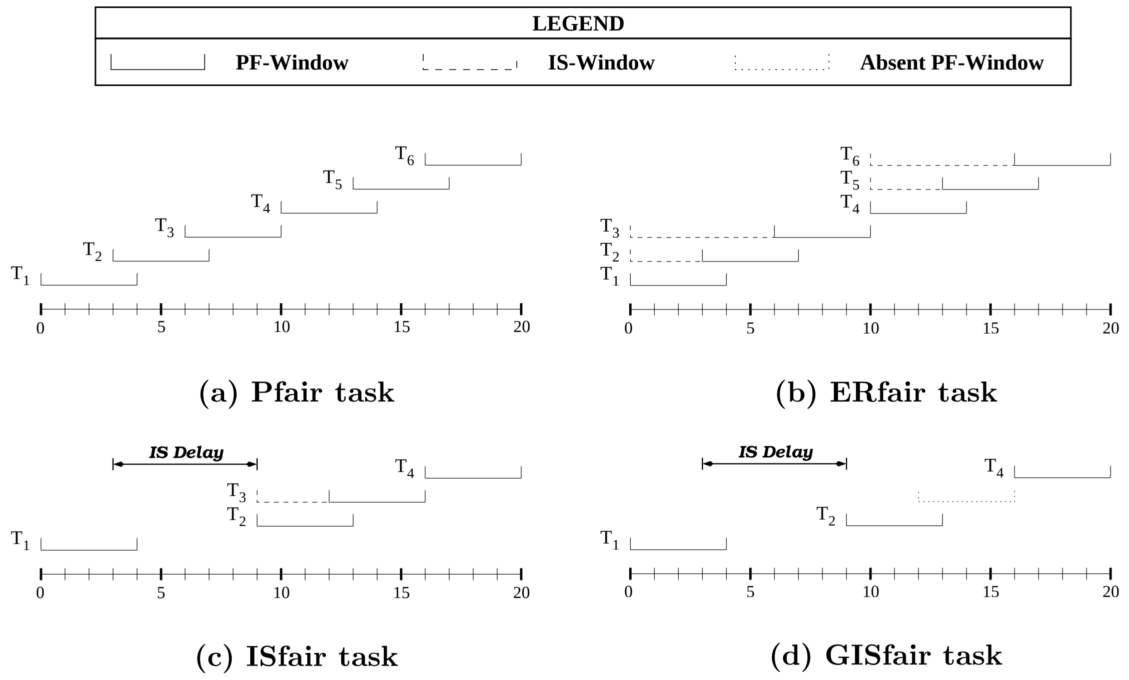


Figure 2.2: The windowing for a task with weight $\frac{3}{10}$ is shown under the Pfair task model and its variants. **(a)** Normal windowing under the Pfair task model. **(b)** Early releasing has been used so that each grouping of three subtasks becomes eligible simultaneously. (In reality, each subtask will not be eligible until its predecessor is scheduled.) **(c)** Windows appear as in inset (b) except that T_2's release is now preceded by an intra-sporadic delay of six slots. (T_5 and T_6 are not shown.) **(d)** Windows appear as in inset (c) except that T_3's window is now absent.

A schedule satisfies Pfairness if and only if each subtask T_i executes in the interval $[r(T_i), d(T_i))$. It is straightforward to show that $|\omega(T_i)|$ equals either $\left\lceil \frac{1}{T.w} \right\rceil$ or $\left\lceil \frac{1}{T.w} \right\rceil + 1$ (see [5]).

To distinguish between the cases in which subtask windows overlap and do not overlap, a *successor bit* (or *b-bit*) $b(T_i)$ is associated with each subtask T_i, and is defined by $b(T_i) = d(T_i) - r(T_{i+1})$ [5]. Informally, $b(T_i)$ is the number of slots by which T_i overlaps with T_{i+1}.

Rational weights and cycles. Anderson and Srinivasan noted that the use of a rational weight results in a repeating, symmetric series of windows [5]. We refer to each occurrence of a series as a *cycle*. For instance, in Figure 2.2(a), each cycle consists of three windows that span ten slots. Two cycles are shown: the first spans $[0, 10)$ while the second spans $[10, 20)$.

Cycles of a task T are defined by two parameters: the per-cycle execution requirement $\mathcal{E}(T)$ and the cycle period $\mathcal{P}(T)$. Given that $T.w = \frac{a}{b}$, where a and b are integers satisfying $b \geq a > 0$, $\mathcal{E}(T)$ and $\mathcal{P}(T)$ are defined as follows [5]:

$$\mathcal{E}(T) = \frac{a}{\mathsf{gcd}\,(a,b)} \qquad\qquad \mathcal{P}(T) = \frac{b}{\mathsf{gcd}\,(a,b)}$$

For instance, in Figure 2.2(a), $\mathcal{E}(T) = \frac{3}{\mathsf{gcd}(3,10)} = 3$ and $\mathcal{P}(T) = \frac{10}{\mathsf{gcd}(3,10)} = 10$.

Interest in $\mathcal{E}(T)$ and $\mathcal{P}(T)$ stems from the fact that these values define the placement of disjoint windows. More formally, these parameters satisfy Property WC, shown below.

Window Cycles (WC):

$$b(T_i) = \begin{cases} 0 & \text{, if } \mathcal{E}(T) \mid i \\ 1 & \text{, otherwise} \end{cases}$$

Informally, Property WC states that the last subtask in each cycle does not overlap with its successor, while all other subtasks within the cycle do. Hence, cycles do *not* overlap.

2.2 Scheduling Policies

In this section, we summarize scheduling policies proposed for the task model discussed in the previous section. Since Pfair scheduling currently requires the use of a quantum-based model, it remains only to consider the prioritization of subtasks.

2.2.1 EPDF

Due to the effectiveness of the EDF prioritization on uniprocessors, it is natural to consider applying such a policy to subtask deadlines (*i.e.*, to pseudo-deadlines). To avoid confusion, this prioritization is called *earliest-pseudo-deadline-first* (EPDF).

Unfortunately, Srinivasan and Anderson proved that using EPDF alone results in optimal scheduling only in systems of at most two processors [6, 63]. However, using EPDF along with appropriate tie-breaking rules *can* result in optimal scheduling. Indeed, all known optimal policies (described below) are extensions of the EPDF policy.

Despite its suboptimality, Srinivasan and Anderson [63] observed that the EPDF policy may be preferable in some cases because it eliminates the overhead of maintaining tie-breaking information. An additional benefit suggested by them is the ability to handle weight changes more seamlessly than is possible under the known optimal policies. To support EPDF scheduling in soft real-time systems, they derived the following parameterized condition,[3] which ensures a tardiness bound of k slots when $M \geq 2$.

$$\mathsf{maxsum}_{M-1}\,\{\; T.w \;|\; T \in \tau \;\} \leq \frac{kM+1}{k+1} \tag{2.7}$$

Later, Devi and Anderson presented an improved condition in the form of a per-task weight restriction of $\frac{M(k+1)-1}{M(k+2)-2(k+1)-1}$ [21].[4]

In (2.7) given above, $\mathsf{maxsum}_m\,\{a_1, \ldots, a_n\}$ denotes the maximum value produced by summing $\min(m, n)$ elements from the multiset $\{a_1, \ldots, a_n\}$. The following examples illustrate this shorthand notation:

- $\mathsf{maxsum}_2\{2, 2, 1\} = \max\{2+2, 2+1, 2+1\} = 4$;
- $\mathsf{maxsum}_4\{2, 2, 1\} = \max\{2+2+1\} = 5$;

[3]A slightly improved, but more complex, condition is presented in [63].

[4]This restriction is a generalization of that presented in [22].

- $\mathsf{maxsum}_0\{2, 2, 1\} = \max\{0\} = 0$.

This notational convention will be used throughout this dissertation.

Figure 2.3(a) shows a five-task, two-processor schedule produced by the EPDF policy. In this schedule (and the others shown in that figure), priority ties not resolved by the policy are broken by favoring the lower-weight task. In slot 0, T_1, U_1, V_1, W_1, and X_1 are eligible with deadlines at times 2, 3, 3, 4, and 4, respectively. Since U_1 and V_1 have equal EPDF priorities, the tie is broken in favor of V. Hence, T_1 and V_1 are scheduled in the slot, as shown.

2.2.2 PF

The PF policy, proposed by proposed by Baruah, Cohen, Plaxton, and Varvel [11], was the first combination of EPDF and tie-breaking rules to be proved optimal. Under PF, the relative priority of two subtasks is determined by applying the following rules in the stated order. (The shorthand $A \prec B$ indicates that subtask A has priority over subtask B.)

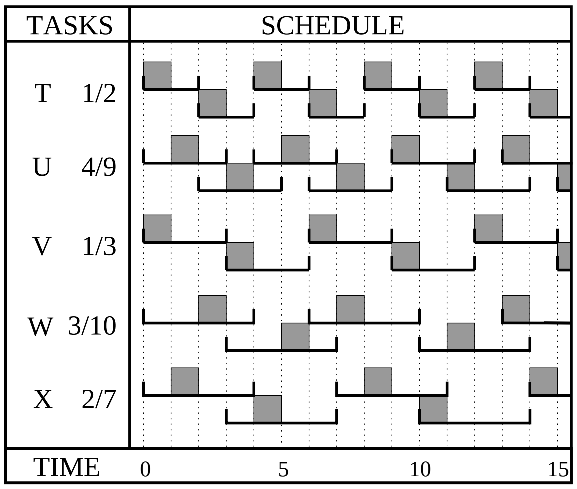
Rule #1 (EPDF): $T_i \prec U_j$ if $d(T_i) < d(U_j)$; $T_i \succ U_j$ if $d(T_i) > d(U_j)$.

Rule #2 (Overlap Bit): $T_i \prec U_j$ if $b(T_i) > b(U_j)$; $T_i \succ U_j$ if $b(T_i) < b(U_j)$. If $b(T_i) = b(U_j) = 0$ (*i.e.*, both subtasks end cycles), T_i and U_j can be prioritized arbitrarily.

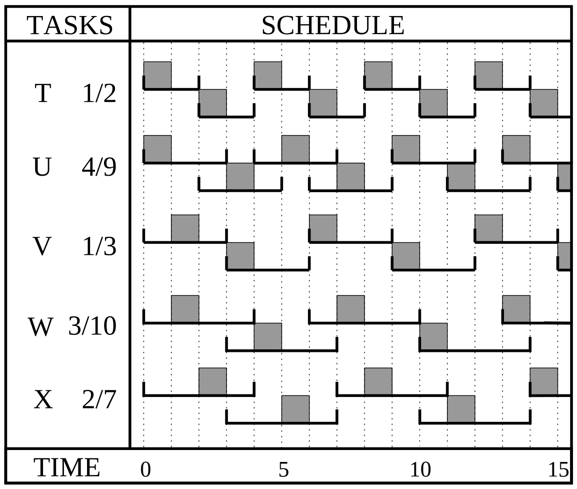
Rule #3 (Recursive): $T_i \prec U_j$ if $T_{i+1} \prec U_{j+1}$; $T_i \succ U_j$ if $T_{i+1} \succ U_{j+1}$.

Though optimal, the PF policy is inefficient due to its recursive third rule.

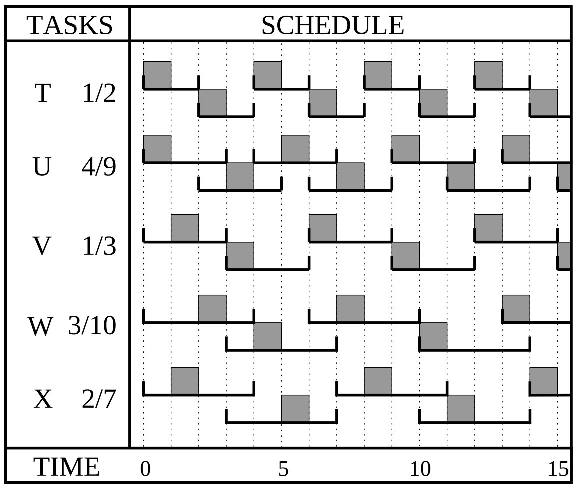
Figure 2.3(b) shows the schedule produced by the PF policy. In slot 1, V_1, W_1, and X_1 are eligible with deadlines at times 3, 4, and 4, respectively. Since W_1 and X_1 tie in both Rules #1 and #2, the recursive rule must be applied. W_2 and X_2 also tie in Rule #1. However, Rule #2 resolves the tie in favor of W_2. Because of this, Rule #3 breaks the tie of W_1 and X_1 in favor of W, as shown.



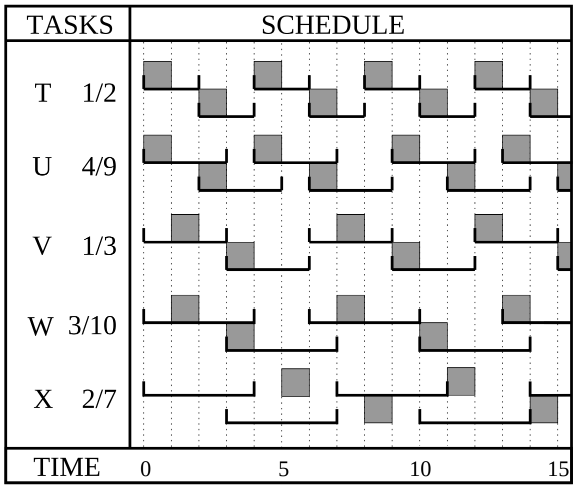
(a) EPDF Schedule



(b) PF Schedule



(c) PD/PD^2 Schedule



(d) WM Schedule

Figure 2.3: Schedules produced by the **(a)** EPDF, **(b)** PF, **(c)** PD or PD^2, and **(d)** WM policies. Priority ties not resolved by the policies are broken in favor of the lower-weight task.

2.2.3 PD

Baruah, Gehrke, and Plaxton [12] later proposed a second optimal policy, called PD, that produces less per-slot overhead than the PF policy. Under PD, the recursive Rule #3 shown above is replaced by three rules that can be applied in constant time. Since the PD^2 policy described next is a simplified version of PD (*i.e.*, some of PD's tie-breaking rules proved to be unnecessary), we forgo further discussion of the PD policy.

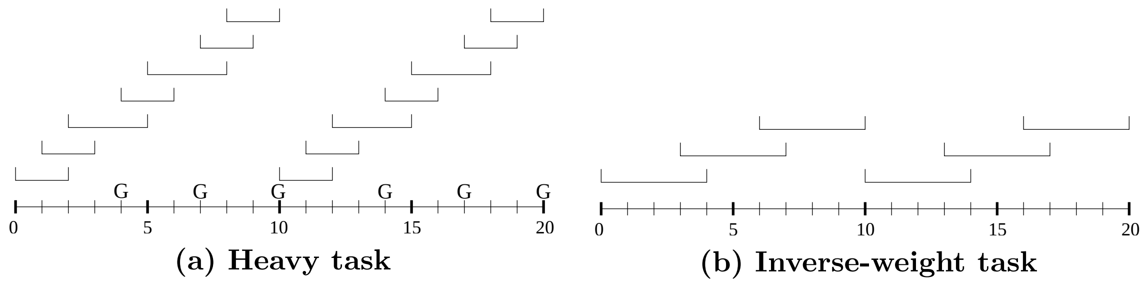


Figure 2.4: Window positions are shown for **(a)** a heavy task $\mathbf{PF}\left(\frac{7}{10}\right)$ and **(b)** the corresponding inversely-weighted light task $\mathbf{PF}\left(\frac{3}{10}\right)$. Note that group deadlines of the heavy task, each of which is denoted by a G, correspond to the pseudo-deadlines of the light task.

2.2.4 PD^2

In [7], Anderson and Srinivasan proved that two of the PD rules were extraneous. The resulting policy was called PD^2 due to its use of two tie-breaking rules. PD^2 is the most efficient of the known optimal policies. Hence, we assume its use unless otherwise stated.

PD^2 replaces PF's recursive Rule #3 with a rule that compares the *group deadlines* of subtasks in constant time. Formally, the group deadline of subtask T_i, denoted $D(T_i)$, is the earliest slot boundary at or after $d(T_i)$ at which either no subtask deadline occurs or a cycle ends. For tasks satisfying either $T.w \leq \frac{1}{2}$ or $T.w = 1$, the group deadline is set to zero. The group deadline for tasks satisfying $\frac{1}{2} < T.w < 1$ can be determined by considering the deadlines of an inverse-weight task, *i.e.*, $\mathbf{PF}(1 - T.w)$, as illustrated in Figure 2.4. Given this definition, PD^2's third rule is as shown below.

Rule #3 (Group Deadline): $T_i \prec U_j$ if $D(T_i) > D(U_j)$; $T_i \succ U_j$ if $D(T_i) < D(U_j)$.

A more detailed description of the PD^2 policy can be found in [7] and subsequent papers [5, 6, 61, 62]. For our purposes, it is sufficient to know that PD^2 is an optimal Pfair policy, that subtask priorities can be calculated and compared in constant time, and that the per-slot time complexity of PD^2 is $O(M \log |\tau|)$.

Figure 2.3(c) shows the schedule produced by the PD^2 policy. (This schedule would also be produced by the PD policy.) In slot 1, V_1, W_1, and X_1 are eligible with deadlines at times 3, 4, and 4, respectively. Since W_1 and X_1 tie in both Rules #1 and #2, the group deadlines must be compared. However, since both tasks are light, the group deadlines are both zero. Hence, these subtasks tie in Rule #3 as well. As a result, the tie is broken in favor of X.

2.2.5 WM

As with job scheduling, Pfair scheduling policies can be classified according to whether they use static or dynamic subtasks priorities. All of the previous policies use dynamic priorities. Presently, the *weight-monotonic* (WM) policy, proposed by Baruah [10], is the only static-priority Pfair policy. Under WM, tasks with higher weights are given higher priority.

In [10], Baruah proved that WM correctly schedules Pfair task systems on a uniprocessor when $\sum_{T \in \tau} T.w \leq \ln 2$. (This bound closely resembles that given for the RM policy.) Ramamurthy and Moir later extended this uniprocessor condition into a sufficient multiprocessor condition [58]. Using the condition from [58], Andersson and Jonsson [8] proved the utilization-based condition shown below.

$$\sum_{T \in \tau} T.w \leq \frac{M}{2} \tag{2.8}$$

As remarked in [8], no static-priority policy can provide a better utilization-based condition than (2.8). Hence, the use of a static-priority Pfair policy sacrifices up to half of the total processing capacity when compared to an optimal Pfair policy.

Figure 2.3(d) shows the schedule produced by the WM policy. By the task weights, the priority order (from highest to lowest) is as follows: T, U, V, W, X. As shown, X (*i.e.*, the lowest-priority task) misses each of its first four deadlines.

2.3 Other Task Models

All work discussed up to this point has focused on the Pfair task model. Unfortunately, this model is far too restrictive in its assumptions. Srinivasan and Anderson [5, 61] addressed this problem by proposing more task models that permit more flexibility in how tasks behave at runtime. In this section, we summarize these task models.

2.3.1 Early-release Task Model

One unappealing aspect of Pfair scheduling is that it is not work-conserving with respect to jobs, *i.e.*, a ready job may not be permitted to utilize an idle processor because doing so would violate the Pfairness constraint. Figure 1.10 (in Chapter 1) illustrates this shortcoming: at time 1, the processor idles because **T3** is considered ineligible.

To avoid such unnecessary idling, Anderson and Srinivasan [5] proposed the *early-release* (ER) task model and proved that PD^2 correctly schedules any ER task set satisfying (2.4). Under this model, each subtask T_i has an *eligibility time* $e(T_i)$ $(\leq r(T_i))$, which is the earliest time at which T_i can be scheduled.[5] A schedule respects ERfairness if and only if each subtask T_i is scheduled in $[e(T_i), d(T_i))$. This interval is called T_i's *IS-window*, while the interval $[r(T_i), d(T_i))$ is called T_i's *PF-window*.[6]

Figure 2.2(b) illustrates how early releasing may be used when each job requires three quanta. Subtasks T_2 and T_3 become eligible when T_1 is released. Similarly, subtasks T_4, T_5,

[5] $e(T_i)$ is required to fall on a slot boundary.

[6] "IS" refers to the intra-sporadic task model, which is described next. Although ER-window would seem more appropriate, these terms were not introduced until the IS task model was presented.

and T_6 all become eligible simultaneously. As with the models described below, early releasing can be done reactively, *i.e.*, eligibility times can be assigned on-the-fly.

2.3.2 Intra-sporadic Task Model

Another problem with the Pfair task model is its lack of support for suspensions. Recall that suspensions can result both from blocking and device I/O. Substantial processor time can be wasted if suspended tasks are required to compete for processor time.

In [61], Srinivasan and Anderson introduced the *intra-sporadic* (IS) task model to address this need and proved that PD^2 successfully schedules any IS task set satisfying (2.4). The IS model extends the ER model by allowing subtasks to be delayed, provided that two restrictions are respected. First, all subtask releases must occur on slot boundaries. Second, the relative separation between each pair of windows must be at least that guaranteed under the Pfair task model. Effectively, the IS model is an extension of the sporadic model [49] to subtasks. A schedule respects ISfairness if and only if each subtask is scheduled in its IS-window.

Suppose that T_2's release in Figure 2.2(b) is delayed for six slots, *i.e.*, it is released at time 9 as shown in Figure 2.2(c) instead of at time 3. Releasing T_3 earlier than time 12 would result in a relative separation between T_2 and T_3 that is less than three slots. Since the relative separation is three slots under the Pfair model (see Figure 2.2(a)), the separation restriction will be violated unless $r(T_3)$ is delayed until at or after time 12. However, $e(T_3)$ is still allowed to be earlier than 12, as shown.

2.3.3 Generalized Intra-sporadic Task Model

In [61], Srinivasan and Anderson also introduced the *generalized intra-sporadic* (GIS) task model. This model extends the IS model by allowing subtasks to be omitted entirely. However, the minimum-separation restriction must still be satisfied for all windows that are present. Unlike the previous models, the GIS task model was developed primarily to facilitate proofs rather than to provide more flexible scheduling.

However, one important ability is provided by the GIS task model. When associating a group of subtasks with a specific job of a task (as Property P suggests), there is a possibility that the job may not require all of its subtasks. Indeed, since execution requirements of jobs are often pessimistic estimates of worst-case requirements, it is *likely* that jobs will complete earlier than expected. In such cases, continuing to release the remaining subtasks will both lead to wasted processor time and delay the execution of other tasks. Allocating processor time to such subtasks is no better than idling those processors. A more desirable solution is to simply mark the unused subtasks as absent. Later, we discuss a mechanism that allows such idle processor time to be safely consumed by other ready tasks.

Figure 2.2(d) shows how the window placement from Figure 2.2(c) appears when task T_3 is marked absent. Notice that T_4 cannot be released before time 16 without violating the minimum separation between T_2 and T_4.

As with the previous models, Srinivasan and Anderson proved that the PD^2 algorithm correctly schedules any GIS task set satisfying (2.4). A schedule respects GISfairness if and only if each present subtask is scheduled in its IS-window. Since this model generalizes the ER and IS models, we consider only the GIS model in the remainder of the dissertation. A GISfair task will be denoted by $\mathbf{GIS}(w)$.

2.4 Other Related Work

In this section, we summarize other prior results relating to Pfair scheduling.

2.4.1 Dynamic Task Sets

In real systems, the composition of the task set may need to change at runtime in response to events. For instance, monitoring software is often based on multiple *modes* of operation. Different modes perform different operations, and hence require different tasks.

To support on-line mode changes, Srinivasan and Anderson [62] derived conditions under which GIS tasks may dynamically join and leave a running system. A task may join as long as (2.4) continues to hold. Leaving, however, is more complicated. A light task T may leave at any time $t \geq d(T_i) + b(T_i)$, where T_i is the last-executed subtask of T. This rule also applies to heavy tasks in the special case in which $\mathsf{maxsum}_{M-1}\{\, T.w \mid T \in \tau \,\} \leq 1$ holds. Otherwise, a heavy task cannot leave until its next group deadline, *i.e.*, $D(T_i)$.

2.4.2 Supertasking

In [48], Moir and Ramamurthy considered the problem of forcing tasks to execute on specified processors. This work was motivated by the observation that some devices may be accessible only from certain processors. Hence, tasks performing device I/O may not be migratable. They further noted that binding tasks to specific processors can reduce migration overhead.

To support bound tasks, they proposed the use of *supertasks*. In this approach, each supertask $\mathcal{S}$ replaces a set of *component tasks*, and is assigned the weight

$$\mathcal{S}.w = \sum_{T \in \mathcal{S}} T.w.$$

(We let $\mathcal{S}$ denote both the supertask and component-task set.) Whenever a supertask is scheduled, one of its component tasks is executed according to an internal scheduling policy.

Although Moir and Ramamurthy proved that EPDF is an optimal internal scheduling

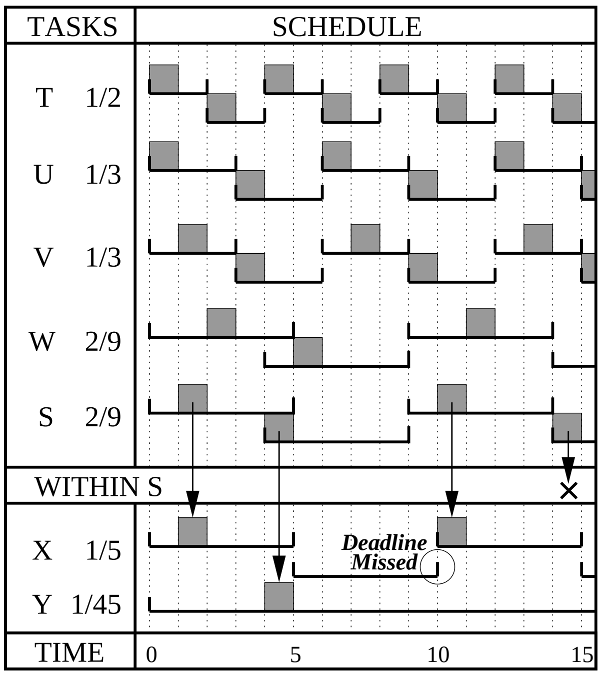


Figure 2.5: Sample supertasking schedule. Supertask S serves two component tasks, X with weight 1/5 and Y with weight 1/45, and is assigned their cumulative weight. The PD^2 policy is used globally, while the EPDF policy is used within S. As shown, task X misses a deadline at time 10.

policy for Pfair component tasks, they also demonstrated that component-task deadline misses may occur when using supertasks with PF, PD, or PD^2. Figure 2.5 illustrates such a miss. In this example, a supertask S replaces two component tasks X and Y, and is assigned a weight as described above. The PD^2 policy is used to schedule tasks globally, while the EPDF policy is used within S. Arrows show where S re-allocates its processor time to its component tasks. As shown, a deadline miss occurs at time 10.

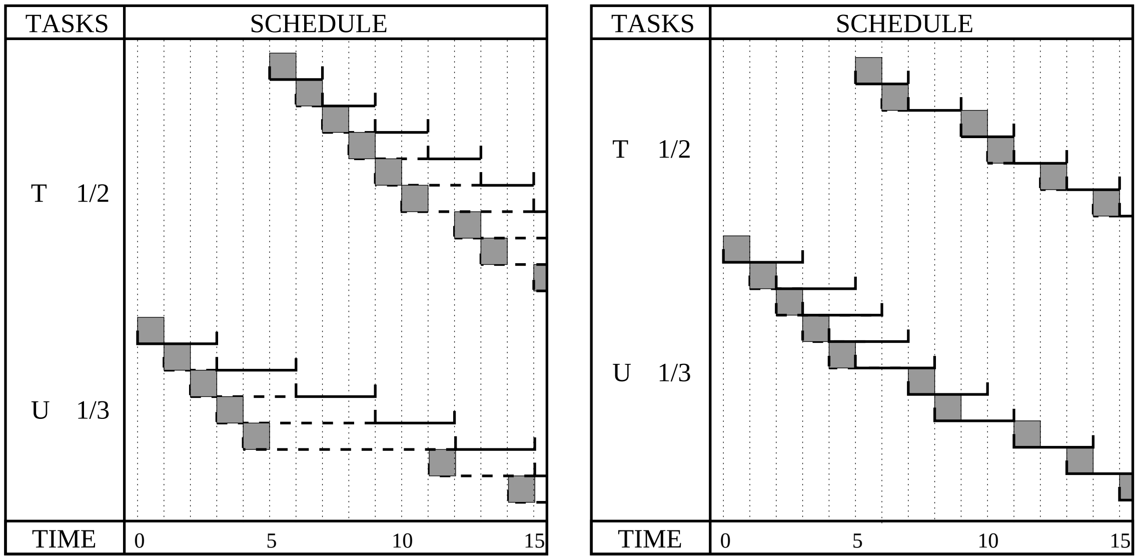
We extend supertasking into a general mechanism for hierarchal scheduling. Specifically, in this context, *supertasking* refers to the act of scheduling a collection of persistent tasks as a single entity. When using supertasks, we assume that the set of supertasks, denoted π, covers τ, *i.e.*, each task in τ must be assigned to exactly one supertask. Also, we assume that each supertask contains at least one component task. Notice that Pfair scheduling without supertasks is analogous to assigning each task to a unique supertask.

2.4.3 Utilizing Idle Time

Idle time can arise from two sources. First, when the processors are underutilized, the unused capacity will result in occasional idling. Second, even when all capacity has been reserved, IS delays and absent subtasks can result in idle processors. In either case, this idle time can be used to speed the execution of unscheduled ready tasks. We briefly summarize techniques for consuming this time below.

The most obvious technique for consuming idle time is the use of early releases. Unfortunately, early releasing has the effect of consuming future deadlines. Hence, tasks may be penalized for consuming idle time using this technique. Figure 2.6(a) illustrates this problem with a simple two-task uniprocessor schedule. Task U uses many subtasks in $[0, 5)$ to consume idle time. As a result, its next deadline is far into the future (at time 18) when T joins at time 5, causing U to be treated unfairly over $[5, 10)$.

To avoid this penalty, Anderson, Block, and Srinivasan proposed an alternative approach called *quick releasing* [2]. In this approach, tasks are allowed to left-shift their subtask windows so that they are released only one or two slots after a detected idle slot. The excess time in the idle slot can then be consumed by a bounded early release. Figure 2.6(b) illustrates this approach. Quick releasing provides an additional advantage in that a maximum weight can be assigned to each task. Maximum weights are enforced by restricting how far subtask windows can be shifted, thereby avoiding the need for additional accounting. The primary disadvantage of this approach is that spare capacity may not always be distributed proportionally among tasks. However, experimental results presented in [2] suggest that proportional



(a) Early Release **(b) Quick Release**

Figure 2.6: Two approaches for consuming idle processor time. **(a)** Using early releases, windows maintain their original position; tasks may be penalized for consuming time. **(b)** Using quick releases, windows are permitted to shift earlier in time when an idle slot is detected.

distribution occurs in all but a few pathological cases.

2.4.4 Best-effort Pfairness

In [18], Chandra, Adler, and Shenoy considered the use of Pfair scheduling in a general-purpose system. They observed that the fairness and isolation guarantees provided by Pfairness are ideal for multimedia applications. However, since only best-effort guarantees were needed, worst-case predictability could be sacrificed to achieve more scheduling flexibility.

For such cases, they proposed the Deadline Fair Scheduling (DFS) policy, which is loosely based on PD^2. Several aspects of this policy are unique among Pfair policies. First, work-conserving behavior is guaranteed by immediately re-assigning a processor when the executing task yields. Second, an auxiliary scheduler is used to distribute idle time to unscheduled ready tasks.[7] Third, priorities are biased in order to account for processor affinity.

This work is interesting for several reasons. First, it suggests that Pfair scheduling has applications outside the real-time domain. Second, the experimental results presented in [18] show only a marginal increase in scheduling overhead when using DFS (compared to the default Linux policy). Since EPDF and PD^2 are less complex than DFS, this study suggests that their scheduling overhead should be reasonable in practice.

[7]This work pre-dated the development of the early-release and quick-release models.

CHAPTER 3

System Overview

In this chapter, we outline the design of a Pfair-based operating system. We begin by discussing practical aspects of quantum-based scheduling in Section 3.1. We then state our assumptions concerning global scheduling and prove some basic properties of the system in Section 3.2. Global scheduling of independent periodic and sporadic tasks is considered next in Section 3.3. Finally, we discuss the use of server tasks in Section 3.4. Later chapters use the results presented in this chapter when addressing more complex issues, including the use of supertasks (Chapter 4) and support for task synchronization (Chapters 5 and 6).

3.1 Quantum-based Scheduling

In this section, we consider practical implications of quantum-based scheduling. As mentioned in Chapter 1, practical overheads impose implicit bounds on the slot length. This section discusses these overheads in more detail. An important implication of the discussion that follows is that quantum sizes and slot lengths may vary at runtime, contradicting the view (from Chapter 2) that these values are fixed. Toward the end this section, we explain how to account for this variance when performing worst-case analysis.

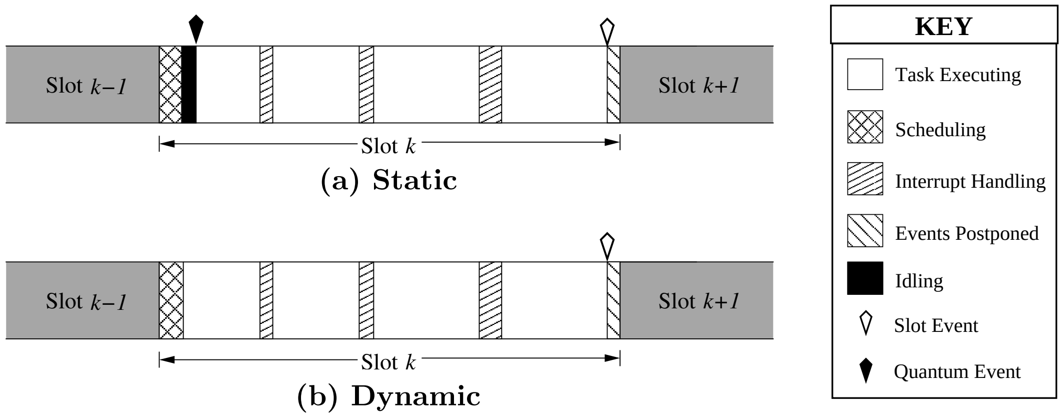


Figure 3.1: Illustration of basic slot structures. **(a)** Under a static structure, events mark both the quantum and slot boundaries, thereby reserving a fixed portion of each slot for scheduling. **(b)** Under a dynamic structure, only slot boundaries are marked; quanta begin immediately after scheduling.

3.1.1 Basic Approaches

Quantum-based scheduling can be achieved using two approaches. Under both approaches, an event (likely a timer interrupt) is needed to signal the end of each slot. We refer to this event as the *slot* event. In the *static* approach, each slot is divided into a scheduling portion and an execution portion by a second event that signals the start of the quantum. We refer to this second event as the *quantum* event. Figure 3.1(a) illustrates the static approach. As shown, this approach will likely result in processor idling since most scheduler invocations will complete before the quantum event occurs. In the *dynamic* approach, each quantum begins immediately after scheduling (*i.e.*, as soon as possible), as shown in Figure 3.1(b).

The benefit of the dynamic approach is that more processor time is allocated to tasks on average. Hence, this approach is more suitable for best-effort scheduling. On the other hand, the static approach usually produces more uniform quanta, which makes it more predictable. Hence, this approach is better suited to real-time scheduling.

3.1.2 Complicating Factors

Several overheads contribute to the per-slot overhead under quantum-based scheduling, most of which are illustrated by Figure 3.1. First, the scheduler consumes some processor time at the start of each slot. We let $\xi_{\max}$ and $\xi_{\min}$ denote upper and lower bounds, respectively, on the time consumed by the scheduler in each slot. Second, the quantum and slot events may not occur exactly at the requested times. For instance, when using hardware interrupts to produce these events, interrupt dispatching and signal delay can introduce a small amount of latency. We let $\jmath_-$ and $\jmath_+$ denote the worst-case magnitude of this event jitter, *i.e.*, an event set to occur at time t will occur within $[t - \jmath_-, t + \jmath_+]$.

Third, to avoid ill-timed preemptions, slot events on the local processor may sometimes need to be temporarily postponed. The reason is that data structures used for scheduling are shared resources. For instance, an operation that modifies task weights must update not only the weights themselves, but also the current priority of each affected task. Hence, accesses to these data structures must be synchronized to ensure correctness of both the operations applied to this data and the scheduling decisions. Typically, a spin lock or similar mechanism is used to prevent simultaneous access from multiple processors. However, because the scheduler invocations are triggered automatically by slot events, this does not prevent such an invocation from preempting a partially completed operation. To prevent such a preemption, an operation can temporarily postpone the delivery of slot events, thereby delaying invocation of the scheduler.[1] We let φ denote the maximum duration of a postponement. Finally, interrupts can result in occasional unavoidable delays. Due to the complexity of this factor,

[1]This is a slight oversimplification since postponing events on the local processor does not prevent scheduler invocations on other processors from accessing the data. Since scheduler invocations should occur simultaneously across processors, a software barrier can be used to delay all invocations whenever one is delayed.

we discuss it in greater detail below.

To bound interrupt overhead, it is necessary to first understand interrupt handling. Most modern systems limit the disruptiveness of interrupts by using *split* interrupt handling. When an interrupt occurs, an immediate handler is dispatched to quickly clear the interrupt, to store any necessary data, and to enqueue a request for further processing. Deferred handling (*i.e.*, the execution of the enqueued request) is then scheduled as any other one-shot task would be. Hence, the disruptions shown in Figure 3.1 are caused solely by immediate handlers.

To account for the execution of handlers, Jeffay and Stone [38] proposed treating handlers like periodic or sporadic tasks. Ramamurthy [57] later considered applying this approach under the WM policy. However, it is unclear whether interrupt periods and execution requirements are sufficiently large for this approach to be effective. For instance, this approach is not applicable when the period of the interrupt is shorter than the slot length. In addition, immediate handlers are designed to operate quickly. Hence, the execution requirements of these handlers are likely to be extremely small relative to the quantum size. Based on these observations, it seems inappropriate to treat immediate handlers as periodic or sporadic tasks under a quantum-based model.

Alternatively, interrupt overhead can be pessimistically bounded by computing the worst-case per-slot overhead. Letting p denote the minimum separation between interrupts of a particular type that must be handled by the same processor, the maximum number of immediate handlers executing within a single slot is given by $\lceil \frac{1}{p} \rceil + 1$. Multiplying this term by the per-interrupt overhead yields a bound on the per-slot overhead induced by that interrupt type. To simplify this discussion, we simply assume that $\imath_{\min}$ and $\imath_{\max}$ are the minimum and maximum amount, respectively, of processor time consumed by all interrupts within each slot. Techniques for determining these bounds and for evenly distributing interrupt-handling overhead across processors are left as future work.

3.1.3 Bounding Slot and Quantum Fluctuations

Using the terms introduced above, we now derive limits on the runtime slot lengths and quantum sizes. At runtime, the slot length on a given processor is the time delay between two consecutive slot events on that processor. In our analysis, we assume this delay is one time unit, *i.e.*, all time parameters are measured in units of the assumed slot length. Below, we derive a lower ($S_{\min}$) and upper ($S_{\max}$) bound on the slot lengths that can be observed in the running system. Similarly, we assume that exactly Q units of processor time are allocated to tasks in each slot. However, in a real system, this quantity may also vary from slot to slot. Again, we derive a lower ($Q_{\min}$) and upper ($Q_{\max}$) bound on the range of this quantity. In the next subsection, we consider how these fluctuations impact the analysis presented later, which is based on the assumption that both the slot and quantum lengths are fixed.

We begin with the slot length. We let t and $t+1$ denote the expected starting and ending times, respectively, of a slot, as illustrated in Figure 3.2. Due to jitter, the slot event that begins the slot can occur at any time within $[t - \jmath_-, t + \jmath_+]$. Similarly, due to event postponement and jitter, the slot event marking the end of the slot occurs within $[t + 1 - \jmath_-, t + 1 + \jmath_+ + \varphi]$.[2] Hence, actual slot lengths are bounded by the formulas given below.

$$\begin{aligned} S_{\min} &= 1 - \jmath_- - \jmath_+ \\ S_{\max} &= 1 + \jmath_- + \jmath_+ + \varphi \end{aligned} \tag{3.1}$$

[2]We assume that the time at which a slot event should occur is independent of the jitter experienced by the previous events, *i.e.*, jitter does not produce propagating error.

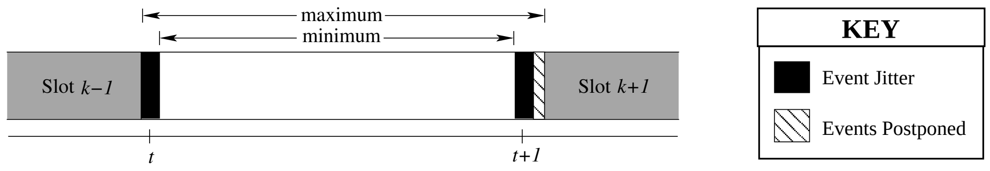


Figure 3.2: Illustration of the minimum and maximum slot length that can occur at runtime.

Under the dynamic approach, the scheduling and interrupt-handling overheads consume at least $\xi_{\min} + \imath_{\min}$ and at most $\xi_{\max} + \imath_{\max}$ additional units of processor time. Hence, the quantum range under this approach is easily derived from (3.1), as shown below.

$$\begin{aligned} Q_{\min} &= S_{\min} - \xi_{\max} - \imath_{\max} = 1 - \jmath_- - \jmath_+ - \xi_{\max} - \imath_{\max} \\ Q_{\max} &= S_{\max} - \xi_{\min} - \imath_{\min} = 1 + \jmath_- + \jmath_+ + \varphi - \xi_{\min} - \imath_{\min} \end{aligned} \tag{3.2}$$

Under the static approach, we must consider the impact of the quantum event. We let $t + q$ denote the requested time of the quantum event, as shown in Figure 3.3. To ensure that the scheduler completes before this event, q must be selected so that $t + q - \jmath_- \geq t + \jmath_+ + \xi_{\max}$ holds. It follows that the minimum safe value of q is given by (3.3).

$$q = \jmath_- + \jmath_+ + \xi_{\max} \tag{3.3}$$

Using the above definition for q, the quantum begins within $[t + \jmath_+ + \xi_{\max}, t + \jmath_- + 2\jmath_+ + \xi_{\max}]$. Since the slot ends within $[t + 1 - \jmath_-, t + 1 + \jmath_+ + \varphi]$ (as explained above), the quantum range under the static approach is bounded by the formulas given below.

$$\begin{aligned} Q_{\min} &= (t + 1 - \jmath_-) - (t + \jmath_- + 2\jmath_+ + \xi_{\max}) - \imath_{\max} = 1 - 2\jmath_- - 2\jmath_+ - \xi_{\max} - \imath_{\max} \\ Q_{\max} &= (t + 1 + \jmath_+ + \varphi) - (t + \jmath_+ + \xi_{\max}) - \imath_{\min} = 1 + \varphi - \xi_{\max} - \imath_{\min} \end{aligned} \tag{3.4}$$

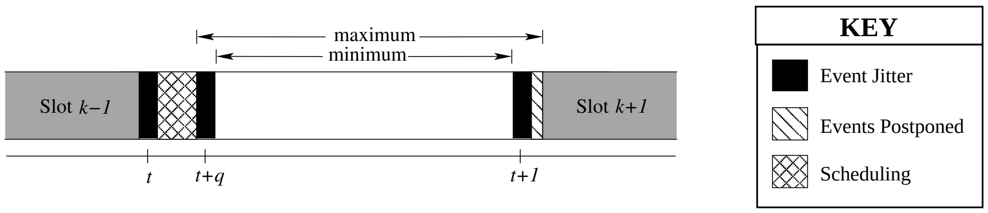


Figure 3.3: Illustration of the minimum and maximum quantum sizes that can occur at runtime under a static slot layout.

3.1.4 Accounting for Fluctuations

The key to accounting for slot and quantum fluctuations is to identify whether using the upper or lower bounds (derived above) produces more pessimism in a given situation. (We are aware of no analysis under which the worst case is not determined by one of these two extremes.) For instance, the maximum number of quanta required to obtain at least C units of processor time is given by $\left\lceil \frac{C}{Q} \right\rceil$ under the assumption that Q is constant. In this case, replacing Q with Q_{min} yields more pessimism. However, when estimating the number of interfering events occurring within a single slot (as is commonly done when analyzing resource contention), taking S_{max} to be the slot length produces more interference and hence more pessimism.

3.1.5 Quantifying Design Efficiency

We now explain how the efficiency of an implementation can be estimated based on the bounds given above. Our goal is to derive a pessimistic lower bound on the total fraction of processor time used to service tasks. This goal is achieved by determining the fraction of a slot that is guaranteed to the task scheduled in that slot. A lower bound on this quantity then provides an estimate of the total fraction of processor time made available to tasks.

First, consider a dynamic implementation. Let s_1 and s_2 denote the beginning and end, respectively, of the slot under consideration, *i.e.*, the times at which the corresponding slot events occur. Since the amount of processor time guaranteed to the scheduled task is lower bounded by $s_2 - s_1 - \xi_{\max} - \imath_{\max}$, the fraction of the slot granted to the scheduled task is given by

$$\frac{s_2 - s_1 - \xi_{\max} - \imath_{\max}}{s_2 - s_1} = 1 - \frac{\xi_{\max} + \imath_{\max}}{s_2 - s_1} \geq 1 - \frac{\xi_{\max} + \imath_{\max}}{S_{\min}}.$$

It follows from the inequality given above and (3.1) that the dynamic approach allocates at least a fraction

$$1 - \frac{\xi_{\max} + \imath_{\max}}{1 - \jmath_- - \jmath_+} \tag{3.5}$$

of the processor time to tasks.

Now, consider a static implementation. Let q_1 denote the time at which the quantum event occurs. In this case, the processor time granted to the scheduled task is at least $s_2 - q_1 - \imath_{\max}$, resulting in the following lower bound.

$$\frac{s_2 - q_1 - \imath_{\max}}{s_2 - s_1} = 1 - \frac{q_1 - s_1 + \imath_{\max}}{s_2 - s_1} \geq 1 - \frac{q_1 - s_1 + \imath_{\max}}{S_{\min}}$$

As explained above, $s_1 \geq t - \jmath_-$ and $q_1 \leq t + \jmath_- + 2\jmath_+ + \xi_{\max}$, where t is the expected value of s_1. Hence, $q_1 - s_1 \leq (t + \jmath_- + 2\jmath_+ + \xi_{\max}) - (t - \jmath_-) = 2\jmath_- + 2\jmath_+ + \xi_{\max}$. It follows from these two inequalities and (3.1) that the static approach allocates at least a fraction

$$1 - \frac{\xi_{\max} + \imath_{\max} + 2\jmath_- + 2\jmath_+}{1 - \jmath_- - \jmath_+} \tag{3.6}$$

of the processor time to tasks.

Although these formulas suggest that the dynamic approach provides a slightly higher fraction of processor time to tasks, this benefit comes at the expense of more variation in the quantum sizes. As suggested in the previous subsection, increased variability translates into increased pessimism when performing worst-case analysis.

3.2 Scheduling Overview

This section provides an overview of scheduling requirements and the guarantees that will be provided to tasks by the scheduler. In Section 3.2.1, we summarize the different sets of tasks that we consider supporting, and the approaches considered for each set. In Section 3.2.2, we summarize parameters that we use to describe the timing constraints of tasks that were not given in the previous chapters. Section 3.2.3 then discusses the scheduling guarantees that we assume are provided by the global scheduler. In addition, to support soft real-time tasks, we formally introduce a tardiness parameter that we assume is given for each task. In Section 3.2.4, we derive basic scheduling properties based on the assumptions made in Section 3.2.3. These properties provide a basis for the results presented later.

3.2.1 Supported Tasks

We consider support for the following three sets of tasks:

- recurrent real-time tasks (τ)
- one-shot real-time tasks
- non-real-time tasks

In the analysis that follows, τ is assumed to consist of only periodic and sporadic tasks, though tasks with rate-based requirements are easily incorporated. Two approaches will be considered for these tasks. First, we consider direct scheduling by the global scheduler. Since the global scheduler expects Pfair tasks, this approach requires that we map the periodic and sporadic requirements onto GIS tasks. Property P, presented earlier in Chapter 2, is an example of a mapping from a synchronous periodic requirement onto a Pfair task. We extend this prior work by explaining how to map requirements when faced with suspensions and with job releases and deadlines that do not fall on slot boundaries. These results are presented later in this chapter. Second, we consider group scheduling of these tasks using the supertasking approach [48]. Chapter 4 is devoted to this approach.

One-shot and non-real-time tasks are supported through the use of server tasks. We use the term *server* to refer to any task that uses its allocated processor time to service requests made by other tasks, called *clients*. In this context, the clients are the one-shot and non-real-time tasks, and the requests are their jobs. To determine the order in which requests are serviced, a server applies an internal policy. Later in this chapter, we discuss the use of servers in more detail. Since any scheduling policy can be applied to non-real-time tasks,[3] we do not consider this set beyond this section. Instead, we make the simplifying assumption that the execution of non-real-time tasks does not impact the execution of tasks in the other sets. Support for one-shot tasks with deadlines is discussed later in the chapter.

3.2.2 Task Requirements

To support soft real-time scheduling, we associated a tardiness threshold, denoted $T.c$ (≥ 0), with each task T in addition to the previously described parameters. $T.c$ is the maximum amount by which T is allowed to miss any deadline. The purpose in making these parameters explicit is to maintain a distinction between the *preferred* requirements of tasks and those that can actually be achieved, *e.g.*, the tardiness threshold of a soft real-time task may need

[3]Though good performance is often desired for these tasks, no formal constraints are imposed.

to be increased to make the task set schedulable.

For Pfair tasks, tardiness thresholds are associated with subtask deadlines. Since allowing a subtask deadline to lie within a slot will result in the previous slot boundary being treated as the actual deadline, we assume that such Pfair-based tardiness thresholds always correspond to a slot boundary, *i.e.*, we assume that $T.c$ is an integer for all Pfair tasks T.

When performing deadline-based scheduling of a task T that has a non-zero miss threshold, there are two basic approaches: prioritize according to preferred deadlines (*i.e.*, $T.d$) or prioritize according to *effective* deadlines (*i.e.*, $T.d+T.c$). Because the latter approach results in more effective scheduling, we focus on it when performing supertask analysis later in Chapter 4. Intuitively, this approach suggests that guaranteeing miss thresholds of *all* tasks is more important than guaranteeing the preferred deadlines of *some* tasks. Our results can easily be adapted to the alternative approach.

3.2.3 Scheduling Guarantees

In most prior work on Pfair scheduling [5, 6, 7, 10, 11, 12, 48], Pfairness was considered as a means to an end. Specifically, the task sets under consideration often do not require this strong notion of fairness in order to satisfy their constraints. In such cases, imposing a strong fairness constraint can *increase* overhead since more complex scheduling policies are needed to satisfy the constraint. For instance, the EPDF policy incurs less overhead than the optimal Pfair policies [63], making it more suitable when scheduling task sets that can tolerate EPDF's shortcomings.

In this dissertation, the scheduling policy is treated as a black box, *i.e.*, we consider only the guarantees provided by the scheduler and base our work on the properties that follow from these guarantees. To describe the scheduler's guarantees, we introduce a simple four-parameter model (described below). The purpose of this model is to allow each presented result to be applied both under the current Pfair scheduling policies and under future policies based on the same derivation. In addition, the use of this model when deriving properties enables a quantitative comparison of using relaxed fairness (*e.g.*, that provided by the EPDF policy) instead of Pfairness.

Model. Our model consists of two pairs of parameters. First, we let β_- (≥ 1) and β_+ (≥ 1) denote (real-valued) lower and upper *lag scalers*. These scalers are multiplied by $-Q$ and Q, respectively, to yield the actual lag bounds guaranteed by the scheduler, as shown below.

$$\text{for all } t,\ -Q \cdot \beta_- < \mathit{lag}(T, t) < Q \cdot \beta_+ \tag{3.7}$$

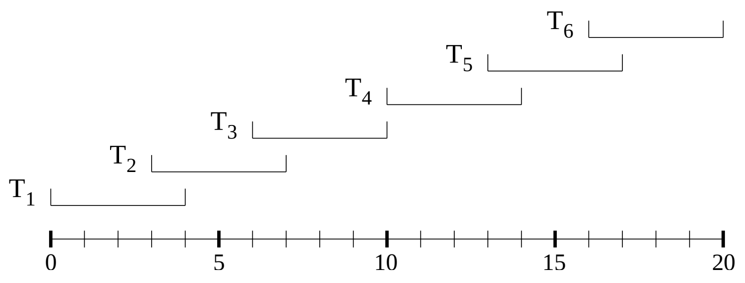
To simplify the presentation, we let

$$\beta = \beta_+ + \beta_-. \tag{3.8}$$

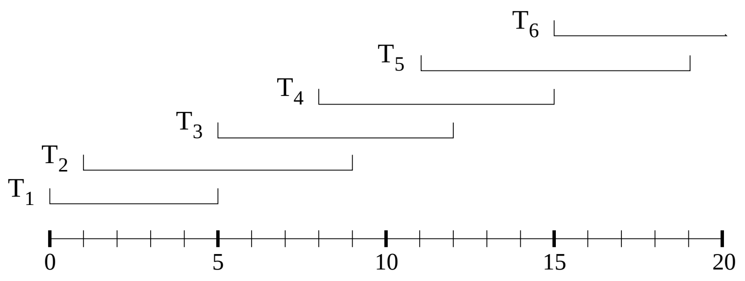
The constraint given by (3.7) generalizes (2.3), which corresponds to the $\beta_- = \beta_+ = 1$ case.

Relaxing lag bounds scales each subtask window. However, due to the use of quantum-based scheduling, windows are clipped to slot boundaries, resulting in non-uniform scaling. We refer to the windows defined by (3.7) as *relaxed* windows. Figure 3.4(b) shows the first six relaxed windows for a task with weight $\frac{3}{10}$ when $\beta_- = \beta_+ = 1.5$; Figure 3.4(a) shows the corresponding Pfair window layout. Notice that $\omega(T_2)$'s release occurs two slots earlier in Figure 3.4(b), while $\omega(T_3)$'s release occurs only one slot earlier.

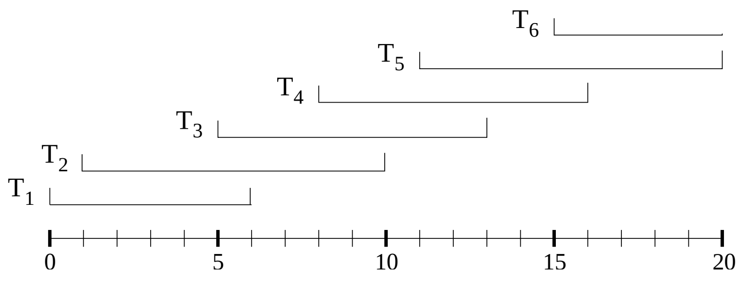
The second parameter pair is ϵ_r and ϵ_d, which denote the number of slots by which each pseudo-release and pseudo-deadline, respectively, is extended (beyond its lag-based place-



(a) Pfair Windows



(b) Relaxed Windows when $\beta_+ = \beta_- = 1.5$



(c) Extended Windows when $\beta_+ = \beta_- = 1.5$**,** $\epsilon_r = 0$**, and** $\epsilon_d = 1$

Figure 3.4: The first six windows of a task with weight $\frac{3}{10}$ are shown up to time 20. **(a)** Windows defined by Pfairness constraint. **(b)** Relaxed windows defined by $\beta_+ = \beta_- = 1.5$. **(c)** Extended windows defined by $\beta_+ = \beta_- = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$.

ment). More precisely, the scheduler treats a subtask with a relaxed window spanning $[t_r, t_d)$ as having the window $[t_r - \epsilon_r, t_d + \epsilon_d)$. Figure 3.4(c) shows the window layout obtained by $\beta_- = \beta_+ = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Notice that each deadline is extended by one slot, relative to Figure 3.4(c), due to ϵ_d. Such windows are called *extended* windows. For example, T_2 in

Figure 3.4(c) has an extended deadline at time 10. We let

$$\epsilon = \epsilon_r + \epsilon_d. \tag{3.9}$$

3.2.4 Basic Properties

In this section, we derive some basic properties of global scheduling based on the parameters described above. We use these properties as the basis for the work presented later.

Window placement. The first lemma and corollary, shown below, derive formulas for relaxed and extended windows. These formulas represent only the *guarantee* provided by the scheduler; we make no assumptions about how this guarantee is provided by the scheduler, beyond those already stated above.

Lemma 3.1 *The following formulas define the placement of relaxed windows:*

$$r(T_i) = \left\lfloor \frac{i - \beta_+}{T.w} \right\rfloor \qquad\qquad d(T_i) = \left\lceil \frac{(i-1) + \beta_-}{T.w} \right\rceil.$$

Proof. The derivations presented here extend those used to derive (2.5) and (2.6). These formulas follow directly from (3.7), which implies that the release and deadline of a subtask T_i satisfy the following constraints:

$$r(T_i) = \min\{\ k \mid k \in \mathcal{Z} \land i \cdot Q - \mathit{fluid}(T, 0, k+1) < Q \cdot \beta_+\ \};\ \text{and}$$

$$d(T_i) = \min\{\ k \mid k \in \mathcal{Z} \land (i-1) \cdot Q - \mathit{fluid}(T, 0, k) \le -Q \cdot \beta_-\ \}.$$

In the constraints given above, $\mathcal{Z}$ denotes the set of all integers. Informally, the $r(T_i)$ constraint identifies the earliest slot (k) such that the upper lag constraint ($Q \cdot \beta_+$) is not violated when the i^{th} quanta is received in that slot (*i.e.*, in the interval $[k, k+1)$). The subtask release corresponds to the start of this slot, *i.e.*, time k.[4] On the other hand, the $d(T_i)$ constraint identifies the earliest time k such that the lower lag bound $(-Q \cdot \beta_-)$ *is* violated when only $i-1$ quanta are received by k. It follows that the i^{th} quantum must be received in the interval $[k-1, k)$, at the latest. Hence, the subtask deadline occurs at time k.

Applying (2.1) and rearranging terms to isolate k produces the following equivalent forms.

$$\begin{aligned} r(T_i) &= \min\left\{ k \;\middle|\; k \in \mathcal{Z} \wedge k > \frac{i - \beta_+}{T.w} - 1 \right\} \\ d(T_i) &= \min\left\{ k \;\middle|\; k \in \mathcal{Z} \wedge k \geq \frac{(i-1) + \beta_-}{T.w} \right\} \end{aligned}$$

The lemma follows. □

Corollary 3.1 *The following formulas define the placement of extended windows:*

$$r(T_i) = \left\lfloor \frac{i - \beta_+}{T.w} \right\rfloor - \epsilon_r \qquad d(T_i) = \left\lceil \frac{(i-1) + \beta_-}{T.w} \right\rceil + \epsilon_d.$$

Observe that the above formulas may result in negative release times. Windows that extend beyond zero are effectively truncated at zero at runtime.

Window overlap. The next lemma given below bounds the number of slots by which consecutive subtask windows can overlap in the absence of IS delays.

[4] Notice that the release time may be negative. It is important to understand that time 0 is simply a reference point that records when scheduling begins. Since no scheduling occurs prior to this point, windows with negative release times are effectively truncated to begin at time 0.

Lemma 3.2 *Each pair of consecutive subtasks T_i and T_{i+1} satisfy the inequality shown below.*

$$\left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon \le d(T_i) - r(T_{i+1}) \le \left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1$$

If T_i is the last subtask in a cycle, then

$$d(T_i) - r(T_{i+1}) = \mathcal{B}(T) \overset{\text{def}}{=} \left\lceil \frac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \frac{\beta_+ - 1}{T.w} \right\rceil + \epsilon.$$

Proof. The following derivation establishes the upper bound.

$$\begin{array}{ll} d(T_i) - r(T_{i+1}) & \\ \quad = \left(\left\lceil \frac{(i-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \frac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) & \text{, by Corollary 3.1} \\ \quad \le \left\lceil \frac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + \left\lfloor \frac{i+1-\beta_+}{T.w} \right\rfloor + 1 - \left\lfloor \frac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d & \text{, } \lceil a+b \rceil \le \lceil a \rceil + \lfloor b \rfloor + 1 \\ \quad = \left\lceil \frac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + 1 + \epsilon_r + \epsilon_d & \text{, simplification} \\ \quad = \left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1 & \text{, by (3.8) and (3.9)} \end{array}$$

The following derivation establishes the lower bound.

$$\begin{array}{ll} d(T_i) - r(T_{i+1}) & \\ \quad = \left(\left\lceil \frac{(i-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \frac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) & \text{, by Corollary 3.1} \\ \quad \ge \left\lceil \frac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + \left\lfloor \frac{i+1-\beta_+}{T.w} \right\rfloor - \left\lfloor \frac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d & \text{, } \lceil a+b \rceil \ge \lceil a \rceil + \lfloor b \rfloor \\ \quad = \left\lceil \frac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + \epsilon_r + \epsilon_d & \text{, simplification} \\ \quad = \left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon & \text{, by (3.8) and (3.9)} \end{array}$$

When T_i is the last subtask in a cycle, then $\frac{i}{T.w}$ is an integer since $\mathcal{E}(T) \mid i$ holds. This leads to the derivation shown below.

$$\begin{aligned}
&d(T_i) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Corollary 3.1} \\
&= \tfrac{i}{T.w} + \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil - \tfrac{i}{T.w} - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, since } \tfrac{i}{T.w} \text{ is an integer} \\
&= \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ - 1}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, } \lfloor -a \rfloor = -\lceil a \rceil \\
&= \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ - 1}{T.w} \right\rceil + \epsilon && \text{, by (3.9)}
\end{aligned}$$

The above derivations establish the lemma. □

The previous lemma introduces the new parameter $\mathcal{B}(T)$, which is the number of slots by which consecutive cycles of T overlap. Recall that $\mathcal{B}(T) = 0$ under Pfairness.

Window span. The next result bounds the number of slots crossed by a sequence of n consecutive windows, which we refer to as an *n-span* of a task. For instance, the interval $[3, 13)$ in Figure 3.4(a) is a 3-span since $r(T_2) = 3$ and $d(T_4) = 14$. In general, each n-span corresponds to an interval $[r(T_{i+1}), d(T_{i+n}))$ for some integer i.

Lemma 3.3 *Every sequence of consecutive subtasks $T_{i+1}, \ldots, T_{i+n}$ satisfies the following:*

$$\left\lceil \frac{n+\beta-2}{T.w} \right\rceil + \epsilon \le d(T_{i+n}) - r(T_{i+1}) \le \left\lceil \frac{n+\beta-2}{T.w} \right\rceil + \epsilon + 1$$

When subtask $i+1$ begins a cycle (i.e., $\mathcal{E}(T) \mid i$ holds) and subtask $i+n$ ends a cycle (i.e., $\mathcal{E}(T) \mid (i+n)$ holds),

$$d(T_{i+n}) - r(T_{i+1}) = \frac{n}{T.w} + \mathcal{B}(T)\,.$$

Proof. The following derivation establishes the upper bound in the first claim.

$$\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Corollary 3.1} \\
&\le \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + 1 - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, } \lceil a+b \rceil \le \lceil a \rceil + \lfloor b \rfloor + 1 \\
&= \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \epsilon_r + \epsilon_d + 1 && \text{, simplification} \\
&= \left\lceil \tfrac{n+\beta-2}{T.w} \right\rceil + \epsilon + 1 && \text{, by (3.8) and (3.9)}
\end{aligned}$$

The following derivation establishes the lower bound in the first claim.

$$\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Corollary 3.1} \\
&\ge \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, } \lceil a+b \rceil \ge \lceil a \rceil + \lfloor b \rfloor \\
&= \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{n+\beta-2}{T.w} \right\rceil + \epsilon && \text{, by (3.8) and (3.9)}
\end{aligned}$$

When $\mathcal{E}(T) \mid i$ and $\mathcal{E}(T) \mid (i+n)$ both hold, it follows that both $\frac{i}{T.w}$ and $\frac{i+n}{T.w}$ are integers. Hence, the derivation shown below establishes the second claim.

$$\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Corollary 3.1} \\
&= \tfrac{i+n}{T.w} + \left\lceil \tfrac{\beta_- -1}{T.w} \right\rceil - \tfrac{i}{T.w} - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, since } \tfrac{i}{T.w} \text{ and } \tfrac{i+n}{T.w} \text{ are integers} \\
&= \tfrac{n}{T.w} + \left\lceil \tfrac{\beta_- -1}{T.w} \right\rceil - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \tfrac{n}{T.w} + \left\lceil \tfrac{\beta_- -1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ -1}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, } \lfloor -a \rfloor = -\lceil a \rceil \\
&= \tfrac{n}{T.w} + \left\lceil \tfrac{\beta_- -1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ -1}{T.w} \right\rceil + \epsilon && \text{, by (3.9)} \\
&= \tfrac{n}{T.w} + \mathcal{B}(T) && \text{, by Lemma 3.2}
\end{aligned}$$

This completes the proof. □

3.3 Mapping Independent Tasks

In this section, we explain how to map an independent periodic or sporadic task T onto a GIS task U. Recall that such a mapping enables the scheduling of T by the global scheduler. The results presented here extend Property P from Chapter 2 by accounting for suspensions and for job releases and deadlines that do not fall on slot boundaries. In this section, we consider only independent tasks. In chapters 5 and 6, support for task synchronization is added.

Constraints. The mapping presented is based on the following constraints:[5]

1. Each job of T is associated with a unique group of k consecutive cycles of U; each cycle is associated with only one job of T.

2. The extended release of the first subtask associated with a job must occur on a slot boundary at or after the job's release.

3. The extended deadline of the last subtask associated with a job must occur on a slot boundary at or before time $t_d + T.c$, where t_d denotes the job's deadline.

4. All subtasks must satisfy the minimum separation imposed by the GIS task model.

If the above constraints are satisfied, then all jobs will meet their deadlines provided that a sufficient amount of processor time is allocated to each job while it is not suspended.

Figure 3.5 demonstrates that these constraints are satisfied when applying Property P. By this property, $T = \mathbf{P}(0, 3Q, 10, 10)$ maps to $U = \mathbf{PF}\left(\frac{3}{10}\right)$. As shown, one cycle is associated with each job of T, satisfying Constraint 1. In addition, each job release and deadline coincides with the first subtask release of the associated cycle and last subtask deadline of the associated

[5]These constraints are not necessary, *i.e.*, mappings that do not satisfy these constraints can still be correct.

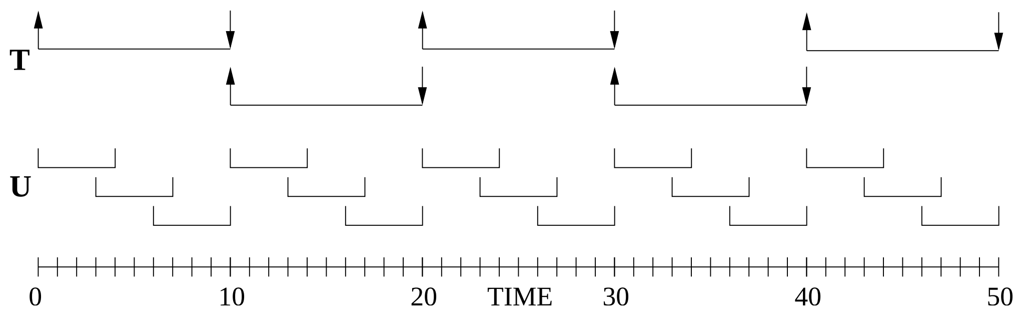


Figure 3.5: Mapping produced by applying Property P to the task $T = \mathbf{P}(0, 3Q, 10, 10)$.

cycle, respectively. For instance, $T(1).a = r(U_1)$ and $T(1).d = d(U_3)$. Hence, Constraints 2 and 3 are satisfied. Finally, the spacing of subtask windows satisfies the minimum separation required by the Pfair task models, as shown. Hence, Constraint 4 is satisfied.

Mapping with no suspensions. First, consider the problem of scheduling a periodic task when all jobs are released on slot boundaries. This case is very similar to that handled by Property P, as the next lemma shows.

Lemma 3.4 *If T is an independent periodic task that never suspends and both $T.\phi$ and $T.p$ are integers, then each job of T will complete within $T.c$ of its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\left\lceil \frac{T.e}{Q} \right\rceil}{\min\left(\lfloor T.d + T.c \rfloor - \mathcal{B}(U), T.p\right)},$$

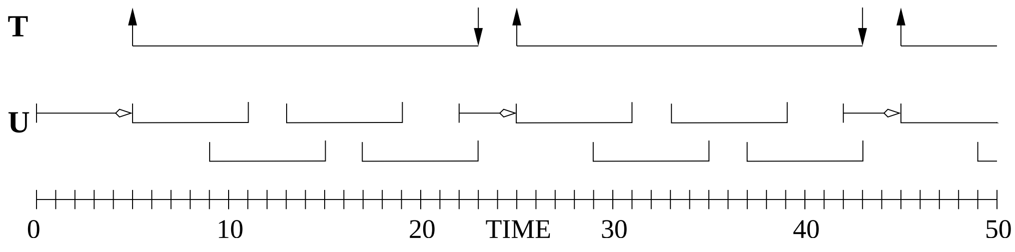
provided that $U.w \in (0, 1]$.

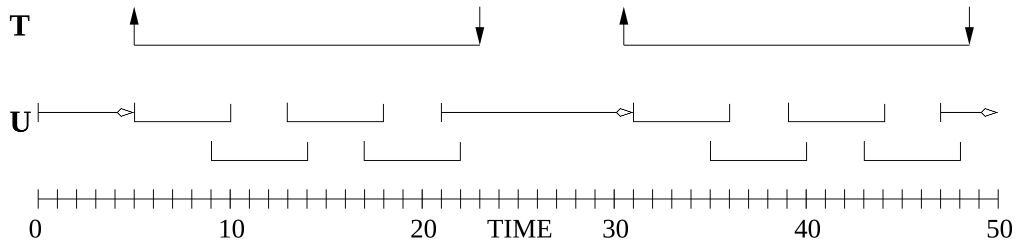
Proof. First, since no suspensions occur, assigning $\left\lceil \frac{T.e}{Q} \right\rceil$ subtasks to each job is sufficient to ensure that the per-job execution requirement is satisfied. In the remainder of the proof, we construct a window layout that ensures that each job receives its associated time by its deadline, while respecting the four constraints given on page 72. We then use the properties proved earlier to determine a weight capable of providing the desired layout. Recall that if a job completes while unused subtasks remain, then these subtasks are marked absent.

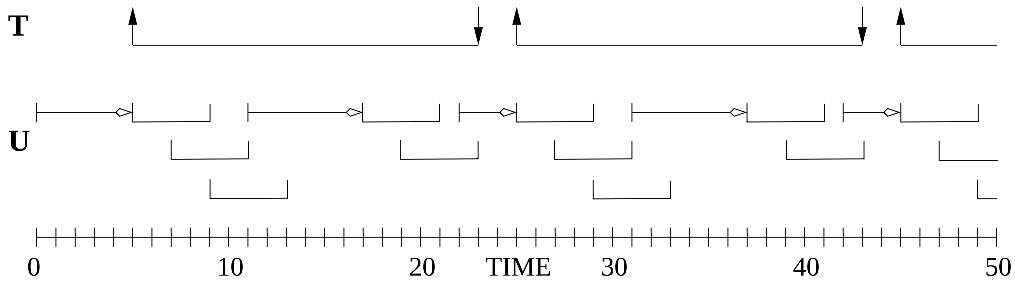
Because $T.\phi$ and $T.p$ are integers, it follows that $T.\phi + k \cdot T.p$ is an integer also for all integers $k \geq 0$. Hence, every job release coincides with a slot boundary. Consider a specific job J that arrives at time a. We let the extended release of the first subtask associated with the job occur at this time, which satisfies Constraint 2. Recall that the next job will be handled by a separate cycle. By Lemma 3.2, consecutive cycles of U can overlap by up to $\mathcal{B}(U)$. Hence, since the next job's first cycle begins at $a + T.p$, the extended deadline of the last subtask associated with J must occur at or before time $a + T.p + \mathcal{B}(U)$ to satisfy Constraint 4. In addition, J's effective deadline occurs at time $a+T.d+T.c$. It follows that this same extended deadline must occur before this time in order to satisfy Constraint 3. Therefore, the extended deadline must occur at or before $a + \min\left(\lfloor T.d + T.c \rfloor, T.p + \mathcal{B}(U)\right)$. Hence, the total window span of the cycle(s) associated with J must be at most $\min\left(\lfloor T.d + T.c \rfloor, T.p + \mathcal{B}(U)\right)$.

We can now use previous results to select a weight such that exactly the span given above is ensured. (A shorter span could be used instead, but would result in an unnecessarily high weight.) Recall that $\left\lceil \frac{T.e}{Q} \right\rceil$ subtask must be associated with each job. By Lemma 3.3, a span of s can be ensured across one or more cycles consisting of n subtasks (combined) by letting $U.w = \frac{n}{s - \mathcal{B}(U)}$. Substituting $s = \min\left(\lfloor T.d + T.c \rfloor, T.p + \mathcal{B}(U)\right)$ and $n = \left\lceil \frac{T.e}{Q} \right\rceil$ into this formula establishes the lemma. $\square$

Figure 3.6(a) shows the application of Lemma 3.4 to the task $T = \mathbf{P}(5, 3.2Q, 20, 18)$ when $T.c = 0$ and the global scheduler is described by $\beta_{-} = \beta_{+} = 1$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Hence, $\mathcal{B}(U) = 1$. The other insets, which consider this same global scheduler with different tasks, are considered later. By Lemma 3.4, $U.w = \frac{\lceil 3.2 \rceil}{\min(\lfloor 18 \rfloor - 1,\ 20)} = \frac{4}{17}$. Notice how IS delays are used


(a) Periodic task with all releases on slot boundaries


(b) Periodic or sporadic task


(c) Multi-phase periodic task with all releases on slot boundaries

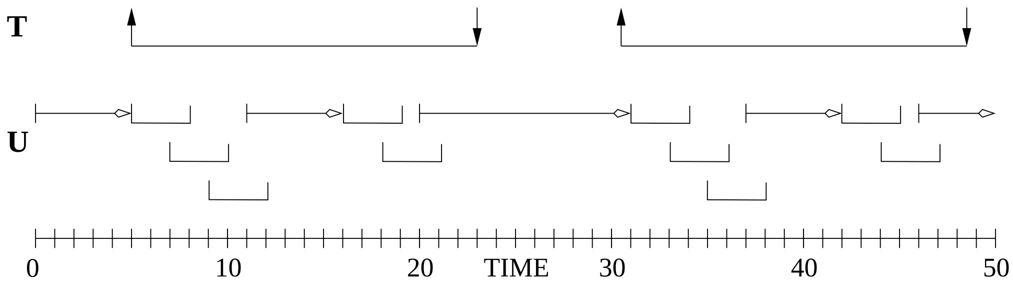

(d) Multi-phase periodic or sporadic task

Figure 3.6: Examples of mapping a periodic or sporadic task T with $T.c = 0$ onto a GIS task U when the global scheduler is described by $\beta_{-} = \beta_{+} = 1$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Examples include **(a)** $T = \mathbf{P}(5, 3.2Q, 20, 18)$, **(b)** $T = \mathbf{S}(5, 3.2Q, 20, 18)$, **(c)** $T = \mathbf{P}(5, 3.2Q, 20, 18) : \mathbf{C}(\emptyset, 2.1Q)\,;\mathbf{I}(3.2)\,;\mathbf{C}(\emptyset, 1.1Q)$, and **(c)** $T = \mathbf{S}(5, 3.2Q, 20, 18) : \mathbf{C}(\emptyset, 2.1Q)\,;\mathbf{I}(3.2)\,;\mathbf{C}(\emptyset, 1.1Q)$.

to ensure that subtask windows properly align properly with the associated job's interval.

The above lemma highlights a problem that is common under rate-based scheduling: weight-assignment formulas are often functions of the task's weight. In this case, the $\mathcal{B}(U)$ term causes the problem. Applying the formula given in Lemma 3.2, the weight assignment given by Lemma 3.4 becomes

$$U.w = \frac{\left\lceil \frac{T.e}{Q} \right\rceil}{\min\left(\lfloor T.d + T.c \rfloor - \left(\left\lceil \frac{\beta_{-}-1}{U.w} \right\rceil + \left\lceil \frac{\beta_{+}-1}{U.w} \right\rceil + \epsilon\right), T.p\right)}.$$

Notice that $U.w$ now occurs on both sides of the formula. Due to the ceiling operators in the right-hand side, these terms cannot be combined. In this case, this dependence can be avoided by letting $\beta_{+} = \beta_{-} = 1$, as is done in our example above. However, in general, some parameters may require the use of iterative computation[6] or similar techniques. Since these techniques are a well-studied area of mathematics, we do not discuss them here.

The next lemma characterizes the cost of allowing jobs to be released off slot boundaries.

Lemma 3.5 *If T is an independent periodic or sporadic task that never suspends, then each job of T will complete within $T.c$ slots of its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\left\lceil \frac{T.e}{Q} \right\rceil}{\min\left(\lfloor T.d + T.c \rfloor - \mathcal{B}(U), \lfloor T.p \rfloor\right) - 1},$$

provided that $U.w \in (0, 1]$.

Proof. This proof closely resembles that of the previous lemma. Suppose a job release occurs

[6]Iterative computations search for a non-trivial value x at which $x = f(x)$ for a given function $f(x)$. The process begins with a value x_0 and iterates using $x_{k+1} := f(x_k)$ until $x_{k+1} = x_k$ holds.

t_Δ (< 1) time units before a slot boundary. In this case, the extended release of the first subtask associated with the job will not occur until the next slot boundary. Following identical reasoning to that given in Lemma 3.4, the span of the cycle(s) associated with each job can be at most $\min\left(\lfloor T.d + T.c - t_\Delta \rfloor, \lceil T.p - t_\Delta \rceil + \mathcal{B}(U)\right) \geq \min\left(\lfloor T.d + T.c \rfloor, \lfloor T.p \rfloor + \mathcal{B}(U)\right) - 1$. The remainder of the proof follows as in the previous lemma. □

Figure 3.6(b) shows the application of Lemma 3.5 to $T = \mathbf{S}(5, 3.2Q, 20, 18)$. In this case, $U.w = \frac{\lceil 3.2 \rceil}{\min(\lfloor 18 \rfloor - 1, 20) - 1} = \frac{4}{16} = \frac{1}{4}$, which implies that 4 cycles are associated with each job.

Mapping with suspensions. The next two lemmas generalize Lemmas 3.4 and 3.5 by adding support for suspension phases. The key idea here is to insert a pessimistic IS delay that prevents the task from being granted processor time while it *may* be suspended. The proof of the following lemma explains this approach in more detail.

Lemma 3.6 *If T is an independent multi-phase periodic task and both $T.\phi$ and $T.p$ are integers, then each job of T will complete within $T.c$ of its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\sum_{T^{[i]} \in \mathcal{C}(\emptyset)} \left\lceil \frac{T^{[i]}.e}{Q} \right\rceil}{\min\left(\lfloor T.d + T.c \rfloor - \mathcal{B}(U), T.p\right) - \sum_{T^{[i]} \in \mathcal{I}} \left(\lceil T^{[i]}.\theta \rceil + \left\lceil \frac{\beta - 2}{U.w} \right\rceil + \epsilon + 1 \right)},$$

provided that $U.w \in (0, 1]$.

Proof. The lemma will be proved by showing that a per-job static window layout is capable of ensuring that each job complete on time. The idea here is very similar to that used in the earlier proofs with one exception: in this case, we associate each subtask with only one *execution phase*. When a suspension follows an execution phase, we insert an IS delay after the last subtask window of that phase to ensure that the next phase does not begin until after the suspension has ended. We describe this approach in more detail below.

Consider $T^{[1]}$. (Phase notation was defined earlier on pages 7 and 11.) For simplicity, assume $T^{[1]} \in \mathcal{C}(\emptyset)$. This phase must be associated with at least $\left\lceil \frac{T^{[1]}.e}{Q} \right\rceil$ subtasks. At some point at or before the extended deadline of the last subtask dedicated to this phase, denoted t_d, the suspension[7] that follows (if one indeed exists) is initiated. This suspension ends at or before time $t_d + T^{[2]}.\theta$. Hence, the first subtask associated with $J^{[3]}$ should not be released before time $t_d + \left\lceil T^{[2]}.\theta \right\rceil$ to ensure that allocated processor time is not wasted. Letting t_r denote the release of this successor subtask, this last restriction implies that $t_r = t_d + \left\lceil T^{[2]}.\theta \right\rceil$ is desired. By Lemma 3.2, $t_r \geq t_d - \left(\left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1 \right)$ when the release is not delayed. It follows that a delay of $\left\lceil T^{[2]}.\theta \right\rceil + \left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1$ is sufficient to ensure the desired separation.

Using the above approach, we can insert IS delays to account for all suspensions. We now explain how a weight can be selected for U when this approach is used. As in Lemma 3.4, the windows associated with each job must span at most $\min\left(\lfloor T.d + T.c \rfloor, T.p + \mathcal{B}(U)\right)$ slots. However, this case differs from that in Lemma 3.4 in that this span reflects the span *after* inserting all IS delays. To select a task weight, we must determine the maximum span when no delays are present.

Inserting a delay has the effect of shifting all future releases and deadlines by the magnitude of the delay. Since the delay inserted for a specific suspension phase $T^{[i]}$ has a magnitude of $\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta - 2}{U.w} \right\rceil + \epsilon + 1$ (as explained above), it follows that the total expansion of the span caused by all delays in a single job is given by $\sum_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta - 2}{U.w} \right\rceil + \epsilon + 1 \right)$. Hence, the span without these delays must be at most $\min\left(\lfloor T.d + T.c \rfloor, T.p + \mathcal{B}(U)\right) -$

[7] Since critical sections are not considered, each execution phase must be followed by a suspension phase. (Consecutive phases of the same type are assumed to be merged.)

$\sum_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{U.w} \right\rceil + \epsilon + 1 \right)$. The remainder of the theorem follows as in Lemma 3.4.

$\square$

Figure 3.6(c) shows the result of applying Lemma 3.6. In this example, $T = \mathbf{P}(5, 3.2Q, 20, 18) : \mathbf{C}(\emptyset, 2.1Q)\,; \mathbf{I}(3.2)\,; \mathbf{C}(\emptyset, 1.1Q)$. By Lemma 3.6, $U.w = \frac{\lceil 2.1 \rceil + \lceil 1.1 \rceil}{\min(\lfloor 18 \rfloor - 1, 20) - (\lceil 3.2 \rceil + 0 + 1 + 1)} = \frac{5}{11}$.

Lemma 3.7 *If T is an independent multi-phase periodic or sporadic task, then each job of T will complete within $T.c$ slots of its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\sum_{T^{[i]} \in \mathcal{C}(\emptyset)} \left\lceil \frac{T^{[i]}.e}{Q} \right\rceil}{\min\left(\lfloor T.d + T.c \rfloor - \mathcal{B}(U)\,, \lfloor T.p \rfloor\right) - 1 - \sum_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{U.w} \right\rceil + \epsilon + 1 \right)},$$

provided that $U.w \in (0, 1]$.

Proof. This proof follows the same reasoning as Lemmas 3.5 and 3.6. Specifically, the windows associated with each job must span no more than $\min\left(\lfloor T.d + T.c \rfloor, \lfloor T.p \rfloor + \mathcal{B}(T)\right) - 1$ slots by reasoning identical to that in Lemma 3.5. As in Lemma 3.6, this span must be shortened to account for inserted delays. The remainder of the proof follows as in Lemma 3.6.

$\square$

In Figure 3.6(d), $T = \mathbf{S}(5, 3.2Q, 20, 18) : \mathbf{C}(\emptyset, 2.1Q)\,; \mathbf{I}(3.2)\,; \mathbf{C}(\emptyset, 1.1Q)$. By Lemma 3.7, $U.w = \frac{\lceil 2.1 \rceil + \lceil 1.1 \rceil}{\min(\lfloor 18 \rfloor - 1, 20) - 1 - (\lceil 3.2 \rceil + 0 + 1 + 1)} = \frac{5}{10} = \frac{1}{2}$.

Implications. As these last two lemmas demonstrate, suspensions are costly and should be avoided whenever possible. The cost of suspensions under Pfair scheduling is actually not surprising since this approach strives to sustain an approximately uniform rate of execution for each task. When a task suspends, its rate is forced to zero for the duration of the suspension.

Consequently, the scheduler must execute the task at a higher rate when ready in order to compensate for these interruptions. For this reason, performance will degrade quickly as the frequency and/or duration of suspensions (in the global schedule) increases. Providing improved handling of special types of suspensions, such as I/O suspensions, is one of the most crucial remaining challenges in this area.

3.4 Using Servers[8]

In this section, we provide an overview of Pfair server tasks. Many extensions to Pfair scheduling can be supported through the use of such servers, including the scheduling of one-shot and time-sharing tasks. Indeed, the supertasking approach, which is considered later in Chapter 4, is based on the use of servers. In addition, we consider using servers to provide task synchronization later in Chapter 6.

3.4.1 Provisioning Servers

Each server maintains a *request queue* and services submitted requests using an internal policy. Two decisions must be made when using a server: what weight to assign and how the server should react to changes in its request queue. We discuss each of these issues below.

Assigning a weight. Two options are available when assigning a server weight. First, the server can be assigned a fixed weight, which never changes once assigned. Second, the server can vary its weight at runtime. Unfortunately, Srinivasan and Anderson's work on dynamic task sets suggests that instantaneous weight changes can lead to deadline misses [62].[9] To

[8]Some of the work presented in this section was previously published in the following paper:
[64] A. Srinivasan, P. Holman, and J. Anderson. Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 19–28, 2002.

[9]A weight change can be viewed as a task leaving with the old weight and rejoining with the new weight.

avoid such a miss, weight changes must be delayed. As a result, the second option will likely be impractical in most cases unless weight changes are infrequent. For this reason, we restrict attention to static weights when discussing servers in this dissertation.

Assigning static weights generally consists of four steps. First, requests are divided into multiple groups, each of which should be serviced from a separate queue. Second, each group is assigned a fraction of the available processor capacity. Third, one or more server tasks is created to service requests in each queue. Finally, the processor capacity assigned to each queue is divided among its associated servers.

We let W denote the processor capacity assigned to a specific queue, where $0 < W \leq M$. Two basic assignment rules exist for creating servers and dividing this capacity among them:

- *Minimize Latency* (*ML*): Create $\lfloor W \rfloor$ server tasks with unit weight and one server task with weight $W - \lfloor W \rfloor$ (if this value is non-zero).

- *Maximize Parallelism* (*MP*): Create n servers with weight $\min\left(\frac{W}{n}, 1\right)$ each, where k is the maximum length[10] of the queue and $n = \min(k, M)$.

The (ML) rule uses unit-weight server tasks to guarantee $\lfloor W \rfloor$ quanta in each slot. The disadvantage of this approach is that at most $\lceil W \rceil$ requests can be simultaneously serviced. Consequently, processors may idle even when unserviced requests exist. The (MP) rule guarantees that if a processor is unassigned in a slot, then all requests in the queue are being serviced within that slot. However, due to the use of lower server weights, requests may experience longer waiting times under this rule.

Server policies. A *server policy* defines the server's reactions to changes in the request queue. Specifically, a server policy defines the server's behavior in the following two scenarios:

[10]This length is determined by the number of requests that can exist simultaneously in the queue.

1. the server is scheduled while its request queue is empty;
2. the server is ineligible while its request queue is non-empty.

In the first scenario, three actions are currently supported by the Pfair task models.

Idle: the server accepts the allocated quantum, thereby consuming its subtask, and idles the processor.

Stall: the server refuses the allocated quantum, which is allocated to some other task, and sets the release of the associated subtask to the next slot boundary.

Drop: the server refuses the allocated quantum, which is allocated to some other task, and marks the associated subtask as being absent.

Each of these behaviors reflects a task model. Specifically, Idle corresponds to the Pfair model, Stall corresponds to the ISfair model, and Drop corresponds to the GISfair model.

Figure 3.7 illustrates each of the above policies.[11] In this example, a server task S competes for a single processor against two other tasks T and U. Five one-shot tasks (jobs) must be serviced by S: $J = \mathbf{J}(3.5, 0.5Q, 8.5)$, $K = \mathbf{J}(10.25, 0.25Q, 13.75)$, $L = \mathbf{J}(2, 0.25Q, 10)$, $M = \mathbf{J}(11, 0.25Q, 15)$, and $N = \mathbf{J}(11, 1.75Q, 40)$. These jobs are scheduled by a fully preemptive EDF policy. (This scenario is also considered in Figure 3.8.)

At time 1, S is scheduled, but has an empty request queue. The Idle policy accepts the quantum and simply idles, as shown in Figure 3.7(a). Although this seems wasteful, it is possible that a request may be issued during that quantum. For instance, this occurs in slot 10: S idles until K's arrival at time 10.25. The Stall (respectively, Drop) policy avoids taking

[11]In these examples, the scheduling of subtasks is denoted by placing a square in the appropriate slot of each subtask's window. These squares denote *quanta* of processor time rather than the total processor time in the slot, *i.e.*, we do not explicitly show processor time consumed by scheduler invocations and other overheads.

the quantum by delaying the subtask release until the next slot (respectively, by marking the subtask absent). These actions are illustrated by insets (b) and (c) of Figure 3.7, respectively.

Two actions are supported by the Pfair task models for the second scenario (*i.e.*, the server is ineligible while its request queue is non-empty).

Wait: the server does nothing and waits for its next subtask release.

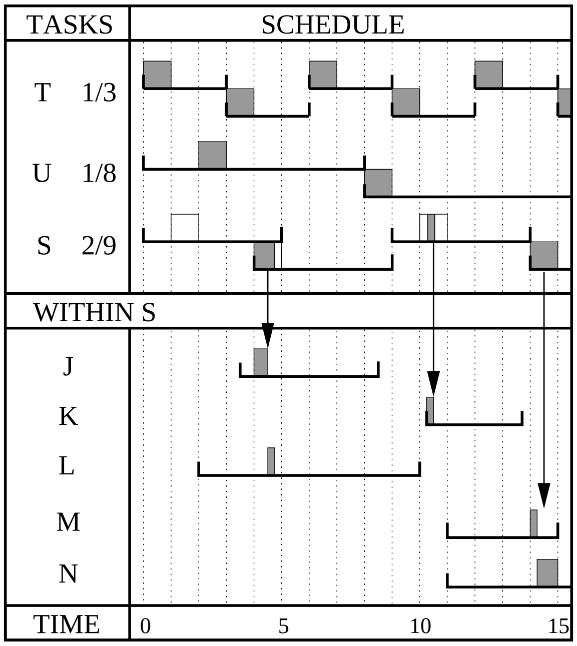
Early Release: the server early-releases subtasks when requests exist.

Again, each of these options represents a Pfair task model. Specifically, Wait corresponds to the Pfair model, while Early Release (obviously) corresponds to the ERfair model.

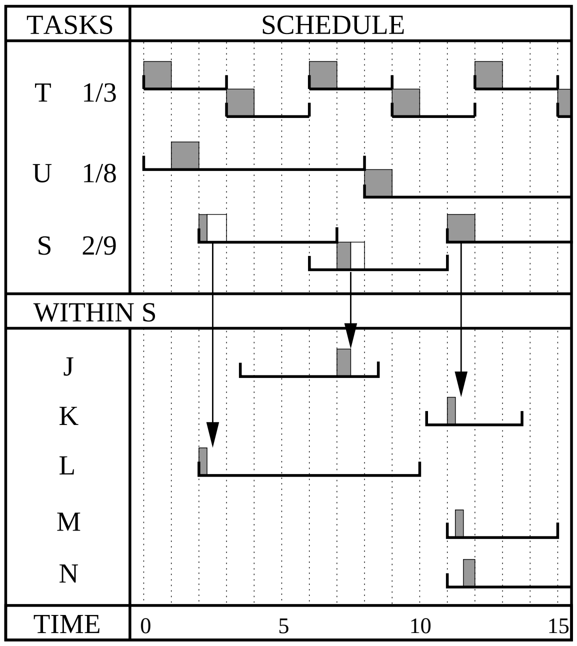
All of the schedules shown in Figure 3.7 assume a Wait policy. Figure 3.8(a) illustrates the use of early releasing with an Idle policy. At time 2, job L arrives and prompts S to release S_2 early. However, S_2 is still not scheduled until time 4. On the other hand, early releasing S_4 and S_5 speeds the servicing of both M and N in slots 11 and 13.

Idle capacity. Performance can be improved when idle capacity exists by allowing servers to quick-release their subtasks. Quick-releasing is advisable as a supplementary measure since it allows idle quanta to be utilized without penalizing the executing task to the same degree as early-releasing. Figure 3.8(b) shows the result of applying an Idle/Wait policy when quick releasing is permitted.

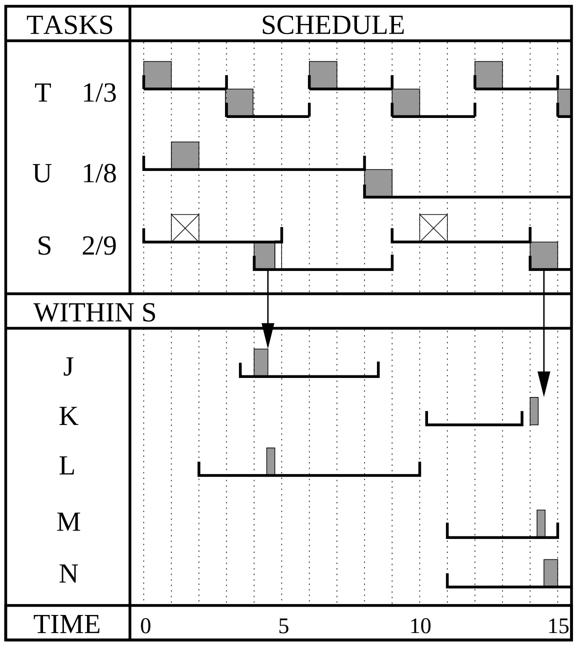
Alternatively, background scheduling can be used to assign idle processors to unserviced requests without any penalty. The primary advantage of quick releasing over background scheduling is that quick releasing allocates idle capacity more fairly on average and is capable of enforcing maximum shares [2]. Figure 3.8(c) shows the result of applying an Idle/Wait policy along with background scheduling. (The background scheduler is denoted BG.)



(a) Idle



(b) Stall



(c) Drop

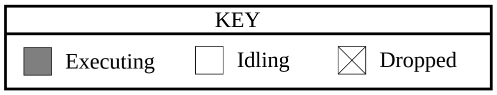
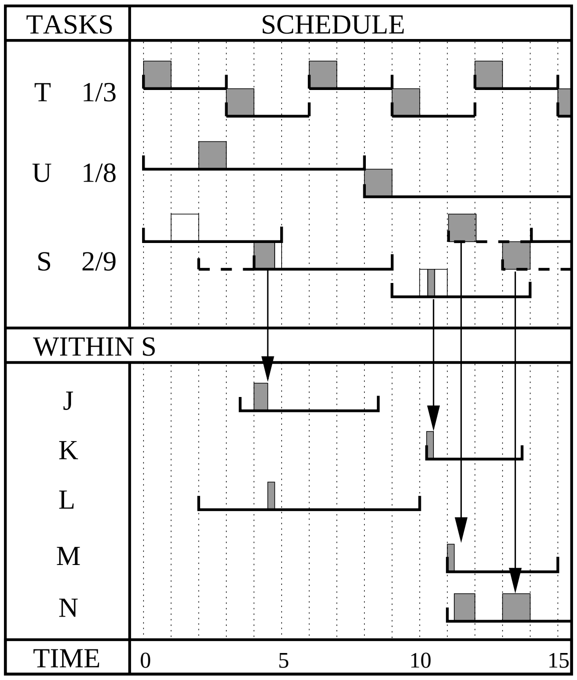
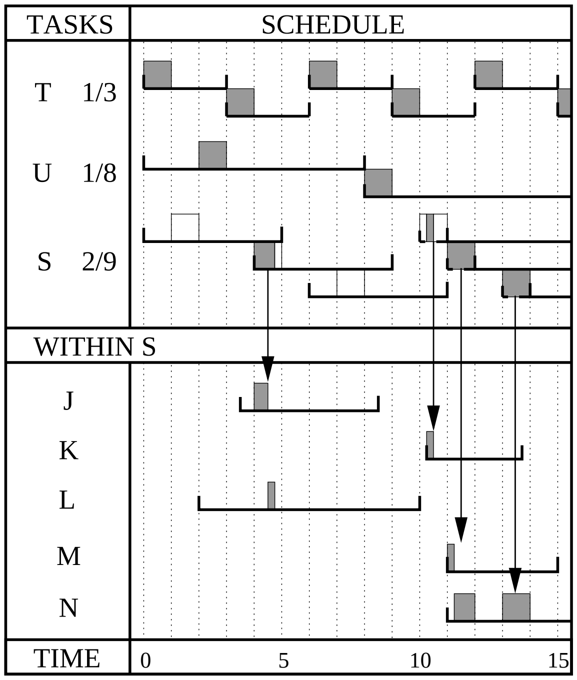


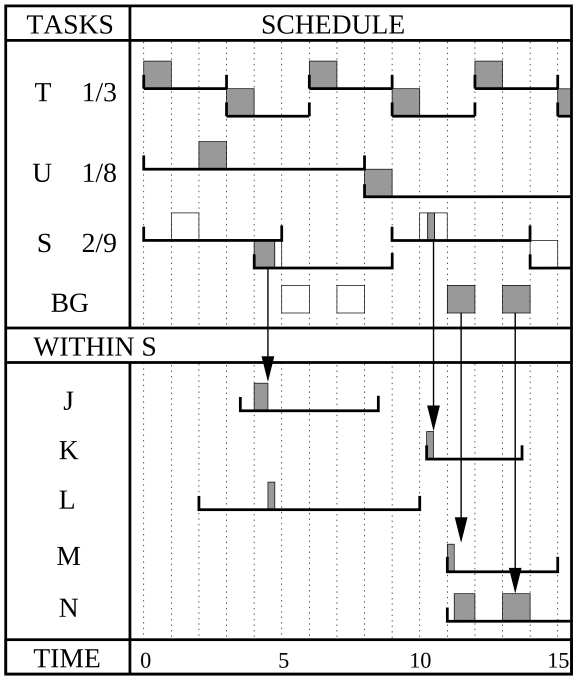
Figure 3.7: Illustration of possible server reactions to being scheduled while the request queue is empty. Insets show the **(a)** Idle, **(b)** Stall, and **(c)** Drop policies.



(a) Early Release



(b) Quick Release



(c) Background Scheduling

Figure 3.8: Illustration of server behaviors that can be used to speed servicing of requests while ineligible. Techniques include **(a)** early releasing, **(b)** quick releasing, and **(c)** background scheduling.

3.4.2 Worst-case Responsiveness

We now present a lemma and corollaries that characterize the worst-case responsiveness of a server task. Here, "responsiveness" refers to the amount of time that elapses before a server task receives a specified amount of processor time. To simplify the presentation of the results below, we let $\mathit{span}(T, n)$ denote the worst-case n-span of T, as shown below.

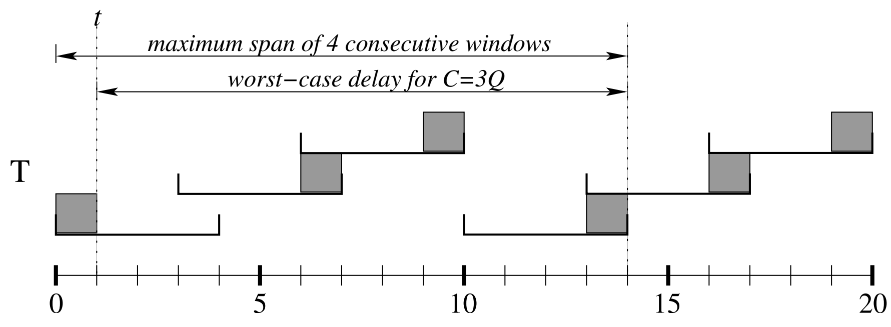
$$\mathit{span}(T, n) = \left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon + 1$$

The above formula follows trivially from Lemma 3.3. In addition, we say that a server S is *inactive* at time t if S has postponed the release of a subtask until the slot after that containing t. Otherwise, we say that S is *active* at t. (S can only be inactive under the Stall policy.)

Lemma 3.8 *If a Pfair-scheduled server task T is active at time t, then the amount of time required for it to receive C units of processor time, provided the server remains active until doing so, is upper-bounded by*

$$\mathit{span}\Big(T, \left\lceil \frac{C}{Q} \right\rceil + 1\Big) - 1 - \Big(\left\lceil \frac{C}{Q} \right\rceil Q - C\Big).$$

Proof. The worst-case delay occurs when T's allocation is delayed as long as possible without violating its constraints. This occurs when a subtask of T executes in the first slot of its window, t corresponds to the end of that slot, and every subsequent subtask executes in the latest possible slot. Figure 3.9 illustrates this scenario for the $C = 3Q$ case. Under such conditions, q quanta are received after a delay of at most $\mathit{span}(T, q + 1) - 1$. Hence, C units of processor time will be received by time $t + \mathit{span}\Big(T, \left\lceil \frac{C}{Q} \right\rceil + 1\Big) - 1$. This estimate rounds up



t
maximum span of 4 consecutive windows
worst−case delay for C=3Q
T
0
5
10
15
20

Figure 3.9: Illustration of the worst-case delay from t required for a server with weight $\frac{3}{10}$ to receive $C = 3Q$ units of processor time when the server is active at t.

C to the next quantum multiple, which results in $\left\lceil \frac{C}{Q} \right\rceil Q - C$ more units of processor time than requested. The actual worst-case response time is obtained by subtracting this amount off of the previous estimate, producing the formula stated in the lemma. $\square$

Lemma 3.9 *If a Pfair-scheduled server task T is inactive at time t, then the amount of time required for it to receive C units of processor time, provided the server remains active until doing so, is upper-bounded by*

$$\mathit{span}\left(T, \left\lceil \frac{C}{Q} \right\rceil\right) + 1 - \left(\left\lceil \frac{C}{Q} \right\rceil Q - C\right).$$

Proof. Again, the worst-case delay is produced by scheduling all subtasks of T scheduled after t in the last slots of their respective extended windows. If T is inactive at t and receives a request at that time, then it will release its next subtask at the next slot boundary, as planned. Figure 3.10 illustrates this scenario for the $C = 3Q$ case. Since this next slot boundary must occur within one slot, T is guaranteed to receive C units of processor time by $t + \mathit{span}\left(T, \left\lceil \frac{C}{Q} \right\rceil\right) + 1$. As in the previous proof, this estimate overestimates the response time by $\left\lceil \frac{C}{Q} \right\rceil Q - C$ and is similarly adjusted. $\square$

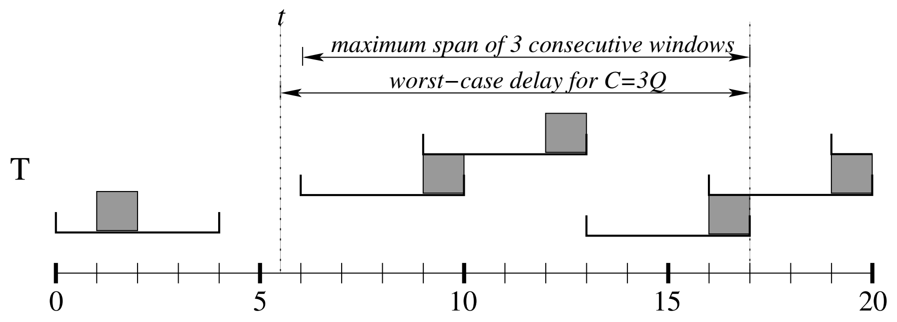

t
maximum span of 3 consecutive windows
worst−case delay for C=3Q
T
0
5
10
15
20

Figure 3.10: Illustration of the worst-case delay from t required for a server with weight $\frac{3}{10}$ to receive $C = 3Q$ units of processor time when the server is inactive at t.

Corollary 3.2 *The amount of time required for a Pfair-scheduled server task T to receive C units of processor time, provided it remains active until doing so, is upper-bounded by*

$$\mathit{span}\left(T, \left\lceil \frac{C}{Q} \right\rceil + 1\right) - 1 - \left(\left\lceil \frac{C}{Q} \right\rceil Q - C\right).$$

Proof. Since the server must be either active or inactive at t, the worst-case delay before receiving C units of processor time can be determined by taking the maximum of the bounds given in Lemmas 3.8 and 3.9. The corollary follows. □

3.4.3 Acceptance Test for One-shot Real-time Tasks

In this subsection, we present a simple approach by which one-shot real-time tasks can be scheduled using server tasks. Upon arrival, each job (*i.e.*, one-shot task) is subjected to an *acceptance test* to determine whether all deadlines can be met if the job is allowed to join the server queue. If so, the job is accepted and stored in the queue. Otherwise, the job is rejected. Since the handling of rejections is a policy decision that is related to the purpose of the job, we do not discuss this aspect of acceptance testing here. In the remainder of this subsection, we present a simple acceptance test to support one-shot tasks. This test assumes that jobs that are ready to execute immediately upon arrival and are scheduled using a fully preemptive EDF policy. Extending this test to other cases is straightforward.

Worst-case responsiveness. Although the responsiveness bounds given in the lemmas above could be used to bound the response time of accepted jobs, more accurate accounting is possible by considering the runtime state of the server. We assume each server task T maintains the following state information:

- θ: the total IS delay of T experienced up to the current time.
- q: the amount of processor time remaining in T's current quantum, or zero if T is not currently executing.[12]
- i: the index of T's next subtask, *i.e.*, the lowest-indexed present subtask that has not started executing.

The theorem that follows characterizes the worst-case responsiveness of a server as a function of this runtime state.

Theorem 3.1 *A server task T that is characterized by θ, q, and i at time t is guaranteed to receive C units of processor time by time*

$$\theta + d\left(T_{i+\lceil\frac{C-q}{Q}\rceil-1}\right) - \left(\left\lceil\frac{C-q}{Q}\right\rceil Q + q - C\right)$$

if $C > q$ and by time

$$t + C$$

[12]This value can be approximated based on the current time, the time at which the current slot ends, and an estimate of the maximum processor time lost to overhead in between. In practice, q would be computed only when needed. We assume it is maintained only to simplify the test presented later.

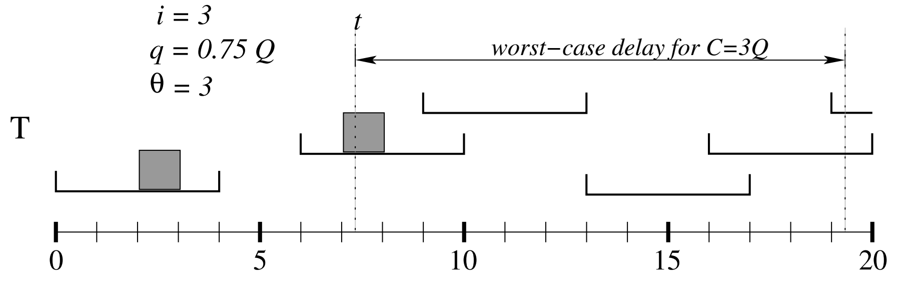


Figure 3.11: Illustration of the worst-case delay from t required for a server with weight $\frac{3}{10}$ to receive $C = 3Q$ units of processor time. The estimate is based on the server's runtime state.

if $C \leq q$*, provided the server remains active until doing so.*

Proof. The $C \leq q$ case is trivial since T executes continuously. Hence, consider the $C > q$ case. Figure 3.11 illustrates the worst-case delay of a server with weight $\frac{3}{10}$ for the $C = 3Q$, $\theta = 3$, $i = 3$, and $q = \frac{3}{4}Q$ case. The net effect of the θ parameter is to shift all future extended deadlines (*i.e.*, those with indexes at least i) to the right by θ slots. Therefore, T is guaranteed to receive k full quanta by time $\theta + d(T_{i+k-1})$, resulting in a total allocation of $k \cdot Q + q$ by that time. It follows that $\left\lceil \frac{C-q}{Q} \right\rceil Q + q$ units of processor time are received by time $\theta + d\left(T_{i+\left\lceil \frac{C-q}{Q} \right\rceil - 1}\right)$. However, since $\left\lceil \frac{C-q}{Q} \right\rceil Q + q \geq C$, only part of the last-received quantum may be needed. Specifically, an excess of $\left\lceil \frac{C-q}{Q} \right\rceil Q + q - C$ is received after the required C units. Subtracting off this amount yields the bound given in the theorem. $\square$

Acceptance test. A simple on-line acceptance test based on the above theorem is shown in Figure 3.12. This algorithm stores the one-shot tasks in a list *jobset* ordered non-decreasingly by absolute deadline. When a new task T arrives, `AdmissionTest` is invoked on T to perform the acceptance test. The test consists of checking T's deadline, denoted $T.deadline$, and all later deadlines, in non-decreasing order, to ensure that the total processor time required by the one-shot tasks by each deadline does not exceed the amount of time guaranteed to the server task. The required time is computed by summing the *remaining* execution time of each task U, denoted $U.execnow$. If the invocation returns true, then T was accepted and stored in the list during the invocation. Otherwise, T was rejected and the list is restored to its original state.

In `AdmissionTest`, the server task is left implicit and `Response` represents the response-time bound given in Theorem 3.1. All other procedures are list operations. A constant time complexity is achievable for all procedures by implementing each list as a queue.

Detailed description. An invocation of `AdmissionTest` begins by calculating the total processor time required by all requests with deadlines at or before $T.deadline$ (lines 1–6). As these requests are processed (lines 4–6), they are moved from *jobset* into *before*. Since these requests were previously accepted, they do not require checking. In lines 7–8, T's execution requirement is added to C, and D is initialized to $T.deadline$, which is the first deadline that must be checked. Lines 9–14 then check that sufficient processor time is guaranteed by each deadline of T and the previously accepted tasks. Specifically, the `Response`$(C) \leq D$ condition in line 10 causes the loop in lines 10–14 to immediately terminate if it is determined that the server cannot guarantee a deadline. When `Response`$(C) \leq D$ holds, the accepted task with the next deadline is considered in lines 11–13 and then moved into the *after* list in line 14. Line 15 then moves all unprocessed tasks into *after* so that all previously accepted tasks reside in either *before* or *after*. (Unprocessed tasks exist at line 15 only when T is rejected.) If the final comparison succeeds (line 16), then T is accepted in lines 17–19. Otherwise, T is rejected in lines 20–21. Each routine invocation has time complexity $O(|jobset|)$.

Remarks. The acceptance test presented here operates by allowing one-shot tasks to

```
typedef nodetype
    record
        begin
            deadline: real;
            execnow: real
        end

var
    jobset, before, after: list of nodetype ordered by non-decreasing deadline;
    T, U: nodetype; C, D: real

procedure AdmissionTest(jobset, T) returns boolean
1:  $C := 0$;
2:  before := MakeList();
3:  while Size(jobset) $> 0 \wedge$ Head(jobset).deadline $\leq T.deadline$ do
4:      $U$ := RemoveFromHead(jobset);
5:      $C := C + node.execnow$;
6:      InsertAtTail(before, $U$)
    od;
7:  $C := C + T.execnow$;
8:  $D := T.deadline$;
9:  after := MakeList();
10: while Size(jobset) $> 0 \wedge$ Response($C$) $\leq D$ do
11:     $U$ := RemoveFromHead(jobset);
12:     $C := C + U.execnow$;
13:     $D := U.deadline$;
14:     InsertAtTail(after, $U$)
    od;
15: after := Concatenate(after, jobset);
16: if Response($C$) $\leq D$ then
17:     Enqueue(before, $T$);
18:     jobset := Concatenate(before, after);
19:     return true
    else
20:     jobset := Concatenate(before, after);
21:     return false
    fi
```

Figure 3.12: Online acceptance test for one-shot real-time tasks.

consume *slack* time (*i.e.*, excess processor time) that exists over an interval. This approach, called *slack stealing*, is quite common under both time-driven and event-driven scheduling.

For instance, similar algorithms are described by Liu in [45].

3.5 Summary

This chapter outlined the design of a Pfair-scheduled operating system and addressed some of the simpler challenges posed by such a design. First, we discussed the implementation of quantum-based scheduling and the impact it has on the assumptions underlying Pfair scheduling. Second, we introduced a model for describing the guarantees provided by global schedulers based on the Pfairness constraint and its relaxed variants. We then derived basic properties of the global scheduler using the model's parameters. This model permits work presented here to be applied under all of the known Pfair policies without requiring re-derivation of results. Third, we derived sufficient mapping rules for scheduling independent periodic and sporadic tasks, both with and without suspensions. Later, Chapters 5 and 6 further extend these rules to support task synchronization. Finally, we discussed the use of server tasks under Pfair scheduling and presented a simple acceptance test for one-shot tasks. Chapter 4 continues the discussion of server tasks by extending the supertasking approach.

CHAPTER 4

Hierarchal Scheduling [1]

In this chapter, we extend the supertasking approach proposed by Moir and Ramamurthy [48]. We present four primary contributions. First, we show that scheduling within a supertask is analogous to scheduling on a dedicated uniprocessor. Second, we identify the cause of the component-task timing violations observed in [48]. Third, we present a flexible framework for selecting a supertask weight that guarantees the timeliness of its component tasks. We demonstrate the utility of this framework by considering several common scenarios. Finally, we compare and contrast Pfair scheduling with supertasks to partitioning. As we explain, the two approaches are quite similar conceptually. Because of this, many benefits of partitioning can also be obtained through the use of supertasks.

Organization. These results are presented in six sections. Section 4.1 provides insight into supertasking and its relationship to other approaches. In Section 4.2, a sufficient schedulability condition for the component tasks is derived. In Section 4.3, this last condition is used to derive a "reweighting" condition, which is a sufficient schedulability condition in the form of a

[1] This chapter is derived from work previously published in the following papers:
[31] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, 2001.
[34] P. Holman and J. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 41–50, 2003.

collection of supertask weight restrictions. Section 4.4 focuses on selecting a "safe" supertask weight, *i.e.*, one that satisfies the reweighting condition and hence guarantees schedulability. Section 4.5 then presents and proves basic properties of an algorithm for selecting safe supertask weights. Section 4.6 presents the results of an experimental evaluation of supertasking.

4.1 Understanding Supertasking

Supertasking is actually a natural extension of Pfair scheduling. Baruah, Cohen, Plaxton, and Varvel [11] observed that a task set τ consisting of independent synchronous tasks that never suspend can be successfully scheduled by assigning each task a dedicated processor that executes at exactly the rate needed by that task. Under such a system, processor speeds vary. Pfair scheduling, by design, *simulates* such a system by time-sharing "virtual" processors (*i.e.*, the Pfair tasks) among M identical processors. Each Pfair task effectively acts as a dedicated processor for the periodic task that it executes.

Virtual processors differ from dedicated processors in that the amount of processor time available to the task set varies over time. Proper analysis requires that these variations be bounded and predictable. Under Pfair scheduling, this variance is a function of the assigned weight and the guaranteed lag bounds, as we later show. When a unit weight is assigned, a virtual processor perfectly imitates a dedicated processor.

Related concepts. Using server tasks (*i.e.*, virtual processors) to multiplex several applications onto a single platform is a relatively old idea. For instance, in 1989, Tucker and Gupta suggested using virtual processors to more seamlessly support workload changes when multiplexing parallel applications onto a multiprocessor [67]. In addition, thread packages, which are now commonly available, are a direct application of the virtual-processor concept.

For instance, Java programs are executed on a virtual processor referred to as the *Java virtual machine* (or JVM) [14].

Using servers as virtual processors is a central concept in work on *open systems*. In open systems, independently developed applications share one or more physical processors and must be isolated from each other. Server approaches, such as Abeni and Buttazzo's constant-bandwidth server (CBS) [1], are ideal for this purpose because they provide a layer of abstraction between the application and the underlying operating system. Not surprisingly, a substantial body of work has focused on the use of CBS and related approaches in open systems (*e.g.*, [40, 15]). The primary difference between the CBS and supertasking approaches is the task model of the server; the CBS approach is based on the periodic and sporadic models while supertasking is based on the Pfair task models.

Supertasking and partitioning. Supertasking extends the one-to-one relationship between virtual processors and tasks (considered in [11]) into a one-to-many relationship. In doing so, a new problem is introduced: how should tasks be grouped? In practice, tasks may not always be implicitly grouped. In such cases, tasks can be artificially grouped in order to reduce contention and overhead. Notice that dividing tasks among groups bears a strong resemblance to partitioning approaches (discussed earlier in Chapter 1).

The primary difference between supertasking and partitioning is that partitioning binds tasks to physical processors, while supertasking binds tasks to virtual processors. Under partitioning, exactly M processors are available, each with unit capacity. Neither of these values (*i.e.*, the processor count and processor capacity) can be varied. On the other hand, the number of supertasks is unrestricted and capacities (*i.e.*, weights) can be assigned *after* making task assignments. Hence, the assignment of tasks to supertasks can be accomplished through an algorithm of the following form:

1. Compute initial weights for all tasks in τ.
2. Create a set π of $|\tau|$ empty supertasks.
3. Assign $\mathcal{S}.w = 1$ to each supertask $\mathcal{S} \in \pi$.
4. Apply a heuristic to assign all tasks in τ to supertasks in π.
5. Remove all empty supertasks from π.
6. Update parameters for all tasks in τ (*e.g.*, weights, blocking estimates, *etc.*) based on assignments.
7. Update weights for all supertasks in π based on assignments.
8. If any supertask weight exceeds unity, discard π and return to Step 2.

Step 7 in the above algorithm is the primary difference between supertasking and partitioning. In this step, supertask capacities are effectively "shrunk" down to the size of the assigned component task set. Such an action is clearly not possible when assigning tasks directly to physical processors.

4.1.1 Benefits of Supertasking

The benefits of supertasking vary with the internal scheduling policy. In this chapter, we consider both quantum-based and fully preemptive policies. In addition, we assume that all supertasks use either an Idle-and-Wait or a Drop-and-Wait server policy[2] (see Chapter 3).

Quantum-based supertasking. When using a quantum-based policy with a supertask, we assume that the supertask slot size matches that of the global scheduler. Hence, component

[2]The presented analysis works for either policy.

tasks are also subdivided into subtasks, just like Pfair tasks. Figure 2.5, shown earlier, illustrates such a policy. When considering this supertasking approach, we assume that $T.c$ (*i.e.*, T's tardiness threshold) is an integer for each component task T. (If not, $\lfloor T.c \rfloor$ can be substituted in place of the actual value.)

The advantages and disadvantages of quantum-based supertasking are basically the same as for any form of quantum-based scheduling. As explained earlier in Chapter 1, the primary advantage is predictability, while the primary disadvantage is the potential for wasted processor time due to both partially used quanta and unnecessary scheduler invocations. However, quantum-based supertasking provides an additional advantage: at most one component task executes within each slot. Consequently, supertasking can be used to reduce the worst-case per-slot contention for a shared resource by restricting parallelism. We discuss this application of supertasking in more detail later in Chapters 5 and 6.

Fully preemptive supertasking. The second form of supertasking that we consider is the use of supertasks that use fully preemptive policies internally. Figure 4.1 illustrates the use of fully preemptive EDF scheduling. This schedule is derived from that shown in Figure 2.5 by replacing the two Pfair component tasks with periodic tasks. Notice that the quantum allocated to the supertask in slot 1 is divided among two different tasks, which implies a context switch within that slot.

The primary advantage of fully preemptive supertasking is that its performance more closely resembles that of traditional fully preemptive scheduling. For example, global quantum-based scheduling wastes less processor time when scheduling fully preemptive supertasks. Processor time allocated to a fully preemptive supertask is wasted only when no component task is ready. For example, in Figure 4.1, idling was avoided in slot 1 by switching to $\mathbf{Y}(1)$

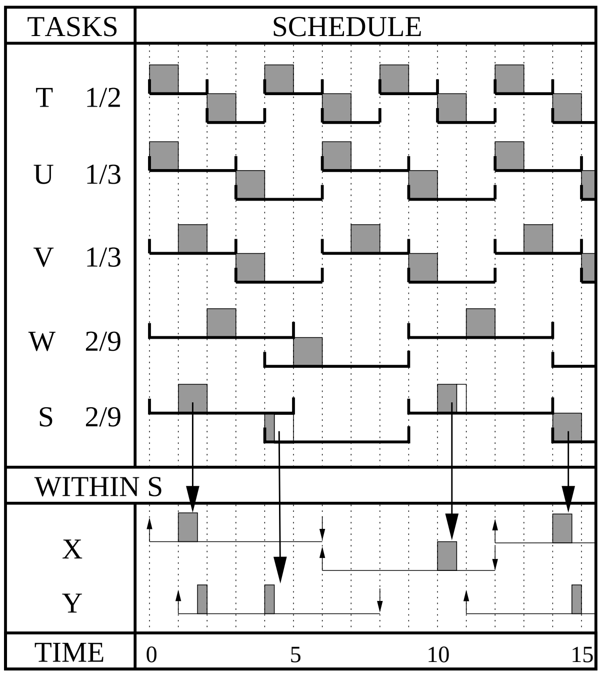


Figure 4.1: Sample supertasking schedule. Supertask S serves two periodic component tasks, $X = \mathbf{P}\Big(0, \frac{3}{4}Q, 6, 6\Big)$ and $Y = \mathbf{P}\Big(1, \frac{1}{2}Q, 10, 7\Big)$. The PD^2 policy is used globally, while the EDF policy is used within S.

upon the completion of $\mathbf{X}(1)$. (Recall that $T(i)$ denotes the i^{th} job of task T.) In general, as the size of the component task set increases, partially used quanta occur less frequently. Furthermore, as the weight of the supertask approaches unity, the supertask is preempted less frequently in the global schedule. Hence, context switching and caching behavior approaches that obtained by partitioning.

The primary disadvantages of this approach are the increased overhead and unpredictability caused by allowing scheduler invocations within slots. However, using an EDF policy somewhat compensates for these problems due to the depth of existing research, *i.e.*, techniques exist for analyzing a fully preemptive EDF policy on a dedicated uniprocessor. As the work in this chapter demonstrates, it is often straightforward to adapt these uniprocessor techniques for use within fully preemptive supertasks.

4.1.2 The Cost of Supertasking

In order to quantify the cost of using supertasks, it is necessary to first define an ideal form to use as a baseline for comparison. We define such an ideal form for both quantum-based and fully preemptive supertasks below.

Quantum-based supertasking. As observed by Moir and Ramamurthy [48], a quantum-based supertask would ideally be granted a weight equal to the cumulative weight of its component tasks. Letting $\mathcal{S}.I_Q$ denote this ideal weight results in the definition given below.

$$\mathcal{S}.I_Q \overset{\text{def}}{=} \sum_{T \in \mathcal{S}} T.w \tag{4.1}$$

Hence, the overhead resulting from the use of a quantum-based supertask is given by $\mathcal{S}.w - \mathcal{S}.I_Q$. We refer to this overhead as *inflation* or *reweighting* overhead.

Fully preemptive supertasking. The ideal supertask weight is similarly defined for fully preemptive supertasks. However, in this case, we must first consider the relationship between weight and utilization, due to the fact that component tasks are not characterized by weights. Over an interval of length L, a task with weight w is allocated approximately $w \cdot L$ quanta, which results in a total allocation of $(w \cdot L) \cdot Q$. Hence, the utilization of the task over the interval is given by $\frac{(w \cdot L) \cdot Q}{L}$, which simplifies to $w \cdot Q$. It follows that a utilization of u is achieved by a weight of $\frac{u}{Q}$.

We now use this relationship to define the ideal weight. Ideally, a supertask would be assigned the smallest weight necessary to ensure the total utilization of its component tasks.

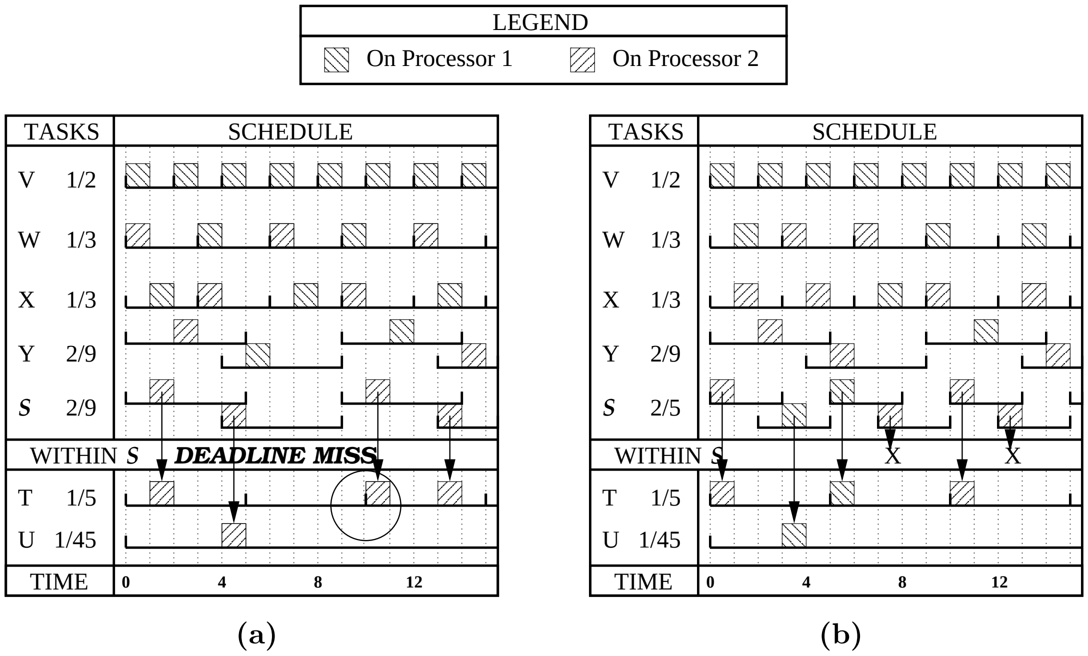


Figure 4.2: Two-processor schedule with a **(a)** normal and **(b)** reweighted supertask $\mathcal{S}$.

This utilization is given by

$$\mathcal{S}.u \overset{\text{def}}{=} \sum_{T \in \mathcal{S}} T.u.$$

Using the relationship between weight and utilization, the ideal weight, denoted $\mathcal{S}.I_P$, is defined as shown below.

$$\mathcal{S}.I_P \overset{\text{def}}{=} \frac{\mathcal{S}.u}{Q} \tag{4.2}$$

Hence, the overhead produced by a fully preemptive supertask is given by $\mathcal{S}.w - \mathcal{S}.I_P$.

4.1.3 Understanding Failures

To see why supertasking (as proposed by Moir and Ramamurthy [48]) can fail, consider the two-processor Pfair schedule shown in Figure 4.2. The task set consists of four Pfair tasks ($\mathbf{V} = \mathbf{PF}\left(\frac{1}{2}\right)$, $\mathbf{W} = \mathbf{PF}\left(\frac{1}{3}\right)$, $\mathbf{X} = \mathbf{PF}\left(\frac{1}{3}\right)$, and $\mathbf{Y} = \mathbf{PF}\left(\frac{2}{9}\right)$) and one supertask $\mathcal{S}$ that represents the two component tasks $\mathbf{T} = \mathbf{PF}\left(\frac{1}{5}\right)$ and $\mathbf{U} = \mathbf{PF}\left(\frac{1}{45}\right)$ (shown in the lower region). All tasks have miss thresholds of zero and Pfair global scheduling is assumed. In Figure 4.2(a), $\mathcal{S}$ competes with its ideal weight, *i.e.*, $\mathcal{S}.w = \mathcal{S}.I_Q = \frac{1}{5} + \frac{1}{45} = \frac{2}{9}$. All scheduling decisions in the upper region are consistent with the PD^2 policy. In the lower region, allocations within $\mathcal{S}$ are shown, which are made according to the EPDF policy.

As the schedule shows, $\mathbf{T}$ misses a pseudo-deadline at time 10. This is because no quantum is allocated to $\mathcal{S}$ in the interval $[5, 10)$. In general, component tasks may violate their timing constraints whenever there exists an interval $[t_1, t_2)$ over which the total processor time required by the component tasks, denoted $\mathit{demand}(\mathcal{S}, t_1, t_2)$, exceeds the minimum amount of processor time guaranteed to the supertask, denoted $\mathit{supply}(\mathcal{S}, t_1, t_2)$. Observe that $[5, 10)$ is such an interval since $\mathit{demand}(\mathcal{S}, 5, 10) = Q$ due to $\mathbf{T}_2$, while $\mathit{supply}(\mathcal{S}, 5, 10) = 0$. To ensure the timeliness of component tasks, it is sufficient (though not necessary in most cases) to guarantee that $\mathit{supply}(\mathcal{S}, t_1, t_2) \geq \mathit{demand}(\mathcal{S}, t_1, t_2)$ over all intervals in which a violation could potentially occur. Selecting a supertask weight that provides such a guarantee, called *reweighting*, is the goal of this chapter.

Figure 4.2(b) illustrates how reweighting can ensure timeliness. In this schedule, $\mathcal{S}.w$ has been increased to $\frac{2}{5}$, resulting in an inflation of $\frac{2}{5} - \frac{2}{9} = \frac{8}{45}$. As shown, no component task violates its timing constraints. However, an unfortunate side effect of reweighting is that a supertask will inevitably be allocated more processor time than its component tasks can utilize; quanta marked with an "X" are allocated to the supertask but cannot be used.

4.1.4 Example Scenarios

One advantage of the reweighting methodology presented here is the ease with which it can be adapted to new scheduling scenarios. Unfortunately, since each scenario typically requires unique reasoning, it is not possible to derive formulas that can be universally applied in all scenarios. To facilitate the presentation, we have tried to use variable and function definitions to isolate scenario-specific details from the parts of the methodology that are common to all scenarios.

However, explaining the methodology without providing any concrete examples to illustrate each step can also be confusing. As a compromise, we have chosen five example scenarios that will be used to illustrate the application of our methodology (*i.e.*, we demonstrate the derivation of the scenario-specific details for each of these five scenarios). These examples have been selected to highlight common problems and to provide a reasonable coverage of the issues involved when reweighting. Specifically, we consider the following scenarios:

Scenario 1: *Quantum-based EPDF scheduling* (QB-EPDF)

Component tasks are GIS tasks and are scheduled by a quantum-based supertask. Hence, this scenario applies to both rate-based tasks and mapped tasks produced by Lemmas 3.4–3.7. Subtasks are prioritized by the EPDF policy. The global scheduler is assumed to respect Pfairness, *i.e.*, $\beta_{-} = \beta_{+} = 1$ and $\epsilon_r = \epsilon_d = 0$.

Scenario 2: *Quantum-based EDF scheduling* (QB-EDF)

Under this scenario, tasks are characterized by weights and are scheduled by a quantum-based supertask. However, unlike the previous scenario, subtasks are not assigned releases and deadlines based upon lag constraints. Instead, each $T.J$ consecutive subtask cycles are taken to represent a logical job of T, called a *pseudo-job*; the release and deadline of each subtask is set to coincide with that of the pseudo-job with which it is associated. (The term "pseudo-job" is used to avoid confusion with jobs of periodic and sporadic tasks.) As in the sporadic task model, the release of a pseudo-job can be delayed. However, releases are constrained to occur on slot boundaries.

Figure 4.3 illustrates the relationship between jobs, subtasks with lag-based windows, and subtasks with pseudo-job-based windows. Task $\mathbf{T}$ is a sporadic task, while task $\mathbf{U}$ is a Pfair task, *i.e.*, windows of $\mathbf{U}$ are defined by the Pfair lag constraint. Task $\mathbf{U}$ was derived from $\mathbf{T}$ by using the mapping given by Lemma 3.4 on page 73. By this mapping, two window cycles service each job of $\mathbf{T}$, as shown. Consider subtasks $\mathbf{U}_1$–$\mathbf{U}_4$. Notice that enforcing the deadlines of subtasks $\mathbf{U}_1$–$\mathbf{U}_3$ and the releases of subtasks $\mathbf{U}_2$–$\mathbf{U}_4$ is not necessary to satisfying the job constraints of T. In short, $\mathbf{U}$ is over-constrained relative to $\mathbf{T}$. As we later show, these additional constraints imposed by $\mathbf{U}$ translate into increased inflation. This inflation can be reduced by setting all subtask releases and deadlines to the earliest and latest times, respectively, that do not produce a timing violation. In this example, the releases cannot be set before time 5, while the deadlines must occur at or before time 23. Hence, we say that $\mathbf{V}$ has a pseudo-job spanning $[5, 23)$. As shown in Figure 4.3, all four subtasks associated with this pseudo-job have releases and deadlines corresponding to that of the pseudo-job. (Subtasks must still be executed in order and cannot execute in parallel.) Since each pseudo-job spans two cycles, $\mathbf{V}.J = 2$ in this example.

Scenario 3: *Fully preemptive EDF scheduling* (FP-EDF)

Component tasks are independent sporadic tasks that never suspend; these tasks are scheduled by a fully preemptive EDF policy within the supertask.

Scenario 4: *EDF scheduling with nonpreemptable code segments* (FP-EDF-NP)

This scenario extends Scenario 3 by providing support for nonpreemptable execution within the supertask.[3] We let $U.v$ denote an upper bound on the execution requirement

[3]We do *not* consider making the supertask nonpreemptable since that would also impact global scheduling.

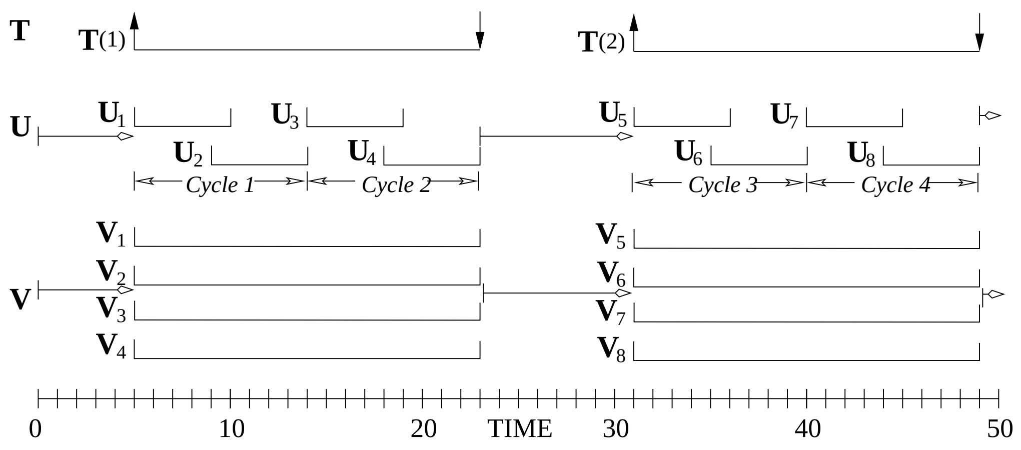


Figure 4.3: Illustration of the three different models for the sporadic task $T = \mathbf{S}(5, 4Q, 18, 18)$ considered in the example scenarios. Job releases are constrained to fall on slot boundaries. Both task U and task V have weights of $\frac{2}{9}$. However, the windows of task U are assigned based on lag constraints, while those of task V are assigned according to V's pseudo-jobs, where $V.J = 2$.

of any nonpreemptable code segment of task U. This scenario is a generalization of fully nonpreemptive scheduling (within the supertask), which can be achieved by letting $U.v = U.e$ for all $U \in \mathcal{S}$.

Scenario 5: *EDF scheduling with locking synchronization* (FP-EDF-CS)

This scenario extends Scenario 3 by providing support for locking synchronization within the supertask. The use of a locking protocol that satisfies the following behavioral properties is assumed:

- Locks are granted to jobs according to scheduling priority.
- A lock is granted only when the requesting job is capable of immediately executing, *i.e.*, the task must have the highest scheduling priority in the supertask and the supertask must have a processor.
- When a job holds no locks, it executes using its normal priority.[4]

[4]The *normal priority* of a job is the priority assigned to it by the scheduling policy. In this case, the

- When a job holds a lock, it executes at a priority level that is at least that of each job that it blocks.

Both the *priority-inheritance protocol* (PIP) and the *priority-ceiling protocol* (PCP) [54] satisfy these constraints. Under *priority inheritance*, a job that blocks a higher priority job "inherits" that job's priority until the lock is released. Unfortunately, the PIP does not avoid deadlock in the presence of nested critical sections.[5] The PCP is based on a more static notion of priority inheritance in which a task's priority is boosted as soon as it obtains any lock that *may* block a higher priority task. As a result, the PCP does avoid deadlock.

To simplify these example scenarios, we assume that all periodic and sporadic tasks have relative deadlines equal to their periods.

4.2 Deriving a Schedulability Condition

We next derive a sufficient schedulability condition for a set of component tasks. To accomplish this, we present a framework for analysis based on uniprocessor *demand-based* analysis (DBA). When discussing DBA, we use the term *request* to refer to any request for processor time issued by a task (*i.e.*, either a job or a subtask depending on the scenario).

DBA on a dedicated uniprocessor. We begin by illustrating the ideas underlying DBA with an example. Figure 4.4 shows a schedule in which four independent, synchronous periodic tasks are scheduled on a dedicated uniprocessor using a fully preemptive EDF policy. As shown, a timing violation (*i.e.*, deadline miss) occurs at time 24.

scheduling policy is that used within the supertask.

[5]Actually, the PIP does avoid deadlock when tasks are constrained to request locks in a specific order.

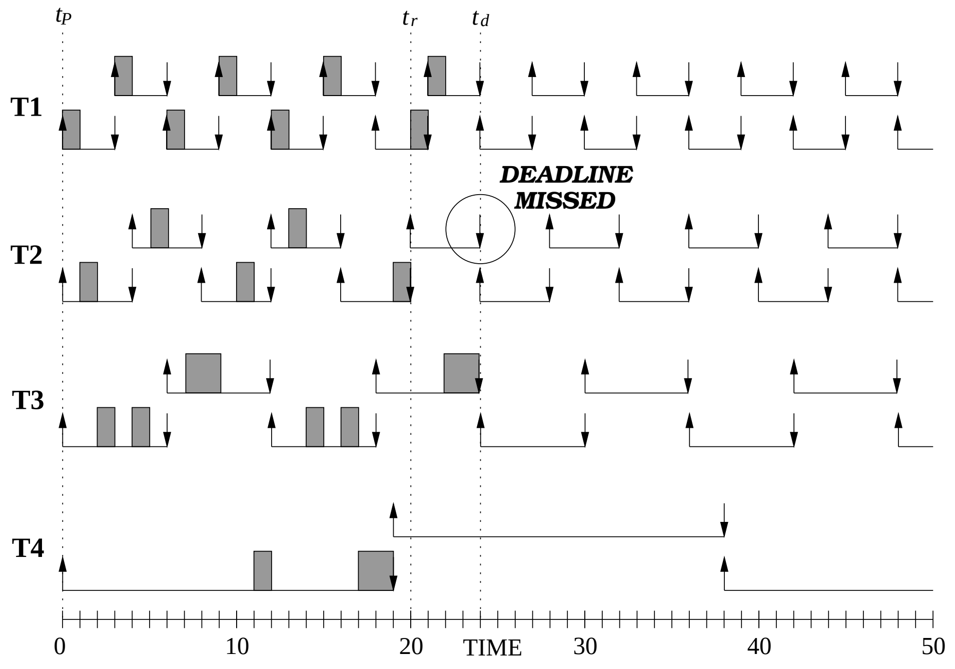


Figure 4.4: Sample DBA schedule consisting of four synchronous periodic tasks: $\mathbf{T1} = \mathbf{P}(0,1,3,3)$, $\mathbf{T2} = \mathbf{P}(0,1,4,4)$, $\mathbf{T3} = \mathbf{P}(0,2,6,6)$, and $\mathbf{T4} = \mathbf{P}(0,3,19,19)$. The task set is scheduled using a fully preemptive EDF policy and experiences its first violation at time 24, as shown.

The goal of DBA is to characterize the state of the system leading up to the timing violation. Let R denote the request (job) that experiences the violation. (We will refer to R repeatedly throughout the chapter.) $R = \mathbf{T2}(6)$ in Figure 4.4. Let t_d and t_r denote the times at which the violation occurs and at which R was released, respectively. In Figure 4.4, $t_d = R.d = 24$ and $t_r = R.a = 20$.

Now, consider the state of the system over $[t_r, t_d)$. First, due to the existence of R and the EDF-based prioritization of tasks, no idling occurred within $[t_r, t_d)$ and every job that executed in the interval had a deadline at or before t_d. Hence, Property BP, shown below, held throughout $[t_r, t_d)$.

Busy Period (BP): A ready or executing job exists that has priority at least that of R.

Unfortunately, characterizing the state of the system at t_r is difficult. Under DBA, this problem is addressed by simply moving the start of the interval under observation to an earlier point at which the state is known. Specifically, let t_P denote the earliest time within the interval $[0, t_r]$ for which a given property P holds throughout $[t_P, t_r)$.[6] In the DBA example, Property BP is P. It is often necessary to modify P to account for events that may occur at runtime, such as task suspensions. Indeed, we necessarily consider a different choice of P below to account for the scheduling of the supertask. Since $t_P = t_r$ trivially satisfies the requirement when no other points do, some valid choice of t_P always exists. In Figure 4.4, $t_P = 0$. Based on the selection criteria for t_P, $[t_P, t_d)$ also satisfies the properties stated above for $[t_r, t_d)$. In addition, the fact that Property BP does not hold immediately prior to t_P implies that all jobs executing in $[t_P, t_d)$ are released at or after time t_P.

It follows from the above observations that all processor time in $[t_P, t_d)$ was consumed by job with releases at or after t_P and with deadlines at or before t_d. The total processor time required to complete all such jobs is called the *demand* and is denoted $\mathit{demand}(\tau, t_P, t_d)$. The available processor time in $[t_P, t_d)$ is then called the *supply* and is denoted $\mathit{supply}(\tau, t_P, t_d)$. On a dedicated uniprocessor, the supply is simply determined by the interval length, *i.e.*, $\mathit{supply}(\tau, t_P, t_d) = t_d - t_P$. The fact that a deadline miss occurs implies the following relationship (and a necessary condition for a deadline miss).

$$\mathit{demand}(\tau, t_P, t_d) > \mathit{supply}(\tau, t_P, t_d)$$

In Figure 4.4, $\mathit{demand}(\tau, t_P, t_d) = 25$ and $\mathit{supply}(\tau, t_P, t_d) = 24$.

[6]Typically, P is selected to be a property that holds over $[t_r, t_d)$. However, this approach does not necessarily produce the best results. Unfortunately, effective selection of P requires insights into the scenario under consideration and some degree of intuition, *i.e.*, we are aware of no systematic method for selecting P.

Formalizing the condition. The goal of DBA is to derive a necessary condition for a timing violation to occur, *i.e.*, to find a condition C that satisfies

$$\textit{a violation occurs} \Rightarrow C.$$

Restating the relationship between supply and demand more formally produces the following definition for C:

$$C \stackrel{\text{def}}{=} (\exists t_P, t_d : t_d - t_P \geq \min\{\ T.p \mid \tau\ \} : \textit{demand}(\tau, t_P, t_d) > \textit{supply}(\tau, t_P, t_d)).$$

The constraint $t_d - t_P \geq \min\{\ T.p \mid \tau\ \}$ follows from the fact that $[t_P, t_d)$, defined above, always contains $[t_r, t_d)$. (The inequality $t_d - t_r \geq \min\{\ T.p \mid \tau\ \}$ follows from the fact that $t_d - t_r = R.d$ and each task's relative deadline equals its period.) Taking the contrapositive of *a violation occurs* $\Rightarrow C$ produces the sufficient schedulability condition

$$\neg C \Rightarrow \textit{no violation occurs}.$$

Hence, no deadline misses can occur if the following condition is satisfied:

$$(\forall t_P, t_d : t_d - t_P \geq \min\{\ T.p \mid \tau\ \} : \textit{demand}(\tau, t_P, t_d) \leq \textit{supply}(\tau, t_P, t_d)). \qquad (4.3)$$

Generalizing DBA for supertasking. Applying DBA to supertasks follows the same steps given above. However, several modifications are necessary. First, timing violations under supertasking are tardiness violations rather than deadline misses. Second, Property BP cannot be applied effectively because it does not account for the scheduling of the supertask.

For the five example scenarios considered here, we consider Property SBP, shown below.

Supertask Busy Period (SBP): Whenever $\mathcal{S}$ is scheduled, an eligible or executing request of a component task exists that has normal priority at least that of R.

Notice that the only significant difference between Property SBP and Property BP is that attention is restricted to times at which $\mathcal{S}$ is executing. To the best of our knowledge, all properties proposed for DBA on a dedicated uniprocessor can be similarly adapted to supertasking with relative ease. Third, since component tasks can only execute when $\mathcal{S}$ is scheduled, it follows that $\mathit{supply}(\mathcal{S}, t_P, t_d) = \mathit{received}(\mathcal{S}, t_P, t_d)$.

Bounding demand. The most difficult part of DBA is deriving bounds on the demand generated by the tasks. In the DBA example given in Figure 4.4, all demand was *mandatory* in the sense that the jobs contributing to the demand existed only within the interval $[t_P, t_d)$. Hence, successful scheduling required that all such jobs, called *mandatory* jobs, be executed within $[t_P, t_d)$. For instance, no combination of scheduling policy and synchronization protocols can successfully schedule the task set without executing job $\mathbf{T3}(3)$ within $[0, 24)$.

Real task systems are typically more complex, which results in additional demand being introduced into each interval. We refer to such demand as *circumstantial demand*. Circumstantial demand is generated by dependencies between requests (*e.g.*, precedence constraints, resource sharing, *etc.*) and the use of suboptimal policies (*e.g.*, using RM on a uniprocessor, allowing non-preemptable execution, *etc.*).

The circumstantial demand stemming from the latter source is the result of how policy choices can the impact the difficulty of scheduling. For instance, if a fully preemptive RM policy is used on a dedicated uniprocessor, then a job J with a later deadline than R may be prioritized over R. Hence, the choice of scheduling policy imposes the unnecessary constraint that J be executed before R (if both requests are pending). When the goal of scheduling is to meet deadlines, such a constraint is simply illogical. Indeed, this policy choice actually makes scheduling more difficult due to the fact that more processor time is needed to guarantee that R's deadline is met (as compared to that required when an optimal prioritization is used). Since it is not necessary to schedule J in $[t_P, t_d)$, the demand contributed by J is circumstantial demand. Similarly, a job J can contribute to the circumstantial demand by holding a lock that is needed by a mandatory job or by initiating a non-preemptive code segment immediately prior to t_P.

To distinguish between these two forms of demand, we decompose $\mathit{demand}(\mathcal{S}, t_P, t_d)$ into mandatory demand, denoted $\mathit{demand}_M(\mathcal{S}, t_P, t_d)$, and circumstantial demand, denoted $\mathit{demand}_C(\mathcal{S}, t_P, t_d)$. Hence, $\mathit{demand}(\mathcal{S}, t_P, t_d) = \mathit{demand}_M(\mathcal{S}, t_P, t_d) + \mathit{demand}_C(\mathcal{S}, t_P, t_d)$. Since requests in the QB-EPDF, QB-EDF, and FP-EDF scenarios are prioritized by an EDF rule and these scenarios consider only independent tasks that never suspend, attention can be restricted to mandatory demand. However, circumstantial demand does exist and must be accounted for in the FP-EDF-NP and FP-EDF-CS scenarios due to the inclusion of non-preemptable code segments and critical sections, respectively.

Schedulability condition. We now state the schedulability condition that will be used to drive the reweighting process, which is discussed later in Section 4.3. The condition, which generalizes (4.3), is shown below.

$$(\forall t, L : L \geq L_0 : \mathit{demand}_M(\mathcal{S}, t, t+L) + \mathit{demand}_C(\mathcal{S}, t, t+L) \leq \mathit{supply}(\mathcal{S}, t, t+L)). \quad (4.4)$$

(4.4) is derived from (4.3) by replacing t_P and t_d with t and $t + L$, respectively. (The purpose of this substitution is to make the interval length, L, an explicit parameter.) L_0 is used to denote the shortest interval over which a request can be released and experience a violation. As explained above, $[t, t+L)$ must always contain these two points. All terms in (4.4) are scenario-specific. In the subsections that follow, we derive bounds for these terms for each of the five scenarios.

Additional notation. For the FP-EDF-CS scenario, let $\Gamma(T) = \left\{ T^{[i]}.R \;\middle|\; 1 \leq i \leq T.k \right\}$. Informally, $\Gamma(T)$ is the set of locks requested by task T.

4.2.1 Defining L_0

Deriving L_0 is straightforward as it depends only on the shortest interval that can contain a release and the associated effective deadline.

Scenario 1 (QB-EPDF). This scenario focuses on subtasks with Pfair windows. Hence, by the $n = 1$ case of Lemma 3.3 on page 70, L_0 can be defined as shown below.

$$L_0 = \min\left\{ \left\lceil \frac{1}{U.w} \right\rceil + U.c \;\middle|\; U \in \mathcal{S} \right\} \tag{4.5}$$

Scenario 2 (QB-EDF). This scenario focuses on subtasks with windows defined by pseudo-jobs. Recall that each pseudo-job of a task U consists of $U.J$ cycles, each of which has length $\mathcal{P}(U)$. Hence, L_0 is defined as shown below.

$$L_0 = \min\{ U.J \cdot \mathcal{P}(U) + U.c \mid U \in \mathcal{S} \} \tag{4.6}$$

Scenarios 3–5 (FP-EDF, FP-EDF-NP, FP-EDF-CS). Under these scenarios, each request is a job. Since periods are assumed to equal deadlines, L_0 is defined as shown below.

$$L_0 = \min \{ \ U.p + U.c \mid U \in \mathcal{S} \ \} . \tag{4.7}$$

4.2.2 Defining *supply*$(\mathcal{S}, t, t + L)$

We now derive a bound on the minimum amount of processor time guaranteed to a task over any interval of length L. Since a supertask behaves like any other Pfair task, *supply*$(\mathcal{S}, t, t + L)$ can be defined using such a bound. We begin by proving the following theorem, which bounds the allocation granted to any Pfair task over the interval $[0, t)$.

Theorem 4.1 *The amount of processor time received by a task T that does not delay any subtask releases over the interval $[0, t)$, where t is an integer, under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, is bounded as shown below.*

$$(\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q \leq \mathit{received}(T, 0, t) \leq (\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q$$

Proof. The proof consists of two parts. First, we restrict attention to the impact of relaxed lag constraints. After addressing this impact, we then consider the impact of ϵ_r and ϵ_d.

Combining (2.1) on page 35, (2.2) on page 36, and (3.7) on page 65 produces

$$-\beta_- \cdot Q < Q \cdot T.w \cdot t - \mathit{received}(T, 0, t) < \beta_+ \cdot Q,$$

which must hold for all values of t. Dividing all terms in the previous inequality by Q and rearranging terms to isolate $\frac{1}{Q} \cdot \mathit{received}(T, 0, t)$ yields the following inequality.

$$T.w \cdot t - \beta_+ < \frac{1}{Q} \cdot \mathit{received}(T, 0, t) < T.w \cdot t + \beta_-$$

By the statement of the lemma, t is an integer, and hence $Q \,|\, \mathit{received}(T, 0, t)$ holds since processor time is allocated in units of Q. It follows that $\frac{1}{Q} \cdot \mathit{received}(T, 0, t)$ is an integer. When $\frac{a}{b}$ is an integer, then $x < \frac{a}{b} < y \Leftrightarrow \lfloor x \rfloor + 1 \leq \frac{a}{b} \leq \lceil y \rceil - 1 \Leftrightarrow (\lfloor x \rfloor + 1) \cdot b \leq a \leq (\lceil y \rceil - 1) \cdot b$. This property implies that the previous inequality can be rewritten as

$$(\lfloor T.w \cdot t - \beta_+ \rfloor + 1) \cdot Q \leq \mathit{received}(T, 0, t) \leq (\lceil T.w \cdot t + \beta_- \rceil - 1) \cdot Q.$$

Intuitively, $\lfloor T.w \cdot t - \beta_+ \rfloor + 1$ (respectively, $\lceil T.w \cdot t + \beta_- \rceil - 1$) is the number of subtasks with relaxed deadlines at or before (respectively, with relaxed releases before) time t.

We now consider the impact of ϵ_r and ϵ_d on the above allocation bounds. A subtask with a relaxed deadline at time t_d may not complete until its extended deadline at time $t_d + \epsilon_d$. Hence, the lower bound may be smaller than that given above. For instance, consider Figure 4.5, which shows the relaxed and extended windows for a task $T = \mathbf{PF}\left(\frac{3}{10}\right)$ when $\beta_- = \beta_+ = 1$, $\epsilon_r = 1$, and $\epsilon_d = 2$. The lower bound of $\mathit{received}(T, 0, 10)$ with respect to relaxed windows includes T_3 since its relaxed deadline occurs at time 10. However, when considering extended windows, T_3 cannot be counted in the lower bound since it may be executed after time 10. In general, the number of subtasks with extended deadlines at or before time t equals the number with relaxed deadlines at or before time $t - \epsilon_d$. By the lower bound derived above for relaxed windows, $\lfloor T.w \cdot t - \beta_+ \rfloor + 1$ subtasks have relaxed deadlines

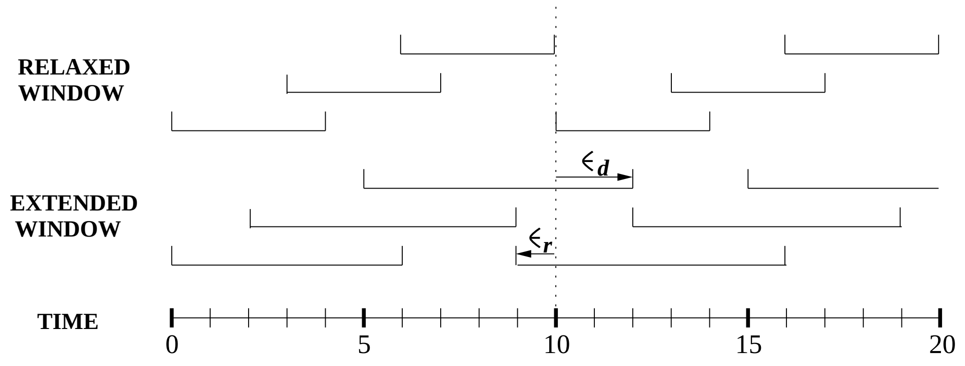


Figure 4.5: The figure shows the task $\mathbf{PF}\left(\frac{3}{10}\right)$ when $\beta_- = \beta_+ = 1$ (*i.e.*, with Pfair lag bounds). The window layouts show the impact of ϵ_r and ϵ_d.

at or before time t. It follows that the lower bound with respect to extended windows is given by $(\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q$.

Similarly, subtasks with relaxed releases before $t + \epsilon_r$ may be scheduled before t when using extended windows. Hence, the upper bound of *received*$(T, 0, t)$ may be larger than that given above. Again, consider *received*$(T, 0, 10)$ in Figure 4.5. T_4 has a relaxed release at time 10. Hence, it must be excluded from the upper bound when considering relaxed windows. However, when using extended windows, the window release occurs at time 9. In general, the number of extended windows with releases at or after time t equals the number of relaxed windows with releases at or after time $t + \epsilon_r$. By the upper bound derived above for relaxed windows, $\lceil T.w \cdot t + \beta_- \rceil - 1$ subtasks have relaxed releases at or after time t. It follows that the upper bound with respect to extended windows is $(\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q$. $\square$

The next theorem extends the above result to an arbitrary interval $[t, t + L)$.

Theorem 4.2 *The amount of processor time received by a task T that does not delay any subtask releases over the interval $[t, t + L)$, where t and L are integers, under scheduling* *characterized by* β_-, β_+, ϵ_r, *and* ϵ_d, *is bounded as shown below.*

$$(\lfloor T.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q \leq \mathit{received}(T, t, t + L) \leq (\lceil T.w \cdot (L + \epsilon) + \beta \rceil - 1) \cdot Q$$

Proof. Due to the absence of IS delays over the interval $[t, t + L)$, it is sufficient to consider the allocation that is guaranteed to a task that never experiences IS delays. Since t and L are integers, Theorem 4.1 can be applied to derive the lower bound, as shown below.

$\mathit{received}(T, t, t + L)$

$= \mathit{received}(T, 0, t + L) - \mathit{received}(T, 0, t)$

, by definition

$\geq (\lfloor T.w \cdot ((t + L) - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q - (\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q$

, by Theorem 4.1

$= \lfloor (T.w \cdot (L - \epsilon_d - \epsilon_r) - \beta_+ - \beta_-) + (T.w \cdot (t + \epsilon_r) + \beta_-) \rfloor \cdot Q$

$- \lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil \cdot Q + 2 \cdot Q$

, by rewriting

$\geq \lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil \cdot Q + (\lfloor T.w \cdot (L - \epsilon_d - \epsilon_r) - \beta_+ - \beta_- \rfloor - 1) \cdot Q$

$- \lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil \cdot Q + 2 \cdot Q$

, $\lfloor a + b \rfloor \geq \lceil a \rceil + \lfloor b \rfloor - 1$

$= (\lfloor T.w \cdot (L - \epsilon_d - \epsilon_r) - \beta_+ - \beta_- \rfloor + 1) \cdot Q$

, simplification

$= (\lfloor T.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q$

, by definition of ϵ and β

Similarly, the derivation given below establishes the upper bound.

$$\begin{aligned}
&\mathit{received}(T, t, t+L) \\
&\quad = \mathit{received}(T, 0, t+L) - \mathit{received}(T, 0, t) \\
&\qquad\qquad\qquad\qquad , \text{ by definition} \\
&\quad \leq (\lceil T.w \cdot ((t+L) + \epsilon_r) + \beta_- \rceil - 1) \cdot Q - (\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by Theorem 4.1} \\
&\quad = \lceil (T.w \cdot (L + \epsilon_d + \epsilon_r) + \beta_+ + \beta_-) + (T.w \cdot (t - \epsilon_d) - \beta_+) \rceil \cdot Q \\
&\qquad - \lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor \cdot Q - 2 \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by rewriting} \\
&\quad \leq (\lceil T.w \cdot (L + \epsilon_d + \epsilon_r) + \beta_+ + \beta_- \rceil + 1) \cdot Q + \lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor \cdot Q \\
&\qquad - \lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor \cdot Q - 2 \cdot Q \\
&\qquad\qquad\qquad\qquad , \lceil a + b \rceil \leq \lceil a \rceil + \lfloor b \rfloor + 1 \\
&\quad = (\lceil T.w \cdot (L + \epsilon_d + \epsilon_r) + \beta_+ + \beta_- \rceil - 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ simplification} \\
&\quad = (\lceil T.w \cdot (L + \epsilon) + \beta \rceil - 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by definition of } \epsilon \text{ and } \beta
\end{aligned}$$

The combination of these bounds establishes the theorem. □

Using the above results, we now bound *supply*$(T, t, t+L)$ in a series of corollaries given below. Each corollary corresponds to a case that can arise under supertasking. Specifically, the interval under inspection can align to slot boundaries on both ends (Corollary 4.1), on only one end (Corollary 4.2), or on neither end (Corollary 4.3). Only Corollaries 4.1 and 4.3 apply to the examples considered in this chapter. The benefit of using Corollaries 4.1 and 4.2 instead of Corollary 4.3 is that the estimate of the supply is higher, which results in lower reweighting overhead. Consequently, imposing artificial restrictions on the component tasks, such as requiring all releases to occur on slot boundaries, can reduce reweighting overhead. (We consider the use of such a restriction when performing experiments later.) Since we assume no such restrictions in the example scenarios, Scenarios 3–5 (FP-EDF, FP-EDF-NP, FP-EDF-CS) must use Corollary 4.3. However, under Scenarios 1 (QB-EPDF) and 2 (QB-EDF), intervals align to slot boundaries on both ends due to quantum-based scheduling. Hence, Corollary 4.1 applies in these cases.

Corollary 4.1 *The supply of a supertask $\mathcal{S}$ that does not delay any subtask releases over the interval $[t, t+L)$, where t and L are integers, while executing under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, satisfies*

$$\mathit{supply}(\mathcal{S}, t, t+L) \geq (\lfloor \mathcal{S}.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q.$$

Proof. Since $\mathit{supply}(\mathcal{S}, t, t+L) = \mathit{received}(\mathcal{S}, t, t+L)$, the corollary follows trivially from Theorem 4.2. □

Corollary 4.2 *The supply of a supertask $\mathcal{S}$ that does not delay any subtask releases over the interval $[t, t+L)$, where either t or $t+L$ is an integer, while executing under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, satisfies*

$$\mathit{supply}(\mathcal{S}, t, t+L) \geq (\lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - \epsilon) - \beta \rfloor + 1) \cdot Q.$$

Proof. If either t or $t+L$ is an integer, then the subinterval $[\lceil t \rceil, \lfloor t+L \rfloor)$ has length $\lfloor L \rfloor$ and aligns to slot boundaries on both ends. The corollary follows from Theorem 4.2. □

Corollary 4.3 *The supply of a supertask $\mathcal{S}$ that does not delay any subtask releases over any interval $[t, t+L)$, while executing under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, satisfies*

$$supply(\mathcal{S}, t, t+L) \geq (\lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta \rfloor + 1) \cdot Q.$$

Proof. The subinterval $[\lceil t \rceil, \lfloor t+L \rfloor)$ always has length at least $\lfloor L \rfloor - 1$ and aligns to slot boundaries on both ends. Hence, the corollary follows from Theorem 4.2. $\square$

4.2.3 Defining $demand_M(\mathcal{S}, t, t+L)$

In this subsection, we derive bounds on the mandatory demand of a component task set. One advantage to separating demand into its mandatory and circumstantial components is that bounds on mandatory demand depend only the timing constraint (*e.g.*, job deadline, pseudo-job deadlines, subtask deadlines, *etc.*) used. Hence, bounds are universal across scheduling policies. Our five example scenarios require consideration of three cases: subtask demand (QB-EPDF), pseudo-job demand (QB-EDF), and job demand (FP-EDF, FP-EDF-NP, FP-EDF-CS). These cases are addressed by Corollaries 4.4, 4.5, and 4.6, respectively, which are given below.

Scenario 1 (QB-EPDF). Subtask demand is bounded by the following theorem and corollary.

Theorem 4.3 *When scheduling subtasks of tasks described by the Pfair task model and its variants, the mandatory demand generated by a task T, over the interval $[t, t+L)$, where t*

and L are integers, is upper-bounded by

$$demand_M(T, t, t+L) \leq \lfloor T.w \cdot (L - T.c) \rfloor \cdot Q$$

when T satisfies $T.c \leq L$. When $T.c > L$, T generates no mandatory demand.

Proof. It is sufficient to bound the maximum demand generated by a Pfair task over any interval of length L. The $T.c > L$ case is trivial since any subtask released by T within $[t, t+L)$ cannot be required to complete by the end of the interval. Hence, T generates no mandatory demand.

When $T.c \leq L$, the mandatory demand of T over $[t, t+L)$ can be computed as the difference between T's minimum allocation at $t+L$ less its maximum allocation at t. Applying Theorem 4.1 with $\beta_+ = \beta_- = 1$, $\epsilon_r = 0$, and $\epsilon_d = T.c$ produces the following formula:

$$demand_M(T, t, t+L) = (\lfloor T.w \cdot (t + L - T.c) \rfloor - \lceil T.w \cdot t \rceil) \cdot Q.$$

Rewriting the first term yields the following formula.

$$demand_M(T, t, t+L) = (\lfloor T.w \cdot t + T.w \cdot (L - T.c) \rfloor - \lceil T.w \cdot t \rceil) \cdot Q$$

Since $\lfloor a + b \rfloor \leq \lceil a \rceil + \lfloor b \rfloor$, it follows that

$$demand_M(T, t, t+L) \leq (\lfloor T.w \cdot (L - T.c) \rfloor + \lceil T.w \cdot t \rceil - \lceil T.w \cdot t \rceil) \cdot Q.$$

Simplifying this bound establishes the theorem. $\square$

Corollary 4.4 *When scheduling subtasks of tasks described by the Pfair task model and its variants, the following formula gives a series of progressively looser upper bounds on the total mandatory demand generated by the component task set of the supertask* $\mathcal{S}$ *over the interval* $[t, t+L)$*, where* t *and* L *are integers, and* $\mathcal{S}_L = \{\ T \mid T \in \mathcal{S} \wedge T.c \leq L\ \}$.

$$\begin{aligned} demand_M(\mathcal{S}, t, t+L) &= \sum_{T \in \mathcal{S}} demand_M(T, t, t+L) \\ &\leq Q \sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot (L - T.c) \rfloor \\ &\leq Q \left\lfloor \sum_{T \in \mathcal{S}_L} (T.w \cdot (L - T.c)) \right\rfloor \\ &\leq Q \sum_{T \in \mathcal{S}_L} (T.w \cdot (L - T.c)) \end{aligned}$$

Proof. The first inequality follows trivially from Theorem 4.3. The second and third inequalities follow from the properties $\lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a + b \rfloor$ and $\lfloor a \rfloor \leq a$, respectively. □

Scenario 2 (QB-EDF). The next theorem and corollary bound mandatory demand generated by subtasks with windows defined by pseudo-jobs.

Theorem 4.4 *When scheduling subtasks with parameters defined by pseudo-jobs, the mandatory demand generated by a task* T *over the interval* $[t, t+L)$*, where* t *and* L *are integers, is upper-bounded by*

$$demand_M(T, t, t+L) \leq \left\lfloor \frac{L - T.c}{T.J \cdot \mathcal{P}(T)} \right\rfloor \cdot T.J \cdot \mathcal{E}(T) \cdot Q$$

when T *satisfies* $T.c \leq L$*. When* $T.c > L$*,* T *generates no mandatory demand.*

Proof. When a job spans $T.J$ subtask cycles, it follows that the per-job execution requirement and period of T are $T.J \cdot \mathcal{E}(T) \cdot Q$ and $T.J \cdot \mathcal{P}(T)$, respectively. (Recall that $\mathcal{E}(T)$ and $\mathcal{P}(T)$ are the per-cycle execution requirement in quanta and cycle period of T, respectively.) Furthermore, an interval of length L, where $L \geq T.c$, contains at most $\left\lfloor \frac{L - T.c}{T.J \cdot \mathcal{P}(T)} \right\rfloor$ jobs that are released and must complete their execution within the interval. Multiplying this count by the per-job execution requirement produces $\left\lfloor \frac{L - T.c}{T.J \cdot \mathcal{P}(T)} \right\rfloor \cdot T.J \cdot \mathcal{E}(T) \cdot Q$, which establishes the theorem. $\square$

Corollary 4.5 *When scheduling subtasks with parameters defined by pseudo-jobs, the following formula gives a series of progressively looser upper bounds on the total mandatory demand generated by the component task set of the supertask $\mathcal{S}$ over the interval $[t, t+L)$, where t and L are integers, and $\mathcal{S}_L = \{\ T \mid T \in \mathcal{S} \wedge T.c \leq L\ \}$.*

$$\begin{aligned}
\mathit{demand}_M(\mathcal{S}, t, t+L) &= \sum_{T \in \mathcal{S}} \mathit{demand}_M(T, t, t+L) \\
&\leq Q \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L - T.c}{T.J \cdot \mathcal{P}(T)} \right\rfloor \cdot T.J \cdot \mathcal{E}(T) \right) \\
&\leq Q \sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot (L - T.c) \rfloor \\
&\leq Q \left\lfloor \sum_{T \in \mathcal{S}_L} (T.w \cdot (L - T.c)) \right\rfloor \\
&\leq Q \sum_{T \in \mathcal{S}_L} (T.w \cdot (L - T.c))
\end{aligned}$$

Proof. All inequalities are derived as in the proof of Corollary 4.4, except for the first two. The first inequality follows from Theorem 4.4. The second then follows from the fact that $\lfloor a \rfloor n \leq \lfloor a \cdot n \rfloor$ whenever n is an integer, as is the case with $T.J \cdot \mathcal{E}(T)$. $\square$

Scenarios 3–5 (FP-EDF, FP-EDF-NP, FP-EDF-CS). Finally, the next theorem and corollary bound the mandatory demand produced by jobs.

Theorem 4.5 *When scheduling jobs of a periodic or sporadic task, the mandatory demand generated by a task T over the interval $[t, t+L)$ is upper-bounded by*

$$demand_M(T, t, t+L) \leq \left\lfloor \frac{L - T.c}{T.p} \right\rfloor \cdot T.e$$

when T satisfies $T.c \leq L$. When $T.c > L$, T generates no mandatory demand.

Proof. The reasoning here is identical to that in the proof of Theorem 4.4. □

Corollary 4.6 *When scheduling jobs of a periodic or sporadic task, the following formula gives a series of progressively looser upper bounds on the total mandatory demand generated by the component task set of the supertask $\mathcal{S}$ over the interval $[t, t+L)$, where $\mathcal{S}_L = \{\ T \mid T \in \mathcal{S} \wedge T.c \leq L\ \}$.*

$$\begin{aligned} demand_M(\mathcal{S}, t, t+L) &= \sum_{T \in \mathcal{S}} demand_M(T, t, t+L) \\ &\leq \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L - T.c}{T.p} \right\rfloor \cdot T.e \right) \\ &\leq \sum_{T \in \mathcal{S}_L} (T.u \cdot (L - T.c)) \end{aligned}$$

Proof. The first inequality follows from Theorem 4.5. The second follows from $\lfloor a \rfloor \leq a$. □

4.2.4 Defining $demand_C(\mathcal{S}, t, t+L)$

Unfortunately, bounding circumstantial demand requires unique reasoning for each scenario. (Indeed, this is why we use the term "circumstantial.") We now bound this demand for each of the five example scenarios. All analysis presented in this section assumes the use of Property SBP on page 110. As before, let $[t, t+L)$ denote the interval under consideration.

Scenarios 1–3 (QB-EPDF, QB-EDF, FP-EDF). In Scenarios 1–3, this step is trivial due to the fact that all tasks are independent and never suspend. Because requests are processed in deadline order, the mandatory demand is always serviced before all other requests. Hence, no circumstantial demand exists.

Scenario 4 (FP-EDF-NP). Under Scenario 4, a job of some task U with priority lower than the job experiencing the violation (*i.e.*, R) can begin nonpreemptable execution immediately before t and, consequently, avoid preemption at t. By Property SBP, such a job, if one indeed exists, does not need to complete until after $t + L$. Since this job must have been released and executed prior to t to initiate its nonpreemptable execution, it follows that $U.p + U.c > L$ holds. Furthermore, at most one such job can execute within $[t, t + L)$ since two component tasks cannot be executing nonpreemptably at the same time. Hence, circumstantial demand under Scenario 4 is upper bounded as shown below.

$$demand_C(\mathcal{S}, t, t + L) \leq \max\{\ U.v \mid U \in \mathcal{S} \wedge U.p + U.c > L\ \} \tag{4.8}$$

Observe that $demand_C(\mathcal{S}, t, t + L) = 0$ for all $L \geq \max\{\ U.p + U.c \mid U \in \mathcal{S}\ \}$. As the next example also demonstrates, circumstantial demand tends to vanish with increasing L. We refer to such terms as *transient* demand.

Scenario 5 (FP-EDF-CS). Consider a job K that has priority lower than R. If K holds a lock at time t that is required by some mandatory job J, then J will block on that lock at some point within $[t, t + L)$. Under the locking-protocol assumptions stated earlier on page 105, the priority of K must be increased to at least the level of J. Due to this, K will be allowed to complete its critical section at some point within $[t, t + L)$ (so that J can obtain the lock and make progress). As a result, K's critical section contributes to the circumstantial demand. (Since K's priority returns to its normal level when the lock is relinquished, only the critical section contributes to the demand.)

We now derive a bound on the amount of circumstantial demand generated by all such critical sections in $[t, t+L)$.To simplify the discussion, we use the term *obstacle* to refer to such a critical section of a low-priority job. (This term is simply a shorthand expression.) For an obstacle to be introduced, a low-priority job must hold a lock at t that is needed by a mandatory job within $[t, t+L)$. Since two tasks cannot hold the same lock at time t, it follows that each obstacle must be associated with a unique lock. In addition, since we do not consider nested critical sections, each obstacle must also belong to a unique task. (This follows from the fact that each task can be holding at most one lock at t.)

Unfortunately, identifying the worst-case combination of obstacles is difficult. Indeed, variants of this problem are known to be NP hard [54]. Since bounding worst-case blocking is not the focus of this chapter, we simply use a loose bound here to illustrate the idea. In expressing this bound, the following notation is used:

- Let $\Gamma_L = \bigcup_{U.p+U.c \leq L} \Gamma(U)$.
- Let $\mu(\ell, L) = \max\left\{ U^{[i]}.e \;\middle|\; U \in \mathcal{S} \wedge U^{[i]} \in \mathcal{C}(\ell) \wedge U.p + U.c > L \right\}$.

Informally, Γ_L is the set of locks that may be requested by mandatory jobs over the interval $[t, t+L)$, while $\mu(\ell, L)$ is an upper bound on the amount of circumstantial demand introduced into any interval of length L as a result of lock ℓ. A loose upper bound on the total circumstantial demand is easily computed from these values, as shown below.

$$demand_C(\mathcal{S}, t, t+L) \leq \sum_{\ell \in \Gamma_L} \mu(\ell, L) \tag{4.9}$$

The above bound is loose because it only considers one of the two restrictions mentioned above, *i.e.*, each obstacle must require a unique lock. The bound in (4.9) does not prevent multiple obstacles from being associated with the same task.

4.2.5 Summary

The table shown below summarizes which of the results presented in the previous subsections define the component parts of (4.4) under each scenario.

Scenario	L_0	*supply*	$demand_M$	$demand_C$
1	(4.5)	Corollary 4.1	Corollary 4.4	*N/A*
2	(4.6)	Corollary 4.1	Corollary 4.5	*N/A*
3	(4.7)	Corollary 4.3	Corollary 4.6	*N/A*
4	(4.7)	Corollary 4.3	Corollary 4.6	(4.8)
5	(4.7)	Corollary 4.3	Corollary 4.6	(4.9)

4.3 Deriving a Reweighting Condition

In this section, we use the schedulability condition derived in the Section 4.2 to derive a reweighting condition. A reweighting condition consists of a set of *weight restrictions* (*i.e.*, lower bounds on the supertask's weight) such that satisfying all restrictions ensures the timeliness of component tasks. A reweighting condition takes the following abstract form:

$$\mathcal{S}.w \geq \max\{\ \Delta(\mathcal{S}, L) \mid L \in \mathcal{L}\ \}\,. \tag{4.10}$$

Condition (4.10) consists of two elements: the *reweighting function* $\Delta(\mathcal{S}, L)$ and the *testing set* $\mathcal{L}$. $\Delta(\mathcal{S}, L)$ is the minimum weight needed to ensure that no timing violation occurs over any interval of length L, *i.e.*, $\Delta(\mathcal{S}, L)$ is the weight restriction imposed by all intervals of length L. Let $\mathcal{S}.w_{\text{opt}}$ denote the smallest weight satisfying (4.10), as shown below.

$$\mathcal{S}.w_{\text{opt}} \overset{\text{def}}{=} \max\{\ \Delta(\mathcal{S}, L) \mid L \in \mathcal{L}\ \} \tag{4.11}$$

As in the previous section, we consider the derivation of each element in a separate subsection. Specifically, the three subsections of this section address the following issues:

1. deriving $\Delta(\mathcal{S}, L)$;
2. defining $\mathcal{L}$;
3. efficient generation of L values from $\mathcal{L}$.

To speed the presentation, only Scenarios 1 (QB-EPDF) and 4 (FP-EDF-NP) are considered in this section. Other scenarios can be similarly handled. Scenarios 1 and 4 are each considered because they represent quantum-based and fully preemptive supertasking, respectively. For fully preemptive supertasking, we consider the FP-EDF-NP scenario instead of the FP-EDF scenario to illustrate the handling of circumstantial demand.

4.3.1 Deriving $\Delta(\mathcal{S}, L)$

We begin by deriving the reweighting formula.

Scenario 1 (QB-EPDF). Filling in the terms of the inequality in (4.4) using Corollary 4.1 and the tightest bound provided by Corollary 4.4 yields the inequality shown below (after canceling the common Q term on both sides).

$$\sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot (L - T.c) \rfloor \leq \lfloor \mathcal{S}.w \cdot (L - \epsilon) - \beta \rfloor + 1$$

By the property $\lfloor x \rfloor \leq \lfloor y \rfloor \Leftrightarrow \lfloor x \rfloor \leq y$, the above inequality is equivalent to

$$\sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot (L - T.c) \rfloor \leq \mathcal{S}.w \cdot (L - \epsilon) - \beta + 1.$$

Rearranging to isolate $\mathcal{S}.w$ yields the following weight restriction (and definition of $\Delta(\mathcal{S}, L)$).

$$\mathcal{S}.w \geq \frac{\sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot (L - T.c) \rfloor + \beta - 1}{L - \epsilon} \overset{\text{def}}{=} \Delta(\mathcal{S}, L)\,, \tag{4.12}$$

where L_0 is required to satisfy

$$L_0 > \epsilon. \tag{4.13}$$

Constraint (4.13) ensures that $L - \epsilon > 0$ for all $L \geq L_0$. When this constraint does not hold, it is not possible to reweight the supertask using the technique presented here. By (4.5), the constraint given in (4.13) can also be stated as shown below.

$$\min \left\{ \left\lceil \frac{1}{U.w} \right\rceil + U.c \;\middle|\; U \in \mathcal{S} \right\} > \epsilon \tag{4.14}$$

Since periods are expected to be much larger than the slot size and ϵ is expected to be small, (4.14) should seldom, if ever, not hold.

Scenario 4 (FP-EDF-NP). Again, filling in the terms of (4.4) using Corollary 4.3, (4.8), and the tightest bound provided by Corollary 4.6 yields the inequality shown below.

$$\sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L - T.c}{T.p} \right\rfloor \cdot T.e \right) + \upsilon_L \leq (\lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta \rfloor + 1) \cdot Q$$

where

$$\upsilon_L \overset{\text{def}}{=} \max\{\ U.\upsilon \mid U \in \mathcal{S} \wedge U.p + U.c > L\ \}\,. \tag{4.15}$$

Rewriting this inequality yields the equivalent form shown below.

$$\sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L - T.c}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) + \frac{\upsilon_L}{Q} - 1 \leq \lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta \rfloor$$

This condition is satisfied if

$$\left\lceil \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L - T.c}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) + \frac{\upsilon_L}{Q} \right\rceil - 1 \leq \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta$$

is satisfied. Notice that this latter condition is slightly stronger. Rewriting this inequality to isolate $\mathcal{S}.w$ produces the following weight restriction (and definition of $\Delta(\mathcal{S}, L)$):

$$\mathcal{S}.w \geq \frac{\left\lceil \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L-T.c}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) + \frac{\upsilon_L}{Q} \right\rceil + \beta - 1}{\lfloor L \rfloor - 1 - \epsilon} \overset{\text{def}}{=} \Delta(\mathcal{S}, L)\,, \tag{4.16}$$

which requires

$$L_0 \geq \epsilon + 2. \tag{4.17}$$

Again, the above restriction ensures that $\lfloor L \rfloor - 1 - \epsilon > 0$ for all $L \geq L_0$. By (4.7), this constraint is equivalent to that given below.

$$\min\{\ U.p + U.c \mid U \in \mathcal{S}\ \} \geq \epsilon + 2. \tag{4.18}$$

As with (4.14), (4.18) should seldom, if ever, be violated.

The impact of the mandatory-demand bound. Both derivations given above are based on the tightest bounds on mandatory demand provided by Corollaries 4.4 and 4.6. Alternatively, one of the looser bounds provided by these corollaries could have been used. The benefit of using a looser bound is that it speeds the reweighting process. This is due to the fact that the running time of the reweighting algorithm (presented later in the chapter) is proportional to the reweighting overhead. Since looser mandatory-demand bounds overestimate demand, they produce more inflation. Consequently, reweighting requires less time.

Unfortunately, using looser bounds is a poor method for reducing the execution time of the reweighting algorithm because the impact of the bound on both the execution time and the reweighting overhead is difficult to predict. For this reason, we do *not* recommend the use of looser bounds or consider their use beyond this point. As an alternative, the reweighting algorithm has been equipped with several input parameters that allow manipulation of the speed-versus-accuracy trade-off in a controlled and predictable way.

4.3.2 Defining $\mathcal{L}$

As a starting point, consider letting $\mathcal{L} = \{\ L \mid L \geq L_0 \wedge L \in \mathcal{N}\ \}$ when using quantum-based supertasking and $\mathcal{L} = \{\ L \mid L \geq L_0\ \}$ when using fully preemptive supertasking. (In the first set definition, $\mathcal{N}$ denotes the set of natural numbers.) These definitions ensure the correctness of the reweighting condition, but include an unnecessarily high number of points in $\mathcal{L}$.

Scenario 1 (QB-EPDF). Consider the reweighting function given in (4.12) and a range $L \in [L_1, L_2)$ over which the numerator remains constant. Due to the L parameter in the denominator, the function's value can only decrease as L takes on larger values within $[L_1, L_2)$. Hence, a search for the maximum value requires only that the smallest L value from the testing set in $[L_1, L_2)$ need be checked. In (4.12), changes are caused by the argument to the floor operation; the k^{th} increase of T's term in the summation occurs at the following value of L.

$$L(k) = \min\{\ L \ \mid T.w \cdot (L - T.c) \geq k\ \} = \frac{k}{T.w} + T.c$$

However, $L(k)$ may not be in $\mathcal{N}$. Applying this additional restriction produces the set $\{\ \lceil L(k) \rceil \mid\ k \geq 1\ \}$, which can also be expressed as

$$\left\{\ \left\lceil \frac{k}{T.w} \right\rceil + T.c \ \middle|\ k \geq 1\ \right\}.$$

(Recall that $T.c$ is assumed to be a non-negative integer under quantum-based supertasking.) The definition of $\mathcal{L}$ for Scenario 1 is then the result of unioning these per-task testing sets, as shown below.

$$\mathcal{L} = \bigcup_{T \in \mathcal{S}} \left\{\ \left\lceil \frac{k}{T.w} \right\rceil + T.c \ \middle|\ k \geq 1\ \right\} \tag{4.19}$$

Scenario 4 (FP-EDF-NP). In Scenario 4, the process of defining $\mathcal{L}$ follows the same steps. Specifically, the numerator of the reweighting function given in (4.16) changes only due to changes in either a floor term of the summation or the v_L term.

First, consider the floor terms. As before, we compute the value of L corresponding to the k^{th} increase in the term's value, as shown below.

$$L(k) = \min\left\{ L \;\middle|\; \frac{L - T.c}{T.p} \geq k \right\} = k \cdot T.p + T.c.$$

Since L is not required to be an integer when using fully preemptive scheduling, the per-task sets $\{\ L(k) \mid k \geq 1\ \}$ are simply unioned to obtain the following definition of $\mathcal{L}$.

$$\mathcal{L} = \bigcup_{T \in \mathcal{S}} \{\ k \cdot T.p + T.c \mid k \geq 1\ \} \tag{4.20}$$

It remains to account for changes in the numerator caused by υ_L. Consider the definition of υ_L given in (4.15). It follows from the $U.p + U.c > L$ condition in the set definition that if υ_L differs from $\upsilon_{L-\varepsilon}$, for arbitrarily small ε, then $U.p + U.c = L$ for the task U that defines $\upsilon_{L-\varepsilon}$, *i.e.*, U is eliminated from consideration at L but is still being considered at $L - \epsilon$. Hence, the set of L values at which the υ_L term may change its value is given by

$$\{\ T.p + T.c \mid T \in \mathcal{S}\ \}.$$

Since this set is a subset of the previous definition of $\mathcal{L}$, it follows that the definition given in (4.20) is sufficient in this scenario. (This subset property also holds for the circumstantial demand bound derived for Scenario 5.)

4.3.3 Generating L Values from $\mathcal{L}$

In this subsection, we present an efficient algorithm for generating a monotonically increasing sequence of L values from $\mathcal{L}$ such that no values in $\mathcal{L}$ are skipped. This algorithm is used as a subroutine by our reweighting algorithm (presented later).

```
typedef nodetype
    record
        begin
            L: real;
            k: integer;
            f(n: integer): real function
        end

var
    H: min-ordered heap of nodetype ordered by L;
    node: nodetype

procedure InitGenerator($\mathcal{S}$)
1:  H := MakeHeap();
2:  for each $T \in \mathcal{S}$ do
3:      node := MakeNode(T);
4:      node.k := 1;
5:      node.L := node.f(node.k);
6:      Insert(H,node)
    od

procedure Generate() returns real
7:  L := Min(H).L;
8:  while Min(H).L = L do
9:      node := ExtractMin(H);
10:     node.k := node.k + 1;
11:     node.L := node.f(node.k);
12:     Insert(H,node)
    od;
13: return L
```

Figure 4.6: Algorithm for generating L values in sequence based from a definition of $\mathcal{L}$.

The generator pseudo-code is shown in Figure 4.6. The algorithm consists of two routines: `InitGenerator` and `Generate`. `InitGenerator` initializes the generator, while `Generate` generates the next L value in the sequence. A detailed description follows.

Data structures. The set of L values generated by a specific task T are represented by a *nodetype* record. The $f(n)$ function field defines the L-value generating function, *i.e.*, the function implied by either (4.19) or (4.20). For instance, under Scenario 4, $f(n) = n \cdot T.p + T.c$, as suggested by (4.20). The *value* field stores the next L value in T's sequence. The minimum of these candidate values is determined by storing the records in a min-ordered heap by *value*.

Detailed description. The generator is initialized by a call to `InitGenerator`. In line 1, the heap is created. In line 3–5, each task's record is created and fields are initialized; the record is then stored in the heap in line 6.

`Generate` retrieves the smallest L value in $\mathcal{L}$ that has not been previously retrieved. Line 7 identifies this value. Lines 8–12 then update the heap entry of each task that has the selected L value. The selected value is returned in line 13. For both routines, $O(|\mathcal{S}| \log |\mathcal{S}|)$ time complexity can be achieved by using a binomial heap.

4.4 Selecting a Safe Weight

The $\mathcal{L}$ definitions given above imply that reweighting may require an unbounded number of computations. In this section, we address this issue by presenting a technique for detecting when the process can safely terminate. The solution presented here not only detects termination, but also provides a means for *forcing* termination at the expense of a predictable amount of extra inflation.

Concept. Our approach is based on examining the behavior of a *bounding function* $\phi(\mathcal{S}, L)$ of $\Delta(\mathcal{S}, L)$. Specifically, $\phi(\mathcal{S}, L)$ must satisfy the following constraints:

Bounding Constraint (BC): $(\forall L : L \geq L_\phi : \phi(\mathcal{S}, L) \geq \Delta(\mathcal{S}, L))$.

Monotonicity Constraint (MC): $\phi(\mathcal{S}, L)$ is a monotonic function of L.

Property BC ensures that $\phi(\mathcal{S}, L)$ upper bounds $\Delta(\mathcal{S}, L)$ for all values of L at and after

L_ϕ, which is called the *activation point*. Hence, if this property holds and $L \geq L_\phi$, then $\mathcal{S}.w \geq \phi(\mathcal{S}, L) \Rightarrow \mathcal{S}.w \geq \Delta(\mathcal{S}, L)$. As explained below, L_ϕ allows $\phi(\mathcal{S}, L)$ to more tightly bound $\Delta(\mathcal{S}, L)$ by skipping over smaller L values at which transient demand exists. We postpone a discussion of Property MC until after the derivation of $\phi(\mathcal{S}, L)$ is presented.

4.4.1 Deriving $\phi(\mathcal{S}, L)$

$\phi(\mathcal{S}, L)$ and L_ϕ can typically be defined using the following rules of thumb:

Rule 1: Let L_ϕ be the maximum of the following values:

- L_0 (*i.e.*, the smallest L value considered when reweighting);
- $\max\{\ T.c \mid T \in \mathcal{S}\ \}$;
- the smallest L value such that no transient demand exists in intervals of length $L' \geq L$.

The second value ensures that $\mathcal{S}_L = \mathcal{S}$ holds for all $L \geq L_\phi$. (Recall that $\mathcal{S}_L$ is the set of component tasks of $\mathcal{S}$ with miss thresholds at most L.) The third value then ensures that transient demand in $\Delta(\mathcal{S}, L)$ can be ignored when applying Rule 2.

Rule 2: Derive $\phi(\mathcal{S}, L)$ from $\Delta(\mathcal{S}, L)$ by replacing all non-continuous terms in $\Delta(\mathcal{S}, L)$ with continuous upper bounds. For instance, x and $x + 1$ are continuous upper bounds of $\lfloor x \rfloor$ and $\lceil x \rceil$, respectively.

Scenario 1 (QB-EPDF). In Scenario 1, no circumstantial demand exists. Hence, transient demand never exists and can be ignored. Applying Rule 1 yields

$$L_\phi \stackrel{\text{def}}{=} \max(L_0, \max\{\ T.c \mid T \in \mathcal{S}\ \}). \tag{4.21}$$

Applying Rule 2 to (4.12) produces the following upper bound:

$$\frac{\sum_{T \in \mathcal{S}} T.w \cdot (L - T.c) + \beta - 1}{L - \epsilon}.$$

Reorganizing the terms produces the following equivalent form.

$$\sum_{T \in \mathcal{S}} T.w + \frac{\left(\sum_{T \in \mathcal{S}} T.w \right) \cdot \epsilon + \beta - 1 - \sum_{T \in \mathcal{S}} T.w \cdot T.c}{L - \epsilon}$$

Applying (4.1) produces the following equivalent form:

$$\mathcal{S}.I_Q + \frac{\mathcal{S}.I_Q \cdot \epsilon + \beta - 1 - \sum_{T \in \mathcal{S}} T.w \cdot T.c}{L - \epsilon}$$

This leads to the following definition:

$$\phi(\mathcal{S}, L) \overset{\text{def}}{=} \mathcal{S}.I_Q + \frac{\Psi(\mathcal{S})}{L - \epsilon}, \tag{4.22}$$

where

$$\Psi(\mathcal{S}) \overset{\text{def}}{=} \mathcal{S}.I_Q \cdot \epsilon + \beta - 1 - \sum_{T \in \mathcal{S}} T.w \cdot T.c. \tag{4.23}$$

By (4.13), the denominator of the second term in (4.22) is always positive. Hence, the behavior of $\phi(\mathcal{S}, L)$ as L ($\geq L_\phi$) increases is determined by the value of $\Psi(\mathcal{S})$. We refer to $\Psi(\mathcal{S})$ as the *characteristic function* of $\phi(\mathcal{S}, L)$. Specifically, $\phi(\mathcal{S}, L)$ is decreasing when $\Psi(\mathcal{S}) > 0$, constant when $\Psi(\mathcal{S}) = 0$, and increasing when $\Psi(\mathcal{S}) < 0$. Since $\Psi(\mathcal{S})$ is independent of L, its value can be pre-computed prior to reweighting.

Scenario 4 (FP-EDF-NP). By Rule 1 and (4.15), L_ϕ is defined as shown below.

$$L_\phi \overset{\text{def}}{=} \max(L_0, \max\{\ T.c \mid T \in \mathcal{S}\ \}, \max\{\ T.p + T.c \mid T \in \mathcal{S} \wedge T.\upsilon > 0\ \}) \tag{4.24}$$

By (4.15), $L = \max\{\ T.p + T.c \mid T \in \mathcal{S} \wedge T.\upsilon > 0\ \}$ is the smallest L value at which circumstantial demand no longer exists, *i.e.*, υ_L becomes 0.

Removing υ_L from (4.16) and applying Rule 2 produces the upper bound shown below.

$$\frac{\sum_{T\in\mathcal{S}} \left(\frac{T.u \cdot L - T.u \cdot T.c}{Q}\right) + \beta}{L - \epsilon - 2}$$

Reorganizing the terms produces the equivalent form shown below.

$$\frac{1}{Q} \cdot \sum_{T\in\mathcal{S}} T.u + \frac{\frac{1}{Q} \cdot \left(\sum_{T\in\mathcal{S}} T.u\right) \cdot (\epsilon + 2) + \beta - \frac{1}{Q} \cdot \sum_{T\in\mathcal{S}} T.u \cdot T.c}{L - \epsilon - 2}$$

Applying (4.2) produces the following equivalent form:

$$\mathcal{S}.I_P + \frac{\mathcal{S}.I_P \cdot (\epsilon + 2) + \beta - \frac{1}{Q} \cdot \sum_{T\in\mathcal{S}} T.u \cdot T.c}{L - \epsilon - 2}.$$

This leads to the following definition:

$$\phi(\mathcal{S}, L) \overset{\text{def}}{=} \mathcal{S}.I_P + \frac{\Psi(\mathcal{S})}{L - \epsilon - 2}, \tag{4.25}$$

where

$$\Psi(\mathcal{S}) \overset{\text{def}}{=} \mathcal{S}.I_P \cdot (\epsilon + 2) + \beta - \frac{1}{Q} \cdot \sum_{T\in\mathcal{S}} T.u \cdot T.c. \tag{4.26}$$

Again, the behavior of $\phi(\mathcal{S}, L)$ is determined by the characteristic function $\Psi(\mathcal{S})$.

4.4.2 Termination

We now explain how Property MC is used to detect and to force termination.

Decreasing monotonicity. The following theorem and corollary characterize how decreasing monotonicity allows the unbounded reweighting search space to be truncated (possibly at the cost of increased inflation) by considering a single value of $\phi(\mathcal{S}, L)$.

Theorem 4.6 *If $\phi(\mathcal{S}, L)$ is decreasing and $w \geq \phi(\mathcal{S}, L)$ for some $L \geq L_\phi$, then*

$$w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L' \geq L\ \}\,.$$

Proof. Since $\phi(\mathcal{S}, L)$ is a decreasing function, Property MC implies that if $w \geq \phi(\mathcal{S}, L)$ for some $L \geq L_\phi$, then $w \geq \phi(\mathcal{S}, L')$ for all $L' \geq L$. It follows from Property BC that $w \geq \Delta(\mathcal{S}, L')$ for all $L' \geq L$. Hence, the theorem holds. □

Corollary 4.7 *If $\phi(\mathcal{S}, L)$ is decreasing, $w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L > L' \geq L_0\ \}$, and $w \geq \phi(\mathcal{S}, L)$ for some $L \geq L_\phi$, then $w \geq \mathcal{S}.w_{\text{opt}}$.*

Proof. Follows trivially from Theorem 4.6. □

Non-decreasing monotonicity. The next theorem and corollary consider the case in which $\phi(\mathcal{S}, L)$ is non-decreasing with increasing L. These results are based on the limit of $\phi(\mathcal{S}, L)$ as $L \to \infty$, which is denoted $\mathcal{S}.w_\phi$. By Property MC[7] and the fact that $\phi(\mathcal{S}, L)$ cannot approach

[7]Specifically, Property MC guarantees that $\phi(\mathcal{S}, L)$ does not oscillate.

∞ in the limit,[8] this limit always exists. When $\phi(\mathcal{S}, L)$ tightly bounds $\Delta(\mathcal{S}, L)$, $\mathcal{S}.w_\phi$ typically equals the ideal weight of the supertask, which is the case when applying Rules 1 and 2 to all five of the example scenarios. For instance, consider (4.25). As $L \to \infty$, the second term (*i.e.*, $\frac{\Psi(\mathcal{S})}{L-\epsilon-2}$) goes to zero. Hence, $\mathcal{S}.w_\phi = \mathcal{S}.I_P$.

Theorem 4.7 *If $\phi(\mathcal{S}, L)$ is non-decreasing and $w \geq \mathcal{S}.w_\phi$, then*

$$w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L' \geq L_\phi\ \}.$$

Proof. Since $\phi(\mathcal{S}, L)$ is a non-decreasing function, the value $\mathcal{S}.w_\phi$ upper bounds $\phi(\mathcal{S}, L')$ for all $L' \geq L_\phi$. It follows from Property BC that the theorem holds. □

Corollary 4.8 *If $\phi(\mathcal{S}, L)$ is non-decreasing, $w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L_\phi > L' \geq L_0\ \}$, and $w \geq \mathcal{S}.w_\phi$, then $w \geq \mathcal{S}.w_{\text{opt}}$.*

Proof. Follows trivially from Theorem 4.7. □

Ensuring termination. The results presented above demonstrate how comparing a candidate weight to a single value of $\phi(\mathcal{S}, L)$ is sufficient to draw conclusions about an unbounded number of comparisons to $\Delta(\mathcal{S}, L)$ values. Forcing termination is equally trivial: we can ensure that the candidate weight w upper bounds either $\phi(\mathcal{S}, L)$ or $\mathcal{S}.w_\phi$ by assigning $w := \max(w, \phi(\mathcal{S}, L))$ or $w := \max(w, \mathcal{S}.w_\phi)$, respectively. In the next section, we present a general reweighting algorithm based on the properties described in this section.

```
procedure Reweight($\mathcal{S}$, $w_{\min}$, $w_{\max}$, $L_{\max}$, $n_{\max}$) returns boolean
1:  $n := 0$;
2:  InitGenerator($\mathcal{S}$);
3:  $L :=$ Generate();
4:  $\mathcal{S}.w := w_{\min}$;
5:  while $(L < L_\phi) \wedge (\mathcal{S}.w \leq w_{\max})$ do
6:      CheckWeight($\mathcal{S}$, $L$, $n$)
    od;
7:  if $\Psi(\mathcal{S}) \leq 0$ then
8:      $\mathcal{S}.w := \max(\mathcal{S}.w, \mathcal{S}.w_\phi)$
    else
9:      while $(L < L_{\max}) \wedge (n < n_{\max}) \wedge (\mathcal{S}.w < \phi(\mathcal{S}, L)) \wedge (\mathcal{S}.w \leq w_{\max})$ do
10:         CheckWeight($\mathcal{S}$, $L$, $n$)
        od;
11:     $\mathcal{S}.w := \max(\mathcal{S}.w, \phi(\mathcal{S}, L))$
    fi;
12: return $(\mathcal{S}.w \leq w_{\max})$

procedure CheckWeight($\mathcal{S}$, $L$, $n$)
13: $\mathcal{S}.w := \max(\mathcal{S}.w, \Delta(\mathcal{S}, L))$;
14: $n := n + 1$;
15: $L :=$ Generate()
```

Figure 4.7: Algorithm for selecting a safe supertask weight.

4.5 The Reweighting Algorithm

Using the results of the last section, a supertask weight can be selected using the algorithm shown in Figure 4.7. This algorithm takes five parameters and returns a boolean value. The return value is true if and only if an acceptable weight was found, where "acceptable" is defined by the parameters, as described below.

[8]Such behavior would imply that demand is unbounded. However, unbounded demand cannot be produced by a finite number of tasks.

4.5.1 Parameter Descriptions

The first and most obvious parameter is the supertask $\mathcal{S}$ for which a weight should be selected. $w_{\min}$ and $w_{\max}$ define the range of acceptable supertask weights, *i.e.*, an algorithm invocation returns true if and only if $\mathcal{S}.w \in [w_{\min}, w_{\max}]$ upon termination and $\mathcal{S}.w$ ensures the timeliness of the component tasks. These parameters are assumed to satisfy $0 \leq w_{\min} \leq w_{\max} \leq 1$. Our algorithm is conservative in that a failure (*i.e.*, a return value of false) does not preclude the existence of a weight in the range $[w_{\min}, w_{\max}]$ that is capable of ensuring timeliness.

$L_{\max}$ and $n_{\max}$ are the length limit and computation limit, respectively. $L_{\max}$ specifies the largest value of L to consider before forcing the invocation to terminate. Similarly, $n_{\max}$ specifies the maximum number of $\Delta(\mathcal{S}, L)$ values to check before forcing termination. Termination is forced using the approach described in the previous section.

Special cases. In addition to searching for safe supertask weights within a range of values, three special cases of reweighting are common in practice. First, the optimal[9] weight (*i.e.*, $\mathcal{S}.w_{\text{opt}}$) can be sought by invoking `Reweight(`$\mathcal{S}$, 0, 1, ∞, ∞`)`. (As discussed below, it may not be possible to identify the optimal weight in bounded time.) Second, the safety of a given weight w can be determined by invoking `Reweight(`$\mathcal{S}$, w, w, ∞, ∞`)`. Third, an upper bound on inflation can be computed by the call `Reweight(`$\mathcal{S}$, 0, 1, 0, ∞`)`. This call immediately forces termination of line 9. As a result, the return value is the most pessimistic solution that can be generated by `Reweight` and equals $\max(\max\{\ \Delta(\mathcal{S}, L) \mid L_\phi \leq L < L_0\ \}, \phi(\mathcal{S}, L_0))$. When $L_\phi = L_0$, this bound is simply $\phi(\mathcal{S}, L_0)$ and can be computed with invoking `Reweight`.

[9]This solution is optimal with respect to the presented approach. This weight is *not* guaranteed to be the smallest weight for which timeliness is guaranteed.

4.5.2 Algorithm Description

`Reweight` begins by initializing variables in lines 1–4. In line 5 and 6, each $L \in \mathcal{L}$ that satisfies $L < L_\phi$ is checked to ensure that $\mathcal{S}.w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L_\phi > L' \geq L_0\ \}$ holds at line 7. The remaining values are then handled by either line 8 or lines 9–11, based on whether $\phi(\mathcal{S}, L)$ is non-decreasing or decreasing, respectively. In the former case, Corollary 4.8 implies that if $\mathcal{S}.w \geq \mathcal{S}.w_\phi$, then $\mathcal{S}.w$ is guaranteed to be safe. Line 8 ensures that $\mathcal{S}.w \geq \mathcal{S}.w_\phi$ holds. In the latter case, weight restrictions must be checked (lines 9–10) until either a user-provided limit is reached (checked by the first two conditions given in line 9) or an value L is found that satisfies the conditions set forth in Corollary 4.7 (checked by the third condition given in line 9). In the event that termination is forced, line 11 ensures that the conditions set forth in Corollary 4.7 still hold and hence that $\mathcal{S}.w$ is a safe weight. (Note that if the **while** loop terminates due to the third condition in line 9, then line 11 has no effect.) Line 12 then reports whether the invocation was successful (see below).

`CheckWeight` is invoked to compare the current supertask weight to the $\Delta(\mathcal{S}, L)$ restriction, which is done in line 13. Line 14 then updates the computation counter to reflect the comparison. Finally, L is advanced to the next highest value of $L \in \mathcal{L}$ in line 15.

Correctness of the return value. An important property of `Reweight` is that $\mathcal{S}.w$ is non-decreasing after line 4, as implied by lines 8, 11, and 13. As a result, both $\mathcal{S}.w \geq w_{\min}$ and $\mathcal{S}.w > w_{\max}$ are invariant once established. Since line 4 establishes the $\mathcal{S}.w \geq w_{\min}$, it follows $\mathcal{S}.w \leq w_{\max} \Leftrightarrow \mathcal{S}.w \in [w_{\min}, w_{\max}]$ at line 12. To avoid unnecessary computation, both loops terminate immediately if failure ($\mathcal{S}.w > w_{\max}$) is detected (see lines 5 and 9).

4.5.3 Properties

We now state and prove basic properties of `Reweight`. First, a lower bound on $\mathcal{S}.w$ following the successful completion of a `Reweight` invocation is proved in the theorem given below. This theorem also establishes a lower bound on the inflation factor produced when reweighting a supertask using the approach described here.

Theorem 4.8 *If an invocation of* `Reweight` *on* $\mathcal{S}$ *terminates, then upon termination,* $\mathcal{S}.w \geq \mathcal{S}.w_\phi$ *holds if* $\Psi(\mathcal{S}) \leq 0$ *and* $\mathcal{S}.w > \mathcal{S}.w_\phi$ *holds if* $\Psi(\mathcal{S}) > 0$.

Proof. The $\Psi(\mathcal{S}) \leq 0$ case follows trivially from line 8 of `Reweight`. When $\Psi(\mathcal{S}) > 0$, $\phi(\mathcal{S}, L)$ approaches $\mathcal{S}.w_\phi$ from above in the limit. Hence, $\phi(\mathcal{S}, L) > \mathcal{S}.w_\phi$ holds for all L. Thus, $\mathcal{S}.w > \mathcal{S}.w_\phi$ holds after the eventual execution of line 11. It follows that $\mathcal{S}.w > \mathcal{S}.w_\phi$ holds upon termination also. $\square$

Theorem 4.8 raises the question of whether termination is guaranteed. When $\Psi(\mathcal{S}) \leq 0$ holds, termination obviously occurs. However, termination may not occur when $\Psi(\mathcal{S}) > 0$ holds. The following lemma, theorem, and corollary characterize the circumstances under which termination does not occur.

Lemma 4.1 *If* $\mathcal{S}.w > \mathcal{S}.w_\phi$ *is established during an invocation of* `Reweight`*, then termination is guaranteed.*

Proof. Termination can only be avoided by taking the code branch leading to line 9,[10] which only occurs when $\Psi(\mathcal{S}) > 0$. In this case, $\phi(\mathcal{S}, L)$ decreases as L increases. Because $\phi(\mathcal{S}, L)$ approaches $\mathcal{S}.w_\phi$ in the limit, the value of $\phi(\mathcal{S}, L)$ will eventually drop (and remain) below any weight in the range $(\mathcal{S}.w_\phi, 1]$.

[10]Since L_ϕ is required to be finite, the loop at line 5 must eventually terminate due to its first condition.

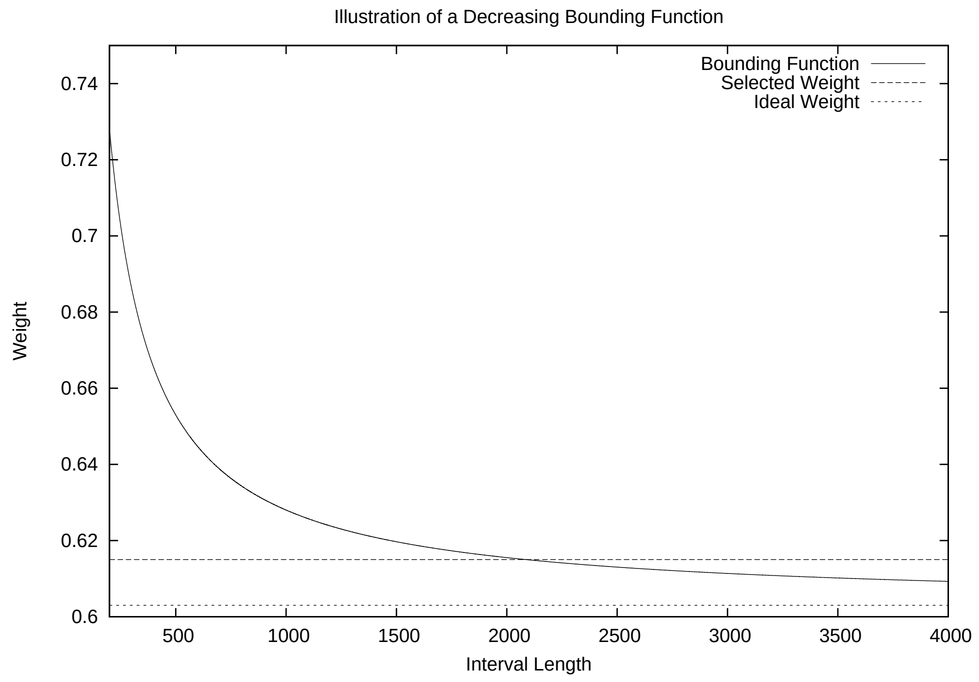


Figure 4.8: When $\phi(\mathcal{S}, L)$ is a decreasing function, the value of the function will eventually drop below any weight in the range $(\mathcal{S}.w_\phi, 1]$. In the graph, the "Bounding Function" and "Ideal Weight" curves correspond to $\phi(\mathcal{S}, L)$ and $\mathcal{S}.w_\phi$, respectively.

Figure 4.8 illustrates this property. In this example, $\mathcal{S}.w_\phi = 0.603$. Suppose that $\mathcal{S}.w = 0.615$ (> 0.603) is established during an invocation of `Reweight`. As shown, the value of $\phi(\mathcal{S}, L)$ drops below $\mathcal{S}.w$ around $L = 2200$. Hence, $\mathcal{S}.w \geq \phi(\mathcal{S}, L)$ holds for all $L \geq 2200$ since $\phi(\mathcal{S}, L)$ is a decreasing function. It follows that the loop at line 9 must eventually terminate due to its third condition. The lemma follows. $\square$

Theorem 4.9 *An invocation of* `Reweight` *on $\mathcal{S}$ does not terminate if and only if all of the following conditions hold:*

1. $\Psi(\mathcal{S}) > 0$;
2. $w_{\min} \leq \mathcal{S}.w_\phi$;
3. $\mathcal{S}.w_{\text{opt}} \leq \mathcal{S}.w_\phi$;
4. $\mathcal{S}.w_{\text{opt}} \leq w_{\max}$;
5. $L_{\max} = n_{\max} = \infty$.

Proof. By the algorithm's code listing, termination can be avoided only by becoming trapped in the **while** loop at line 9. To prove sufficiency, we show that the conditions given above guarantee that the loop is reached and that the four conditions in line 9 always hold (and hence that the loop never terminates). Condition 1 is sufficient to ensure that the loop is reached via line 7. Now consider the four conditions in line 9. By Condition 5, the first two conditions in line 9 always hold. It remains to show that the third and fourth conditions always hold.

To show that the third condition in line 9 holds, we show that $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant after the execution of line 4. We begin by explaining why $\mathcal{S}.w \leq \mathcal{S}.w_\phi \Rightarrow \mathcal{S}.w < \phi(\mathcal{S}, L)$. By Condition 1, $\phi(\mathcal{S}, L)$ is decreasing. Hence, $\phi(\mathcal{S}, L)$ approaches (but never equals) $\mathcal{S}.w_\phi$ in the limit. For instance, consider the definition of $\phi(\mathcal{S}, L)$ given in (4.22). In this case, $\mathcal{S}.w_\phi = \mathcal{S}.I_Q$ and $\phi(\mathcal{S}, L) = \mathcal{S}.w_\phi + \frac{\Psi(\mathcal{S})}{L-\epsilon}$. The term $\frac{\Psi(\mathcal{S})}{L-\epsilon}$ approaches (but never equals) zero as $L \to \infty$. Hence, if $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant, then it follows that $\mathcal{S}.w < \phi(\mathcal{S}, L)$ holds for all L (and that the third condition in line 9 always holds).

We now prove that $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant after line 4. It follows from Condition 2 that $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ holds immediately after line 4. By (4.11), $\Delta(\mathcal{S}, L) \leq \mathcal{S}.w_{\text{opt}}$ for all L. Hence, by Condition 3, $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ holds after each execution of line 13. Hence, $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant after line 4.

Similarly, we show that the fourth condition in line 9 always holds by showing that $\mathcal{S}.w \leq w_{\max}$ is invariant after line 4. By Condition 2 and the property $w_{\min} \leq w_{\max}$, $\mathcal{S}.w \leq w_{\max}$ holds immediately after line 4. By Condition 4, $\mathcal{S}.w \leq w_{\max}$ holds after each execution of line 13. Hence, $\mathcal{S}.w \leq w_{\max}$ is invariant after line 4. This completes the proof of sufficiency.

Now, consider necessity. If Condition 1 does not hold, then the loop in line 9 is never executed and hence cannot prevent termination. We now argue that negating each of the remaining conditions guarantees termination. First, if Condition 5 does not hold, then one of the first two conditions in line 9 will eventually cause termination. If Condition 4 does not hold, then $\mathcal{S}.w > w_{\max}$ is eventually established by line 13 (unless it is established sooner by line 4), which results in termination of the loop by its fourth condition. Similarly, if either Condition 2 or 3 does not hold, then $\mathcal{S}.w > \mathcal{S}.w_\phi$ is eventually established by either line 4 or 13, respectively. By Lemma 4.1, termination is ensured once $\mathcal{S}.w > \mathcal{S}.w_\phi$ is established. This completes the proof of necessity. □

Corollary 4.9 *If* `Reweight` *is invoked on $\mathcal{S}$, then termination is guaranteed whenever either $w_{\min} > \mathcal{S}.w_\phi$ or $\Psi(\mathcal{S}) \leq 0$ holds.*

Proof. Follows trivially from Theorem 4.9. □

4.5.4 Balancing Speed and Inflation

`Reweight` provides three methods for controlling the speed-versus-accuracy trade-off, each of which involves the use of one of $w_{\min}$, $L_{\max}$, or $n_{\max}$. Each method discussed here provides a predictable lower bound on reweighting overhead and an upper bound on the amount of computation performed. For the initial description of each method (given below), we assume that only one of $w_{\min}$, $L_{\max}$, or $n_{\max}$ is set to a non-trivial value.[11] When two or more parameters are used simultaneously, the speed and accuracy of the process is determined by the parameter that implies the smallest number of computations.

[11] $L_{\max}$ and $n_{\max}$ are set to ∞ when not used; $w_{\min}$ can be set to any value in $[0, \mathcal{S}.w_\phi]$.

First, the reweighting process can be seeded with a non-ideal minimum weight, *i.e.*, $w_{\min}$ can be assigned a value larger than $\mathcal{S}.w_{\phi}$. The benefit of such an assignment can be seen in Figure 4.8. Specifically, $\phi(\mathcal{S}, L)$ crosses larger weights at lower values of L. It is straightforward to calculate the smallest L value for which $\phi(\mathcal{S}, L)$ is less than $w_{\min}$ (and hence, for which the third condition at line 9 is guaranteed to hold). Let L_{last} denote this derived value. It follows that using $w_{\min}$ produces the same result as passing L_{last} as the $L_{\max}$ parameter.

Second, the $L_{\max}$ parameter can be used to bound the search space of the reweighting process. As in the previous case, the $w_{\min}$ parameter can be used to achieve the same result. Specifically, the use of $L_{\max}$ achieves the same result as passing $\phi(\mathcal{S}, L_{\max})$ as the $w_{\min}$ parameter. Hence, the accuracy and speed bounds are computed as in the previous case.

Finally, the $n_{\max}$ parameter can be used to bound the number of $\Delta(\mathcal{S}, L)$ computations. Unfortunately, it is difficult to characterize how $n_{\max}$ relates to using the $L_{\max}$ parameter due to the fact that only a subset of the L values are actually checked. Under the pessimistic assumption that every L value is tested, $L_{last} = L_0 + n_{\max} - 1$. (Here, we assume that termination is forced.) Using this relationship, $n_{\max}$ can be related to both $L_{\max}$ and $w_{\min}$ as described above.

4.6 Experimental Results

In this section, we present the results of an experimental evaluation of `Reweight`. This evaluation consisted of two studies. The first study focused on the reweighting overhead produced by scheduling periodic tasks with each of the QB-EPDF, QB-EDF, and FP-EDF approaches.[12] The goal of this study was to determine both the relative inflation produced

[12]For the QB-EPDF and QB-EDF scenarios, task parameters must first be mapped to weights using Lemmas 3.4–3.7. See pages 73–79.

by each approach and which approach is likely to provide the lowest overhead. Since the FP-EDF-NP and FP-EDF-CS scenarios produce the same reweighting overhead as FP-EDF, they were not considered. Experimental studies that consider the synchronization overhead (with and without supertasking) are presented later in Chapters 5 and 6.

The second study focused on the speed-versus-accuracy trade-off. Again, the QB-EPDF, QB-EDF, and FP-EDF scenarios were considered. The primary goal of this study was to approximate the relationship between inflation and the amount of computation performed. Specifically, the objective of this study was to determine the number of computations that must be performed to produce a solution that approximately equals $\mathcal{S}.w_{\text{opt}}$. A secondary goal of this study was to determine whether this point of diminishing returns is stable, *i.e.*, whether the location of the point is reasonably constant across task sets.

4.6.1 Study 1: Reweighting Overhead

In this section, we present the details and results of the first study. This study was designed to approximate the inflation produced by each of the QB-EPDF, QB-EDF, and FP-EDF scenarios. In addition, this study considers both cases in which jobs do and do not align with slot boundaries. Recall that alignment reduces both reweighting (see page 118) and mapping overhead (see pages 73–79). The purpose of including alignment in this study is to determine whether the degree of this benefit is significant.

Sample space. Both studies presented in this chapter are based on comparing randomly generated component task sets uniformly drawn from a sample space. For this study, the sample space was defined as follows:

- $|\mathcal{S}|$ is in the range $2, \ldots, 40$;

- $\mathcal{S}.u$ is in the range $0.02, \ldots, 0.9$;
- $T.p$ is in the range $100, \ldots, 5000$;
- the minimum $T.p$ is in the range $5, \ldots, 250$.

Sampling. To ensure that the random samples provided reasonable coverage of this sample space, the following parameters were systematically chosen as described below:

- $|\mathcal{S}|$ was set to each value in the range $2, \ldots, 40$;
- $\mathcal{S}.u$ was set to each value in the set $\{\ 0.02x \mid 1 \leq x \leq 45\ \}$;
- the minimum $T.p$ was set to each value in the set $\{\ 5x \mid 1 \leq x \leq 50\ \}$.

For each combination of the above parameter assignments, four component task sets were generated and reweighted.

Relevance. This experimental methodology (*i.e.*, defining a sample space and then generating samples so that reasonably uniform coverage is ensured) is used throughout the dissertation. The benefit of this approach is that it reveals how inflation is impacted by changing each systematically chosen parameter. The primary disadvantage is that such a study illustrates only general performance trends and the relative performance of the alternative approaches. Since it is not clear where real task sets lie within the sample space, it is not possible to draw conclusions concerning the amount of inflation that is likely to occur in practice.

Measurement. The following four measurements were taken during the experimental runs.

No Supertasks: a baseline measurement of the total weight of all tasks before reweighting; this measurement reflects mapping overhead alone.[13]

[13] Mapping overhead results from the use of Lemmas 3.4–3.7 to map periodic and sporadic tasks to weights.

QB-EPDF: the weight assigned to the supertask under the QB-EPDF scenario; this measurement reflects both mapping and reweighting overhead.

QB-EDF: the weight assigned to the supertask under the QB-EDF scenario; this measurement reflects both mapping and reweighting overhead.

FP-EDF: the weight assigned to the supertask under the FP-EDF scenario; this measurement reflects reweighting overhead alone since fully preemptive supertasking does not require parameter mapping.

To ensure termination, $\mathcal{S}.w_\phi + 10^{-5}$ was passed as $w_{\min}$ when invoking `Reweight`. The quantity 10^{-5} was used because we consider such a small degree of inflation to be negligible. Only samples which were schedulable under all approaches were considered. We refer to such samples as *valid*. When invalid samples were generated, they were simply discarded and replaced by a new sample. We consider the implications of this action below.

Sampling validity. Restricting attention to valid samples is necessary to enable quantitative comparison of the approaches, *i.e.*, a task set that is schedulable cannot be compared to one that is not schedulable. Unfortunately, filtering samples in this way can produce results that reflect only the performance of a subset of the sample population. To estimate the impact of filtering, the random task set studies presented in this dissertation include estimates of the *sampling validity*. We define sampling validity for each sample mean as follows:

$$\text{sampling validity} = \frac{\text{number of valid samples}}{\text{total number of generated samples}}.$$

Sampling validity is an estimator of the fraction of the sample population represented by the sample mean. This measurement carries the additional implication that the unrepresented

Aligned	**Mean**	**Standard Deviation**	**Halflength**	**Best**
No Supertasks	0.017711	0.014587	0.00006342	*N/A*
QB-EPDF	0.023027	0.028324	0.00012315	0.034%
QB-EDF	0.017732	0.014678	0.00006381	0.065%
FP-EDF	0.000173	0.004108	0.00001786	99.901%

Not Aligned	**Mean**	**Standard Deviation**	**Halflength**	**Best**
No Supertasks	0.018957	0.015318	0.00006660	*N/A*
QB-EPDF	0.024224	0.028523	0.00012401	0.040%
QB-EDF	0.018987	0.015574	0.00006771	0.043%
FP-EDF	0.000215	0.005332	0.00002318	99.918%

Figure 4.9: Summary of the results of the first reweighting experiment when job releases and deadlines aligned to slot boundaries (upper) and did not align (lower).

fraction of the population consists of samples that are not schedulable under at least one of the considered approaches.

Ideally, sampling validity should be unity. Indeed, the sampling validity of all means presented in this section are approximately unity, *i.e.*, the difference is negligibly small. For this reason, we omit graphs of the sampling validity estimates here. However, in later chapters, it is not always possible to achieve high sampling validity in all cases. Hence, considering the sampling validity of a mean will be essential to the proper interpretation of the experimental results.

Overall Performance. The tables shown in Figure 4.9 summarize the overall results of the experiments. Separate summaries are given for the use of aligned and unaligned jobs.

Consider the aligned results first. The second column gives the mean inflation (*i.e.*, $\mathcal{S}.w - \mathcal{S}.u$) observed over all collected samples. As shown, FP-EDF provided two orders of magnitude improvement by avoiding mapping overhead. QB-EDF shows very little reweighting overhead on average, *i.e.*, its mean is approximately equal to that of the "No Supertasks" case. QB-EPDF, on the other hand, performs the worst, but still wastes only around 2.3% of a processor on average.

The second column shows the standard deviation of samples. These measurements seem to suggest that variation tracks the mean reasonably well, *i.e.*, higher means seems to be accompanied by more variation in the samples. Notice that the standard deviation of FP-EDF is more than an order of magnitude greater than the mean. The low mean suggests that the vast majority of samples showed small inflation. However, the high variance suggests that some samples showed a significant degree of inflation. Hence, these results suggest that FP-EDF performs poorly in some pathological cases.

The halflength[14] of a 99% confidence interval is given in the fourth colum. As shown, the halflengths are negligibly small compared to the means for all measurements except FP-EDF. For this sole exception, however, the magnitude of the halflength is of no importance due to the (negligibly) small magnitude of the mean.

The fifth column reports the percentage of sample task sets that favored each approach. Specifically, 0.034% of the sample sets showed no advantage to using the QB-EDF or FP-EDF approaches instead of the QB-EPDF approach. Similarly, 0.065% of the sets favored using QB-EDF instead of QB-EPDF, but did not improve further when using FP-EDF. Finally, 99.901% of the sample sets favored using FP-EDF instead of either QB-EPDF or QB-EDF.

Finally, consider the non-aligned results. The trends in this data match those in the aligned results. As shown, using non-aligned jobs does increase the mean inflation. However, the magnitude of this increase does not appear to be significant on average, *i.e.*, the improvement produced by using aligned jobs represents less than 0.25% of a single processor.

Impact of the task count. We now consider how inflation varied as each of the three

[14]The halflength is the absolute difference between the mean and either endpoint of the interval.

parameters used to define the sample space was varied. Consider the task count first. Figures 4.10 and 4.11 show how inflation varies as the task count changes when using aligned and non-aligned jobs, respectively. As shown, the impact of aligned jobs appears negligible. Notice also that the QB-EDF and "No Supertasking" curves are virtually co-linear. We summarize relationships suggested by these graphs below.

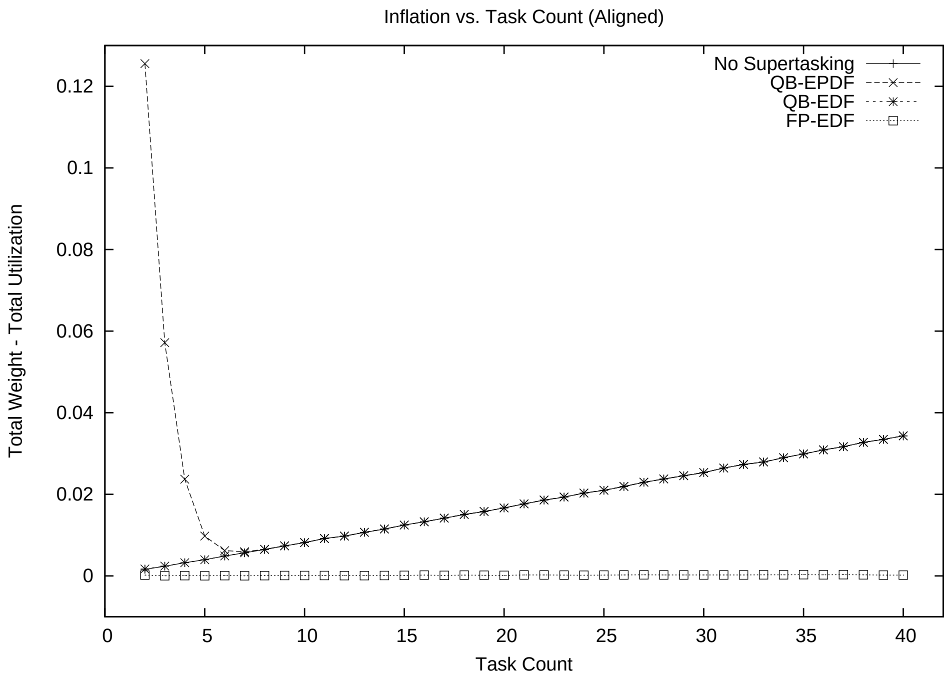
Observation 4.1 *The overhead of quantum-based supertasking increases linearly with the task count.*

Explanation: Mapping overhead is a per-task overhead. Hence, overhead must increase linearly when applying quantum-based supertasking to periodic or sporadic tasks due to the mapping overhead. (Notice that the "No Supertasking" measurement shows this same relationship.) Since mapping overhead depends on the task parameters, no conclusions can be drawn concerning the rate at which this overhead increases.

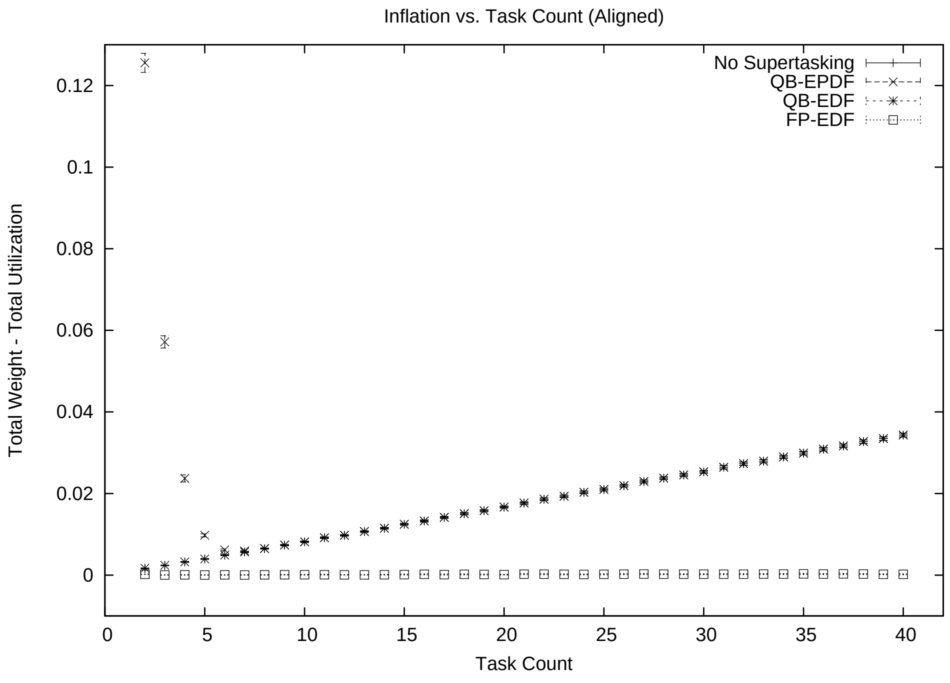
Observation 4.2 *Reweighting overhead under QB-EPDF is high for very low task counts, but this overhead diminishes quickly.*

Explanation: The fact that inflation under QB-EPDF is higher than under QB-EDF and FP-EDF follows simply from the fact that pseudo-deadlines impose additional constraints that the scheduler must satisfy. Hence, scheduling is inherently more difficult under QB-EPDF than under QB-EDF and FP-EDF, which do not impose these intermediate deadlines.

The property that reweighting overhead is higher for smaller task counts follows from the mandatory demand bound given in Corollary 4.4. Letting $T.c = 0$ for all tasks, $Q = 1$, and $\mathcal{S}_L = \mathcal{S}$, the tightest bound from the corollary simplifies to $\sum_{T \in \mathcal{S}} \lfloor T.w \cdot L \rfloor$. The observed tendency follows from the use of the floor operator. Because this operator decreases the value

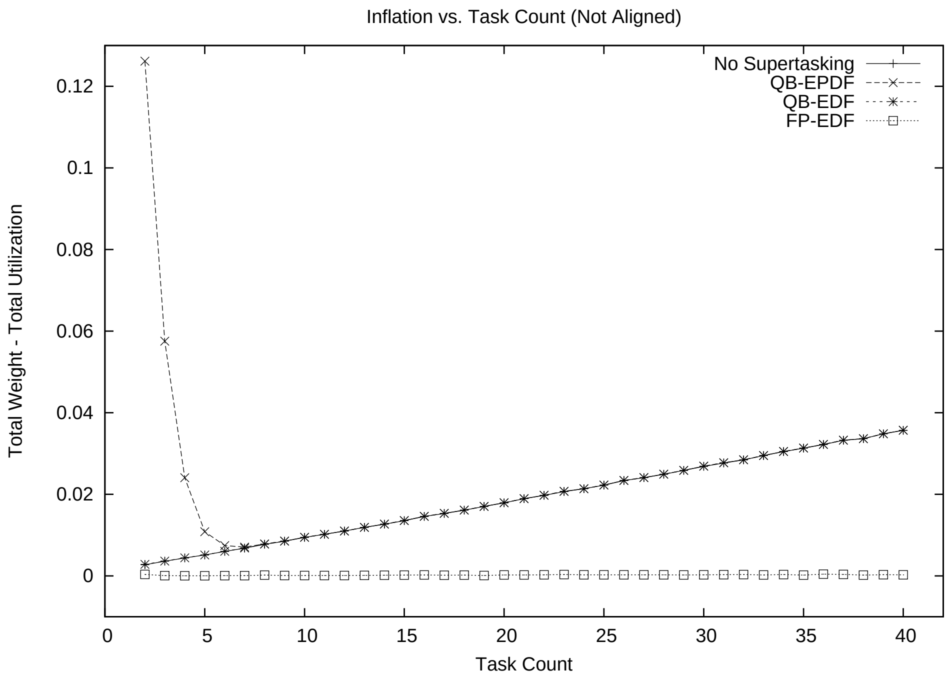


(a) Sample Means

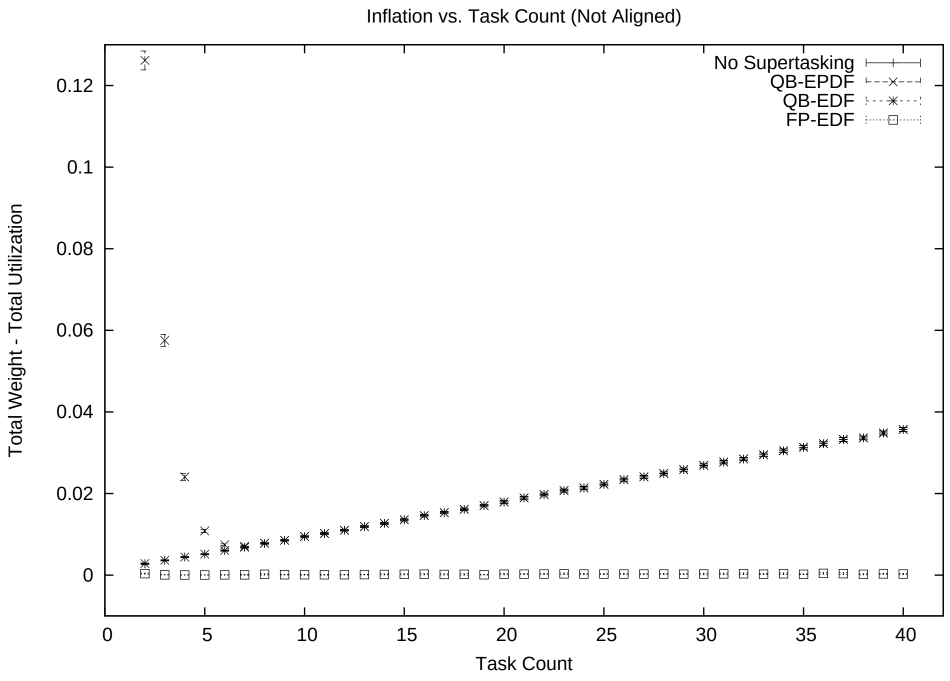


(b) 99% Confidence Intervals

Figure 4.10: Plots show how inflation scales with increasing task count when jobs are constrained to align to slot boundaries. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) Sample Means



(b) 99% Confidence Intervals

Figure 4.11: Plots show how inflation scales with increasing task count when jobs are not constrained to align to slot boundaries. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

of its argument, increasing the number of such operators tends to produce lower demand estimates on average. Lower demand estimates then translate into less inflation. Since the number of floor operators in the summation equals the number of tasks, it follows that higher task counts tend to produce less inflation. In addition, since the mandatory demand bounds used by QB-EDF and FP-EDF also utilize such floor operators, these approaches should also exhibit this tendency. (Due to the very low reweighting overhead produced by these approaches, the trend is not visible in the graphs presented here.)

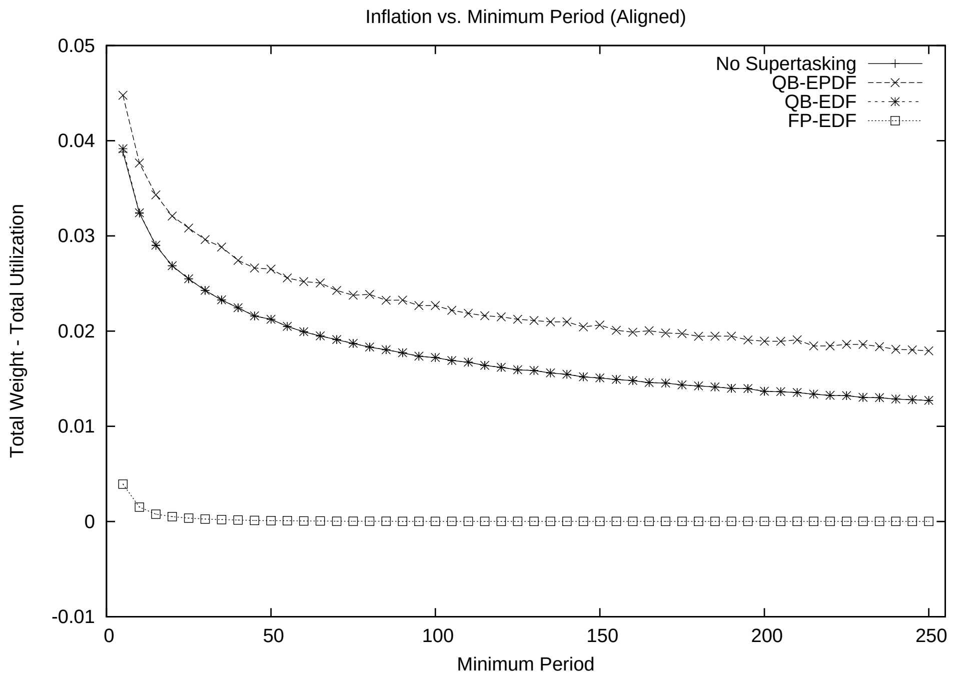
Observation 4.3 *For practical purposes, the task count has a negligible impact on the reweighting overhead of the job-based supertasking approaches.*

Explanation: Intuitively, all forms of supertasking should exhibit increased inflation at low task counts, as QB-EPDF does. Apparently, the reweighting overhead under QB-EDF and FP-EDF is simply too small in magnitude for this behavior to be observed.

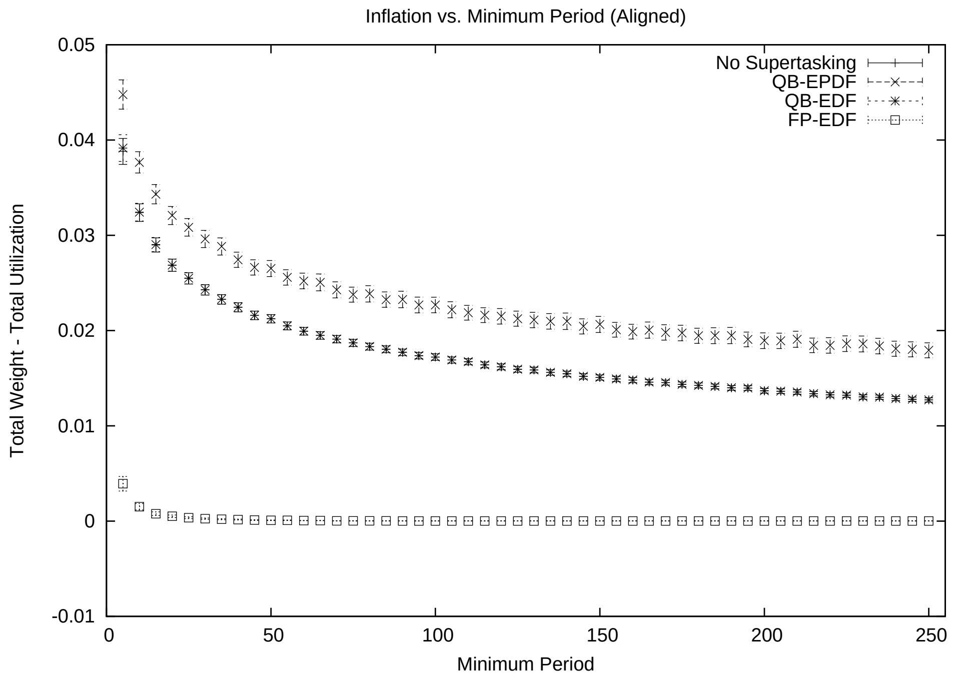
Observation 4.4 *Reweighting overhead appears to be negligible on average for all approaches when the task count is 7 or more.*

Impact of the minimum period. Figures 4.12 and 4.13 show plots of the mean inflation against the minimum period for aligned and non-aligned jobs, respectively. Again, the QB-EDF and "No Supertasking" cases are virtually co-linear. Unlike the previous graphs, these graphs show some benefit (up to around 0.005) to using aligned jobs, but only when periods can be very small.

Observation 4.5 *The reweighting overhead of the QB-EPDF approach is approximately independent of the minimum period.*

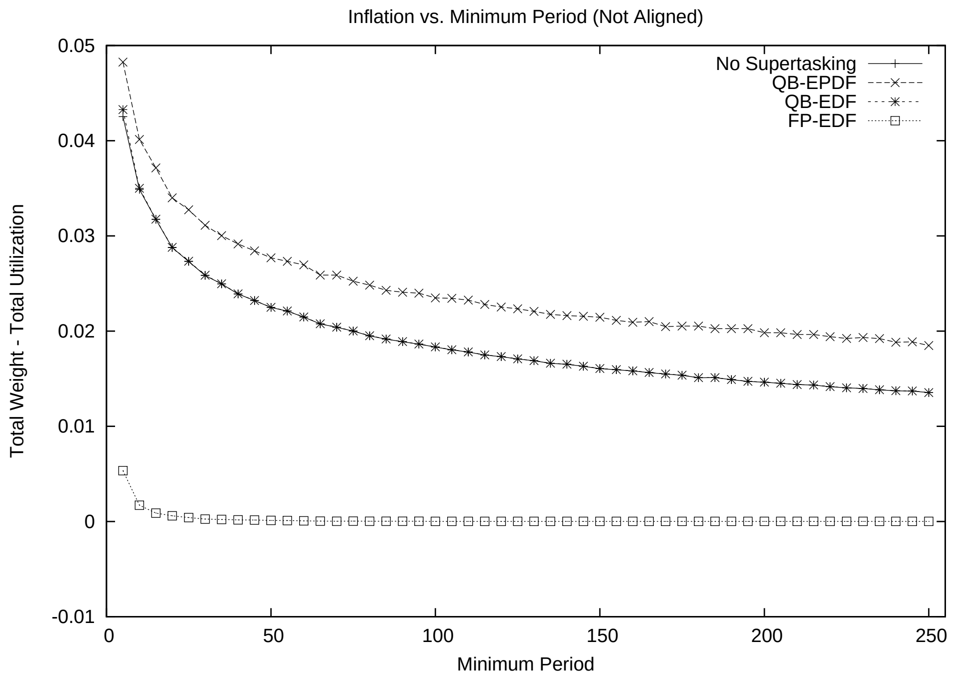


(a) Sample Means

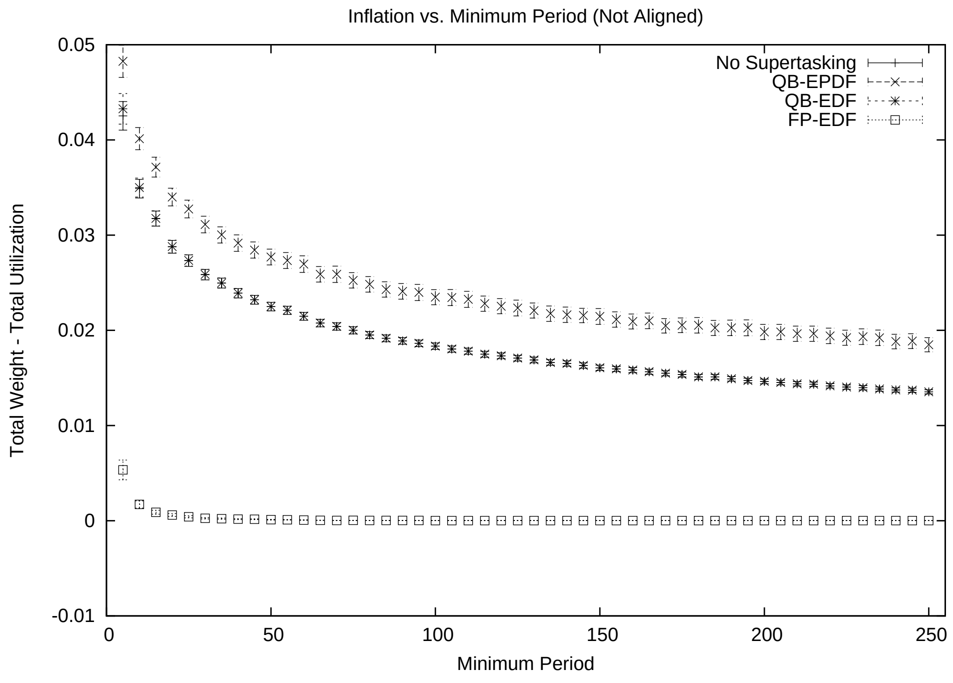


(b) 99% Confidence Intervals

Figure 4.12: Plots show how inflation scales with minimum period when jobs are constrained to align to slot boundaries. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) Sample Means



(b) 99% Confidence Intervals

Figure 4.13: Plots show how inflation scales with the minimum period when jobs are not constrained to align to slot boundaries. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

Explanation: The inflation produced by the QB-EPDF approach is approximately 0.005 higher than that of the "No Supertasks" case at all points.

Observation 4.6 *Constraining job arrivals and deadlines to lie on slot boundaries provides minor benefit to the QB-EPDF and QB-EDF approaches on average when the smallest period is at most ten slots, but only negligible benefit otherwise.*

Observation 4.7 *Both reweighting and mapping overheads are inversely proportional to the minimum task period.*

Explanation: Consider mapping overhead first. When rounding an execution requirement up to the next quantum multiple, the impact on the amount of processor time requested by the task in the limit depends on the task period. Specifically, the rounding impacts tasks with higher periods to a lesser degree. The observed relationship follows.

Now, consider the reweighting overhead. A simple upper bound on inflation is given by $\phi(\mathcal{S}, L_0)$. As demonstrated by (4.22) and (4.25), $\phi(\mathcal{S}, L) \approx I + \frac{\Psi(\mathcal{S})}{L}$ where I denotes the ideal weight of the supertask (*i.e.*, either $\mathcal{S}.I_Q$ or $\mathcal{S}.I_P$). Hence, reweighting overhead is bounded (approximately) by $\frac{\Psi(\mathcal{S})}{L_0}$. By (4.7) and (4.6), L_0 scales directly with periods when scheduling jobs or subtasks defined by pseudo-jobs. By (4.5), L_0 scales with windows sizes under EPDF scheduling. Window sizes decrease with increasing weight. By Lemmas 3.4–3.7, weights increase as periods decrease. Hence, L_0 tends to scale proportionally to periods under EPDF as well. As a result, the worst-case and average-case reweighting overhead tends to decrease as periods are increased.

Impact of the utilization. Figures 4.14 and 4.15 show plots of the inflation against utilization for aligned and non-aligned jobs, respectively. Again, the QB-EDF and "No Supertasking" cases are virtually co-linear in these graphs. These graphs suggest that an improvement of up to 0.005 is produced by job alignment for higher utilizations.

Observation 4.8 *Constraining job arrivals and deadlines to lie on slot boundaries provides minor benefit under the QB-EPDF and QB-EDF approaches on average when the utilization exceeds approximately 0.25, but only negligible benefit below that threshold.*

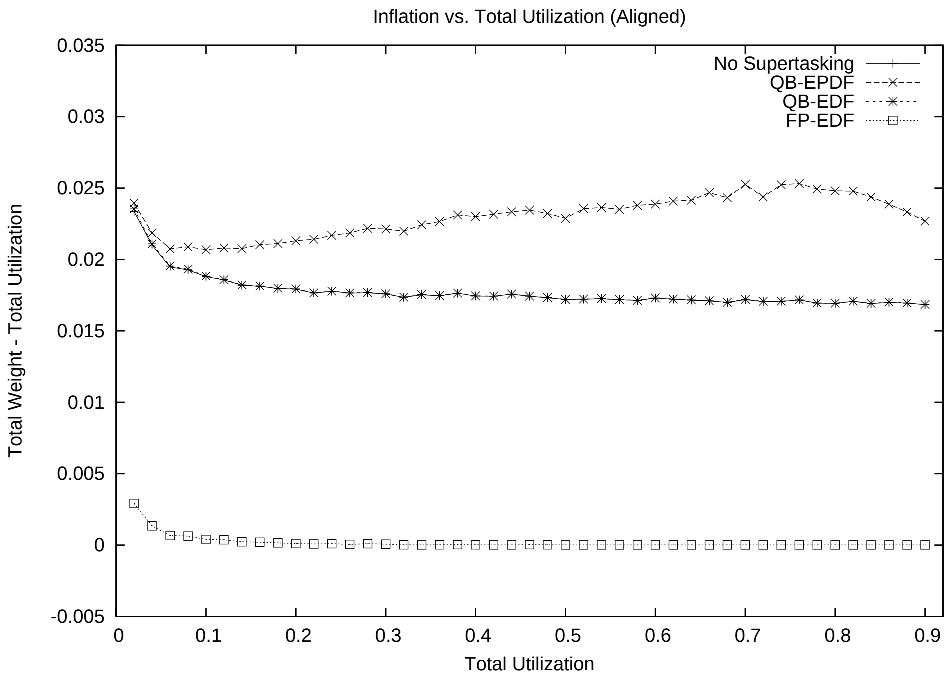
Observation 4.9 *Both reweighting and mapping overheads increase as the utilization decreases below 0.15.*

Explanation: When utilization is low, execution requirements tend to be small relative to the slot size. As a result, task execution requirements are much less than one quantum. Such parameters increase the mapping overhead. This same problem occurs for supertasks, but takes a slightly different form. Specifically, suppose that a supertask requires $\frac{3}{4}Q$ units of processor time to satisfy the demand over a given interval length. In this case, the supertask must still request a full quantum due to quantum-based scheduling.

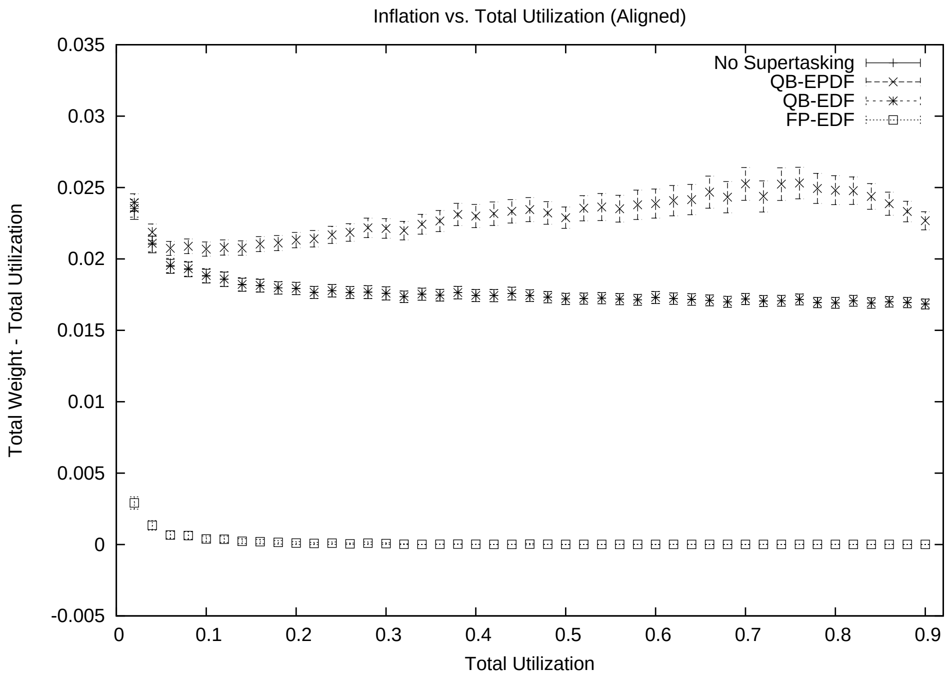
The behavior of QB-EPDF with respect to utilization was unexpected. Specifically, the shape of QB-EPDF's curve was expected to more closely resemble that of QB-EDF, as it does on the previous graphs. Further study is apparently needed to determine whether inflation under QB-EPDF is somehow dependent on the utilization.

4.6.2 Study 2: Speed Versus Accuracy

In this section, we present the details and results of the second study. This study was designed to evaluate the speed-versus-accuracy trade-off. Specifically, two experiments were conducted. In the first experiment, the value of $n_{\max}$ was varied and the impact on the inflation was

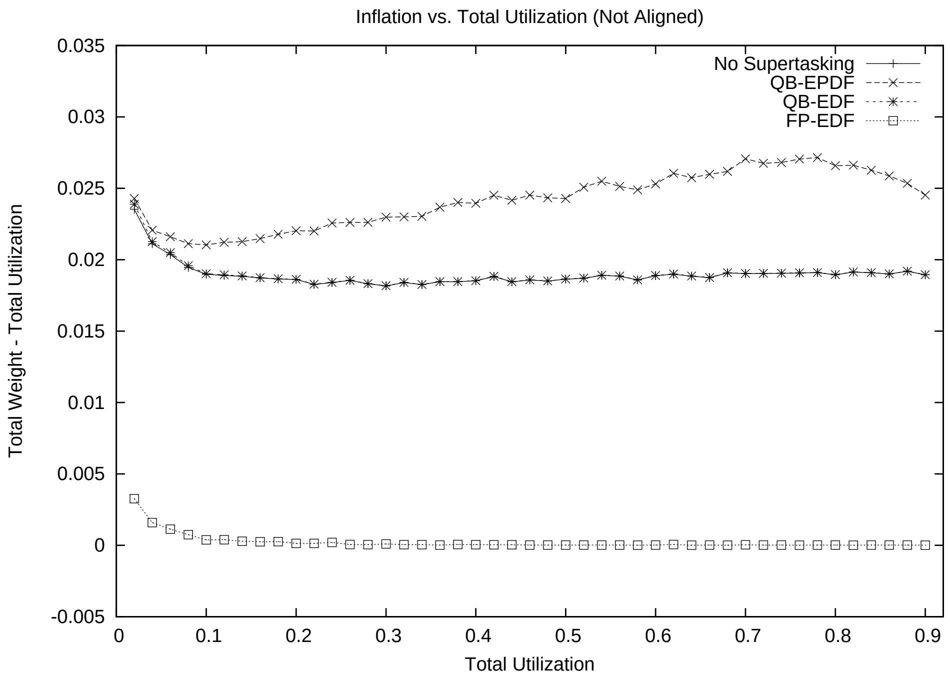


(a) Sample Means

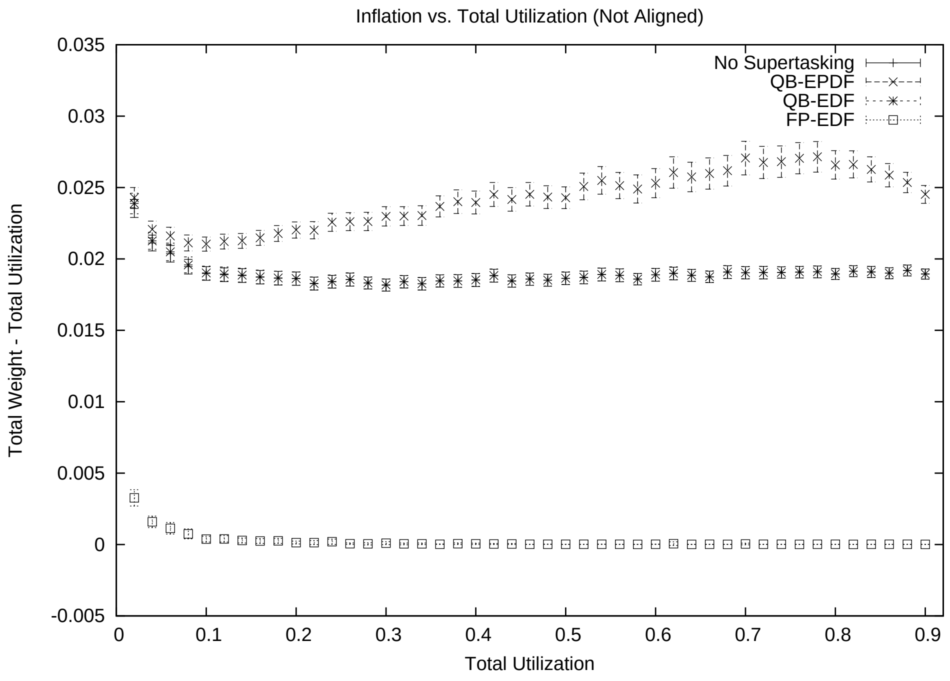


(b) 99% Confidence Intervals

Figure 4.14: Plots show how inflation scales with increasing utilization when jobs are constrained to align to slot boundaries. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) Sample Means



(b) 99% Confidence Intervals

Figure 4.15: Plots show how inflation scales with increasing utilization when jobs are not constrained to align to slot boundaries. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

observed. In the second experiment, the value of $w_{\min}$ was varied and the impact on the number of computations performed was observed. Since the impact of $L_{\max}$ depends on the task periods (due to $\mathcal{L}$), the impact of setting this parameter to a specific value will certainly not impact all task sets equally. Hence, this parameter was not considered in this experiment.

Sample space. For this study, the sample space was defined as follows:

- $n_{\max}$ is in the range $1, \ldots, 100000$; (first experiment only)
- the difference $w_{\min} - \mathcal{S}.w_\phi$ is in the range $0.00001, \ldots, 0.1$; (second experiment only)
- $|\mathcal{S}|$ is in the range $2, \ldots, 40$;
- $\mathcal{S}.u$ is in the range $0.02, \ldots, 0.9$;
- $T.p$ is in the range $100, \ldots, 2000$;
- the minimum $T.p$ is in the range $5, \ldots, 250$.

Sampling. For these experiments, component task sets were randomly generated while respecting the following constraints:

- $n_{\max}$ was set to each value in the set $\left\{ 10^{\frac{x}{4}} \;\middle|\; 2 \leq x \leq 18 \right\}$; (first experiment only)
- the difference $w_{\min} - \mathcal{S}.w_\phi$ was set to each value in the set $\left\{ 10^{\frac{x}{4}} \;\middle|\; -20 \leq x \leq -5 \right\}$; (second experiment only)
- $|\mathcal{S}|$ was set to each value in the set $\{ 4x \mid 1 \leq x \leq 10 \}$;
- $\mathcal{S}.u$ was set to each value in the set $\{ 0.05 + 0.1x \mid 0 \leq x \leq 8 \}$;
- the minimum $T.p$ was set to each value in the set $\{ 5x \mid 1 \leq x \leq 50 \}$.

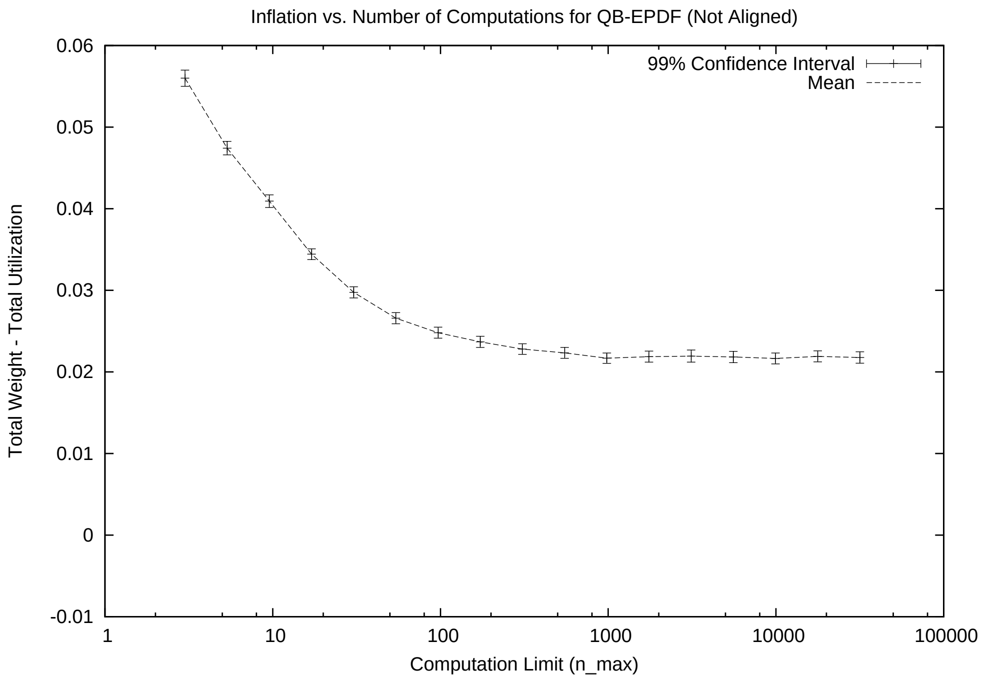
For each combination of these parameter assignments, one component task set was generated and reweighted.

Per-computation execution time. To place the estimates of computation in perspective, we ran the reweighting algorithm on a 1.4 GHz Pentium 4 desktop computer and measured the resulting mean per-computation execution time, *i.e.*, the total execution time divided by the value of n upon termination of the algorithm. Since several computations within the algorithm have time complexity $O(|\mathcal{S}|)$, we varied $|\mathcal{S}|$ to determine how the execution time scales. The collected data suggests that the per-computation execution time in microseconds is approximately $1.32 + 0.1357 \cdot |\mathcal{S}|$, *i.e.*, on the order of a few microseconds. Hence, a single task set can be reweighted in at most a few seconds, even when high precision is desired.

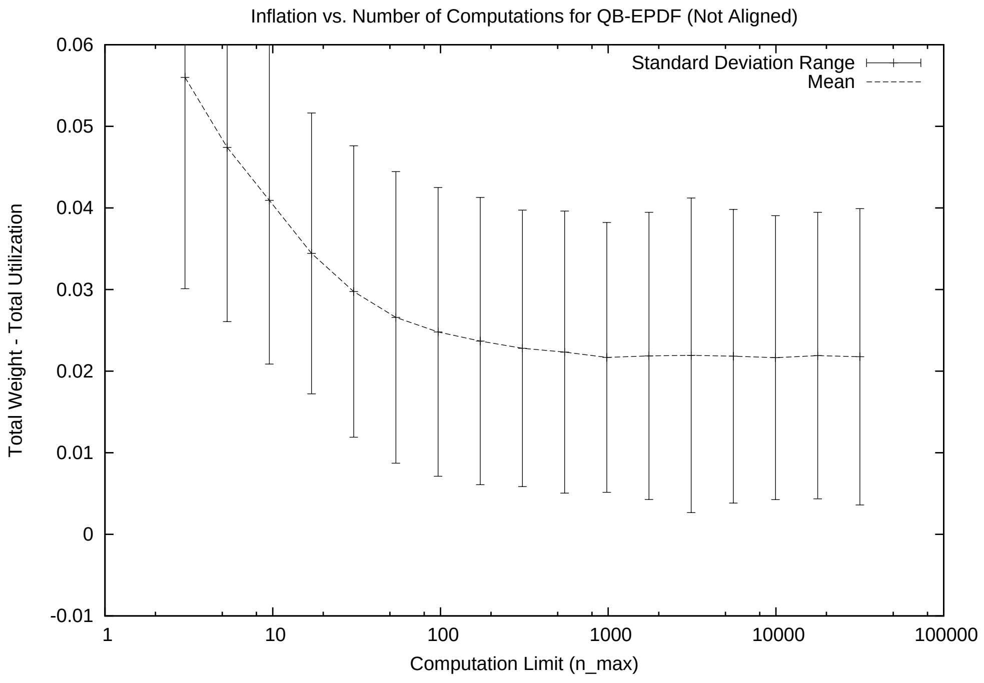
Using $n_{\max}$ with QB-EPDF. Figure 4.16 shows the impact of $n_{\max}$ under the QB-EPDF scenario. (In this and later cases, both aligned and non-aligned jobs were tested and found to produce very similar results. Because of this, only the graphs for the non-aligned case are presented.) As shown in Figure 4.16(a), the mean inflation produced by reweighting stabilizes at approximately $n_{\max} = 1000$. Unfortunately, the variance of the observed samples was relatively high, as Figure 4.16(b) shows. This suggests that the location of the stabilizing point will likely vary significantly between task sets.

Using $n_{\max}$ with QB-EDF. Figure 4.17 shows the impact of $n_{\max}$ under the QB-EDF scenario. In this case, the mean inflation stabilizes very quickly, *i.e.*, around $n_{\max} = 10^{1.5} \approx 32$. Unfortunately, the variance of the observed samples was high, as in the QB-EPDF case (see Figure 4.17(b)).

Using $n_{\max}$ with FP-EDF. Figure 4.18 shows the impact of $n_{\max}$ under the FP-EDF

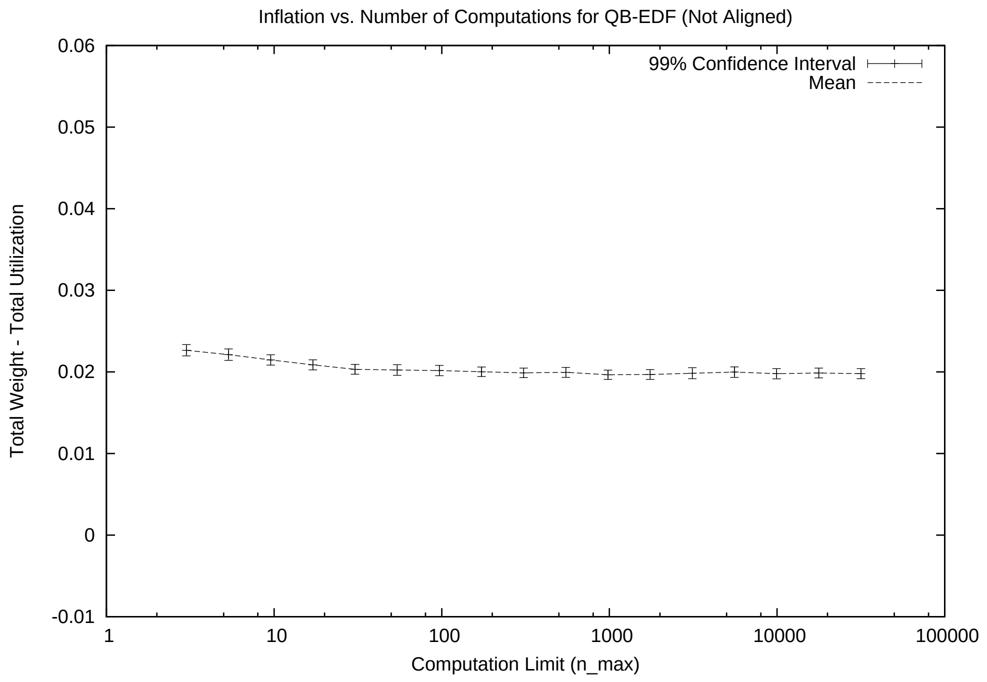


(a) 99% Confidence Interval

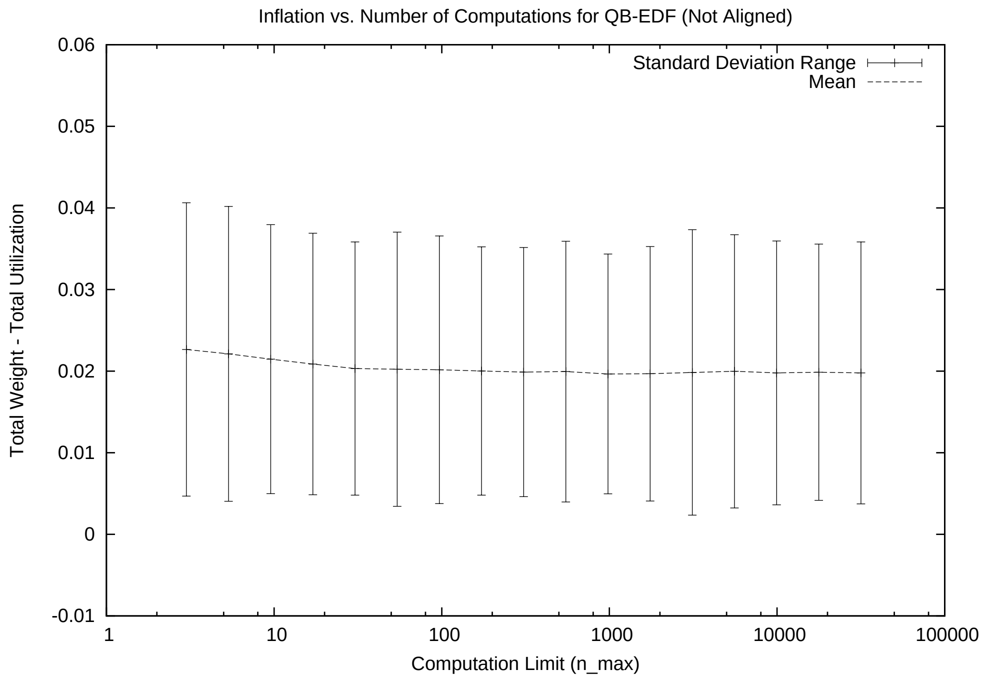


(b) Standard Deviation

Figure 4.16: Plots show how the inflation varies as the computation limit is increased under the QB-EPDF scenario. The figure shows **(a)** the sample means and **(b)** the interval defined by one standard deviation.



(a) 99% Confidence Interval



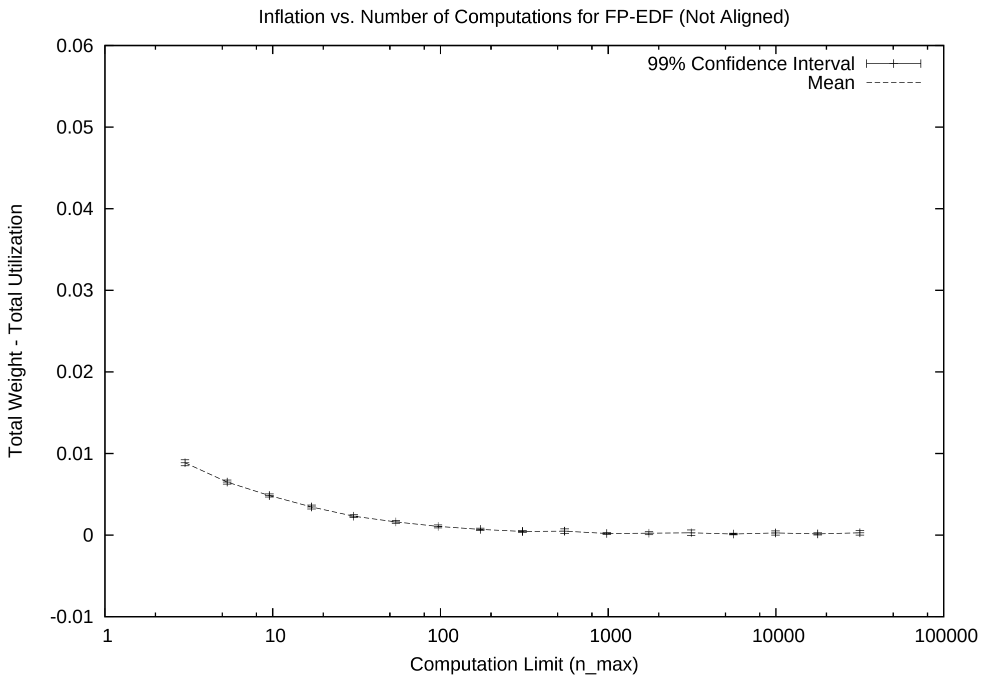
(b) Standard Deviation

Figure 4.17: Plots show how the inflation varies as the computation limit is increased under the QB-EDF scenario. The figure shows **(a)** the sample means and **(b)** the interval defined by one standard deviation.

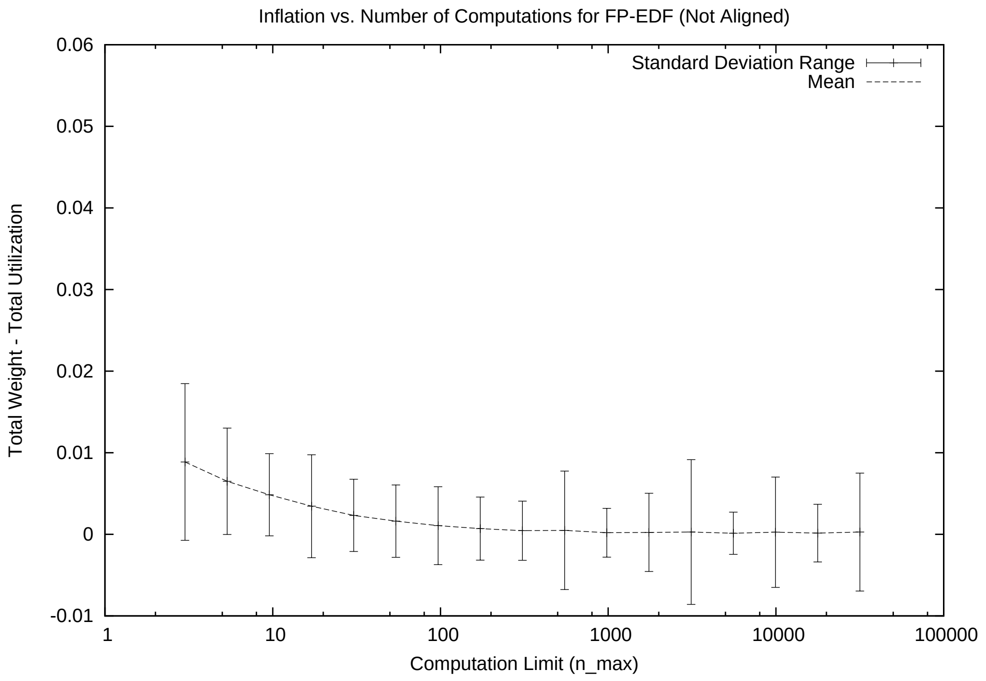
scenario. As in the QB-EPDF case, the mean does not appear to stabilize until approximately $n_{\max} = 1000$. However, the mean is reasonably stable much earlier, *i.e.*, around $n_{\max} = 100$. Also, the variance in the samples appears to be smaller under FP-EDF than under the quantum-based scenarios (see Figure 4.18(b)). This is likely due to the small magnitude of the inflation. Unfortunately, the variance is still sufficiently high to suggest that the stabilizing point may vary significantly between task sets.

Using $w_{\min}$ with QB-EPDF. Figure 4.19 shows the impact of using an inflated $w_{\min}$ value under the QB-EPDF scenario. Surprisingly, the relationship between $w_{\min} - \mathcal{S}.w_{\phi}$ and the number of computations performed appears to be approximately linear[15] throughout the graph (see Figure 4.19(a)). By extrapolation, the line segment over the x range $[0.00001, 0.1]$ appears to fit the equation $y = 10^{-0.5}\frac{1}{x}$. The cause of this behavior is not immediately obvious; further study is needed to better understand the relationship between these quantities. For larger $w_{\min}$ values, this relationship does not hold. We speculate that the reason for this is that the $w_{\min}$ parameter is set so high that $\phi(\mathcal{S}, L_0) > w_{\min}$ holds initially. When this happens, the invocation terminates after performing only minimal computation and leaves $n = 0$. (In these experiments, $L_{\phi} = L_0$. Hence, the loop in line 5 performs no computation.) As a result, the mean is skewed toward zero. As in the first experiment, the observed samples exhibit a high degree of variance (see Figure 4.19(b)). The strange behavior of the standard deviation range is due solely to the use of log scale. This range is actually symmetric around the mean. Because the means are close to zero for high values of $w_{\min}$, the lower points of the ranges are below zero. Hence, the ranges extend beyond the bottom of the graph.

[15]By linear, we mean only that the graph depicts a straight line. Because the graph is presented in log scale, this does not suggest that the relationship can be expressed as a linear equation.

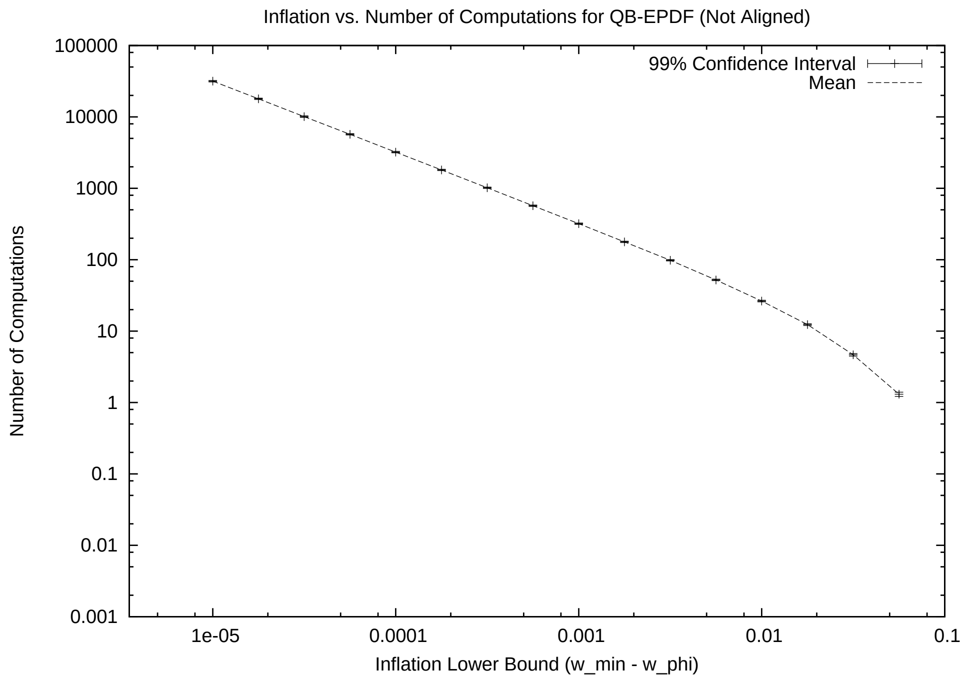


(a) 99% Confidence Interval

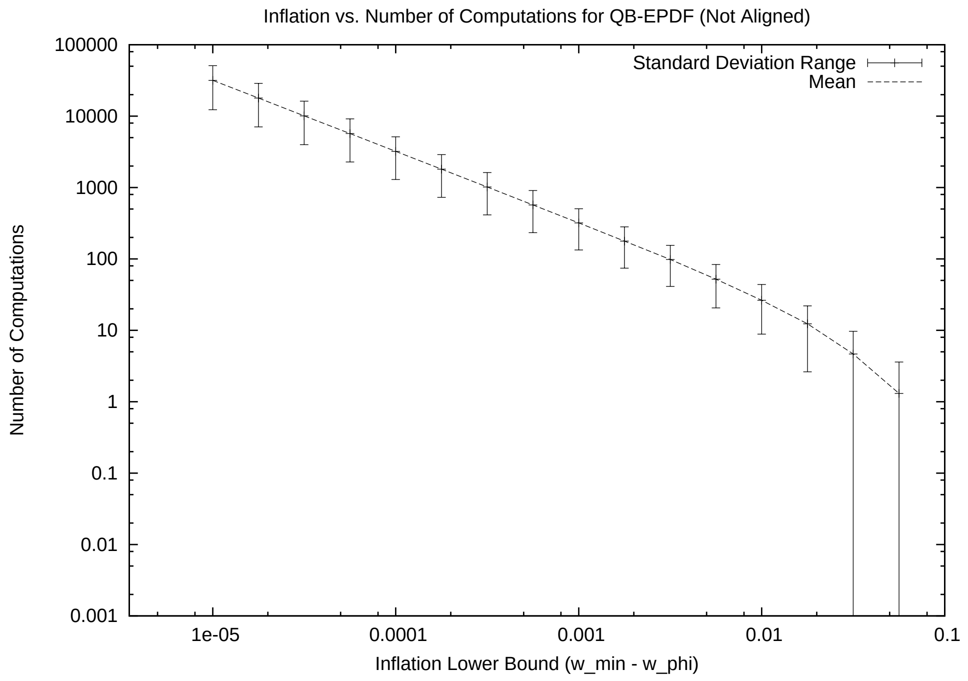


(b) Standard Deviation

Figure 4.18: Plots show how the inflation varies as the computation limit is increased under the FP-EDF scenario. The figure shows **(a)** the sample means and **(b)** the interval defined by one standard deviation.



(a) 99% Confidence Interval



(b) Standard Deviation

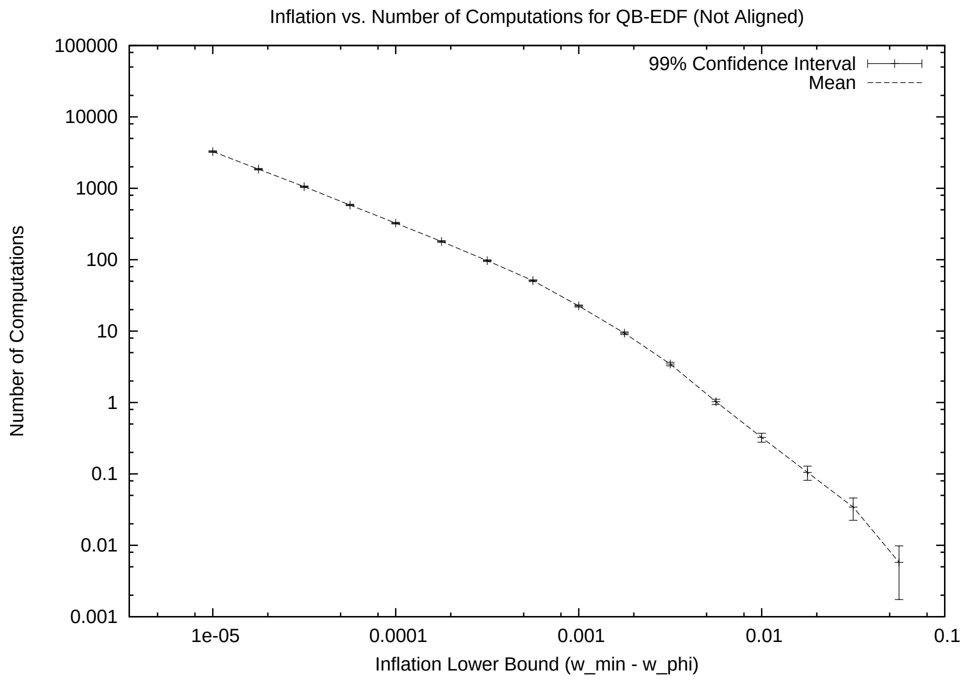
Figure 4.19: Plots show how the number of computations performed varies as the initial inflation (*i.e.*, $w_{\min} - \mathcal{S}.w_{\phi}$) is increased under the QB-EPDF scenario. The figure shows **(a)** the sample means and **(b)** the interval defined by one standard deviation.

Using $w_{\min}$ with QB-EDF. Figure 4.20 shows the impact of using an inflated $w_{\min}$ value under the QB-EDF scenario. Again, the relationship between $w_{\min} - \mathcal{S}.w_{\phi}$ and the number of computations performed appears to become approximately linear for $x < 0.001$ (see Figure 4.20(a)). By extrapolation, the line segment over the x range $[0.00001, 0.001)$ appears to fit the equation $y = 10^{-1.5}\frac{1}{x}$. Notice also that the QB-EDF scenario shows an order of magnitude reduction in computation when compared to the QB-EPDF scenario. Unfortunately, the variance of samples does not show improvement, as can be seen in Figure 4.20(b).

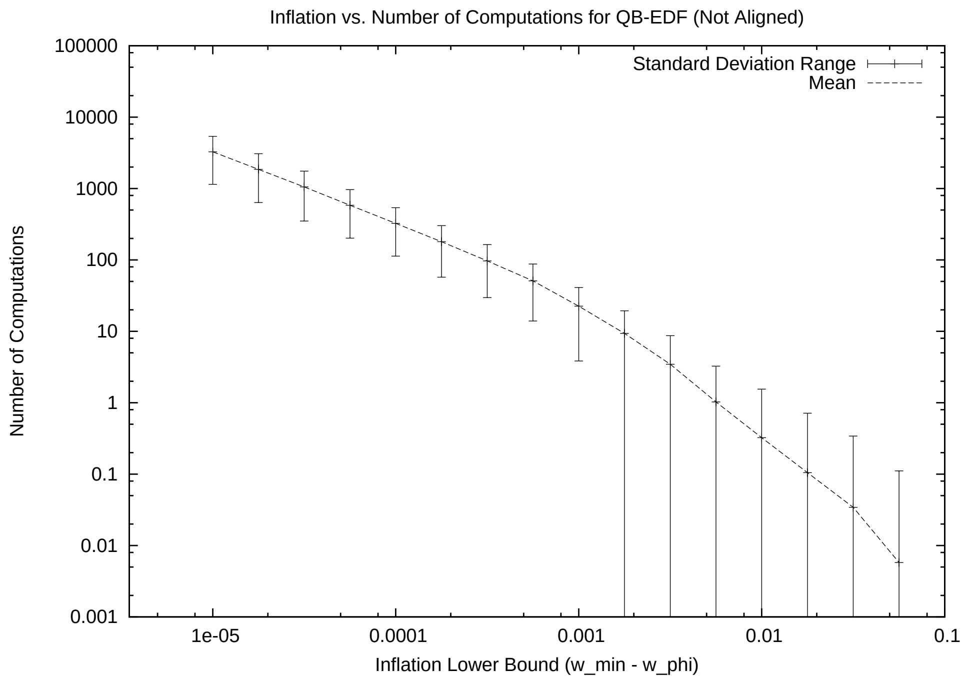
Using $w_{\min}$ with FP-EDF. Figure 4.21 shows the impact of using an inflated $w_{\min}$ value under the FP-EDF scenario. As with previous approaches, Figure 4.21(a) shows an approximately linear relationship over the range $[0.00001, 0.001]$. Again, by extrapolation, this relationship appears to fit the equation $y = 10^{-1}\frac{1}{x}$. Remarkably, the curve depicted in Figure 4.21(a) appears to fall directly between those produced by the quantum-based scenarios. Again, the nature of this relationship is unclear and requires further study. The observed sample variance is also strikingly similar, as shown in Figure 4.21(b).

4.6.3 Conclusions

In this section, we presented the results of two experimental studies that consider the relative performance of the QB-EPDF, QB-EDF, and FP-EDF scenarios. The first study suggests that the FP-EDF scenario is, by far, the most likely to produce the minimum overhead. In addition, this study suggests that the benefit of using aligned jobs, as it relates to reweighting overhead, is quite small on average. When considering how inflation varied across the sample space, several relationships were noted, most of which were not surprising. One notable exception was the inflation fluctuations observed for the QB-EPDF scenario when utilization

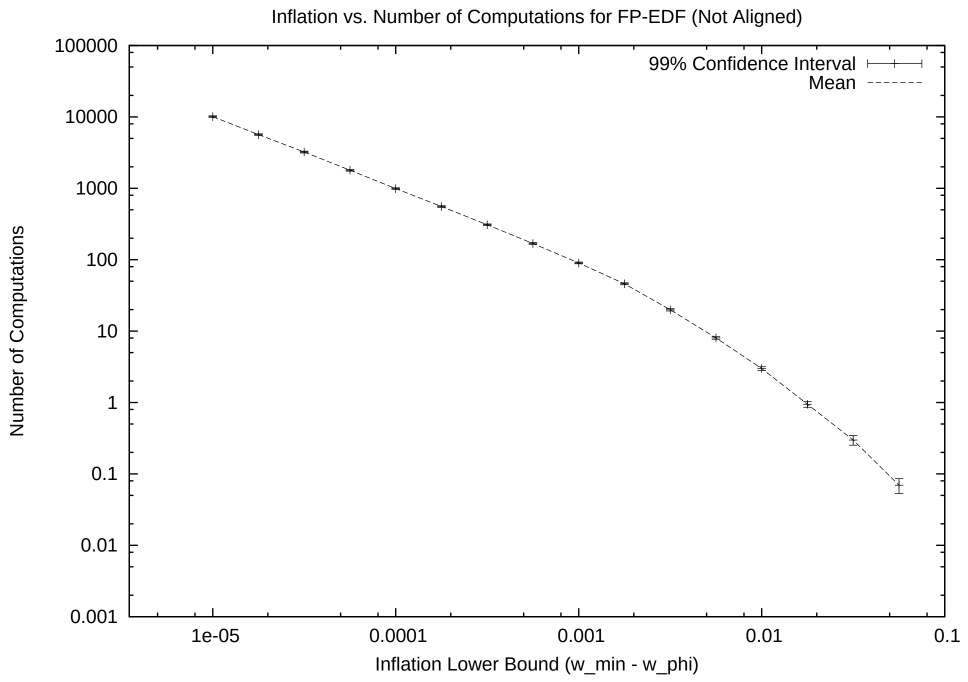


(a) 99% Confidence Interval

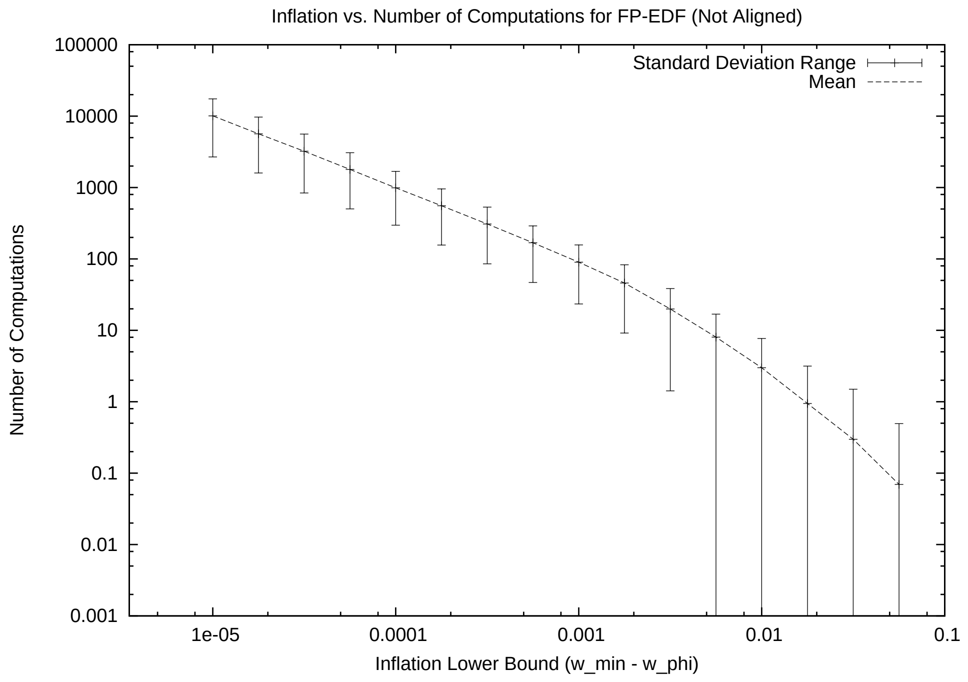


(b) Standard Deviation

Figure 4.20: Plots show how the number of computations performed varies as the initial inflation (*i.e.*, $w_{\min} - \mathcal{S}.w_{\phi}$) is increased under the QB-EDF scenario. The figure shows **(a)** the sample means and **(b)** the interval defined by one standard deviation.



(a) 99% Confidence Interval



(b) Standard Deviation

Figure 4.21: Plots show how the number of computations performed varies as the initial inflation (*i.e.*, $w_{\min} - \mathcal{S}.w_\phi$) is increased under the FP-EDF scenario. The figure shows **(a)** the sample means and **(b)** the interval defined by one standard deviation.

was varied. Further study is needed to determine whether this behavior is a by-product of the experimental methodology used here or rather some dependency between reweighting parameters that is not yet understood.

In the second study, the speed-versus-accuracy trade-off was characterized by two experiments. In the first experiment, the $n_{\max}$ parameter was varied to estimate the point of diminishing returns for each supertasking scenario. For the QB-EPDF, QB-EDF, and FP-EDF scenarios, this point was estimated to be approximately $n_{\max} = 1000$, 32, and 1000, respectively. (However, the FP-EDF scenario shows little improvement after the $n_{\max} = 100$ point.) Unfortunately, the variance of the observed samples suggests that this point is likely to vary significantly between task sets. In the second experiment, the impact of the difference $w_{\min} - \mathcal{S}.w_{\phi}$ on the number of computations performed was estimated. When the difference is at most 0.001, the relationship between $w_{\min} - \mathcal{S}.w_{\phi}$ (denoted x) and the number of computations (denoted y) appears to be estimated with reasonable accuracy by the equation $y = 10^{-k}\frac{1}{x}$, where $k = 0.5$, 1.5, 1 for QB-EPDF, QB-EDF, and FP-EDF, respectively. It is currently unclear why this equation (seemingly) relates these values. Further study is warranted.

4.7 Summary

In this chapter, we have presented a general framework for assigning supertask weights based on the characteristics of the internal scheduling policy and the component tasks. We have also illustrated the use of the framework by considering five example scenarios. To facilitate use of these results, we presented an efficient reweighting algorithm that supports manipulation of the speed-versus-accuracy trade-off through its arguments. We concluded by presenting the results of two experimental studies. The first study compared the overhead experienced when supporting periodic tasks under each of the basic supertasking approaches considered in the chapter and the mapping approach considered in Chapter 3. The second study evaluated the speed-versus-accuracy trade-off by fixing the number of computations and then observing the impact on inflation, and vice versa.

4.8 Related Work

As explained earlier, the concepts underlying supertasking have been applied in several areas. However, the most closely related work at present appears to be the resource partition model of Mok and Feng [60, 44, 24, 50, 51]. This model is intended to provide a system-independent description of a server and the guarantees that it provides. Effectively, this model imposes a second level of indirection between the underlying system and the server. Analysis under this model closely parallels that presented in this chapter (see [60, 44] for examples).

As with CBS (discussed earlier), the primary difference between supertasks and resource partitions is the server task's model. Indeed, a supertask could be expressed as a resource partition. However, assuming the use of Pfair scheduling enables the use of stronger assumptions when reweighting, resulting in lower weights. Interestingly, resource partitions can be easily supported under Pfair scheduling by simply defining a mapping from partition parameters to a supertask weight, as was done earlier with periodic parameters in Chapter 3. Hence, Pfair scheduling can be used to implement open systems that are based on this model.

CHAPTER 5

Lock-free Synchronization [1]

In this and the next chapter, we add support for task synchronization under Pfair scheduling. Synchronization raises two issues. First, it entails additional overhead, which must be taken into account when determining schedulability [4, 9, 53, 54, 55, 59]. Second, blocking prevents tasks from executing at a consistent rate, thereby making rate-based guarantees more difficult to provide. Unfortunately, prior work on real-time synchronization has been directed at uniprocessor systems, or systems implemented using non-rate-based scheduling policies (or both), and thus cannot be directly applied to Pfair-scheduled multiprocessors.

In this chapter, we consider a method of synchronization that is particularly well-suited to rate-based scheduling: lock-free algorithms. Under lock-free synchronization, shared objects are implemented without critical sections or related mechanisms. Instead, operations are optimistically attempted; if an operation fails, then it is simply retried until successful. As a result, tasks never suspend and hence do not waste processor time. (Recall that suspensions can produce processor idling under quantum-based scheduling.) Lock-based synchronization is considered in the next chapter.

[1] This chapter is derived from work previously published in the following paper:
[32] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 111-120, 2002.

This chapter presents two primary contributions. First, we explain how to account for retry overhead when assigning task weights under Pfair scheduling. Second, we demonstrate how quantum-based supertasking can be applied to reduce retry overhead. Recall that a supertask is merely a collection of tasks that are scheduled as a single entity; when a supertask is scheduled, it selects one of its component tasks for execution. In addition to reducing contention for a shared resource, supertasking can also enable to use of less-costly uniprocessor lock-free algorithms (as explained later). We use a case study to illustrate this last benefit.

We begin by briefly describing the concept of lock-free synchronization in Section 5.1. In Section 5.2, we explain how to bound the overhead induced by the use of lock-free algorithms and account for this overhead when assigning task weights, both with and without supertasking. Two implementations of a shared queue are then presented in Section 5.3 to demonstrate the algorithmic benefits that can be obtained by using supertasks. We end the chapter by presenting and experimentally evaluating a simple heuristic for assigning tasks that share lock-free objects to supertasks in Sections 5.4 and 5.5, respectively.

5.1 Lock-free Object Sharing

Lock-free algorithms are an alternative to traditional semaphore-based techniques when implementing software-based shared objects. These algorithms work particularly well for simple objects like buffers, queues, and lists. Lock-free algorithms avoid locking by using *retry loops*.

Figure 5.1 depicts a lock-free enqueue operation with such a loop. In this example, an $\&x$ operation returns the address at which the variable x is stored in memory, while the $*x$ operation returns the data stored at the memory address given by x. The queue is implemented as a singly linked list, where the `next` field acts as the inter-node link.

```
typedef Qtype:
    record data: valtype; next: pointer to Qtype
shared var  Head, Tail: pointer to Qtype
private var old, new: pointer to Qtype; input: valtype;
            addr: pointer to pointer to Qtype
procedure Enqueue(input)
    *new := (input, nil);
    do old := Tail;
        if $old \neq nil$ then addr := &((*old).next)
        else addr := &Head fi
    while ¬CAS2(&Tail, addr, old, nil, new, new)
```

Figure 5.1: Lock-free enqueue

An item is enqueued by using a *two-word compare-and-swap* (`CAS2`) instruction[2] to atomically update both the tail pointer and either the `next` pointer of the last node or the head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the `CAS2` instruction succeeds. An important property of lock-free implementations such as this is that operations may *interfere* with each other. In this example, an interference results when a successful `CAS2` call by one task causes another task's `CAS2` call to fail.

We also consider the use of *wait-free* algorithms, which are stricter forms of lock-free algorithms. Wait-freedom strengthens lock-freedom by requiring that each operation eventually makes progress, *i.e.*, starvation must be avoided. To provide such a guarantee, wait-free algorithms must consist only of code segments with a bounded number of loop iterations.

5.1.1 Requirements of Analysis

When lock-free objects are used in real-time systems, bounds on loop retries must be computed for scheduling analysis. On uniprocessors, aspects of priority schedulers can be taken into

[2]The first two parameters of `CAS2` specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed. Although `CAS2` is uncommon, it makes for a simple example here.

account to determine such bounds [4]. On real-time multiprocessors, lock-free algorithms have been viewed as being impractical, because deducing bounds on retries due to interferences *across* processors[3] is difficult. However, as we later show, the predictability of quantum-based scheduling can be exploited to help bound interferences more tightly than is possible under fully preemptive scheduling.

5.1.2 Limitations

Although lock-free algorithms are ideal for rate-based scheduling, they are not always appropriate. Specifically, lock-free techniques tend to produce efficient implementations only for simple objects, such as queues and buffers, and often rely on the use of strong synchronization primitives, such as the `CAS2` instruction described above. For complex objects, overhead (both time *and* space) can be prohibitive. In such cases, locking synchronization is more suitable. Furthermore, lock-free synchronization can only be applied to software-based shared objects. Locking *must* be used to synchronize accesses to external devices. Hence, support for locking synchronization (discussed in the next chapter) is a necessity in many systems.

5.2 Analysis

We now describe how to account for lock-free synchronization when performing schedulability analysis. To the best of our knowledge, this is the first work to consider the use of lock-free objects in real-time multiprocessors. The efficient use of lock-free algorithms is made possible by the quantum-based model underlying Pfair scheduling, as explained below.

[3]An interference is said to occur across processors if the interfering task resides on a different processor than the task experiencing the failed retry-loop attempt.

5.2.1 Definitions and Assumptions

Let Γ denote the set of lock-free shared objects. We do *not* assume that only lock-free synchronization is used by the tasks. Instead, our analysis in this section consists of showing how each lock-free execution phase $\mathbf{C}(\ell, e)$, where ℓ is a lock-free shared object, can be treated as an equivalent phase $\mathbf{C}(\emptyset, e')$ for the purpose of applying the mapping lemmas from Chapter 3. This conversion is accomplished by defining e' so that it includes not only the actual execution requirement of the phase, but also the worst-case retry overhead. The analysis of locking synchronization presented in the next chapter uses this same approach.

We define the following additional notation to simplify the statement of results:

- Let $A(T, \ell)$ be the maximum number of accesses to object ℓ in a single quantum.[4] When using supertasks, let $A(\mathcal{S}, \ell) = \max\{\ A(T, \ell) \mid T \in \mathcal{S}\ \}$.

- Let $N(\ell)$ denote $\min(M, |\{\ T \mid A(T, \ell) > 0\ \}|)$ when not using supertasks, and let it denote $\min(M, |\{\ \mathcal{S} \mid A(\mathcal{S}, \ell) > 0\ \}|)$ when using supertasks. Informally, $N(\ell)$ is the maximum number of processors that may try to access ℓ in a single slot.

- Let $e_B(\ell)$ and $e_R(\ell)$ denote the base and retry overhead of an operation on ℓ, *i.e.*, an operation that retries k times consumes at most $e_B(\ell) + k \cdot e_R(\ell)$ units of processor time.

- Let $e_B^{[m]}(\ell)$ and $e_R^{[m]}(\ell)$ denote the overheads of an m-processor implementation of ℓ. In the analysis considered below, we assume that $e_B(\ell) = e_B^{[N(\ell)]}(\ell)$ and $e_R(\ell) = e_R^{[N(\ell)]}(\ell)$

To simplify the analysis, we consider only a single bound for all operations on a given object ℓ. The analysis presented here is easily extended to support a separate bound for each operation.

We make the following assumptions regarding retries and interferences.

[4]Timing analysis is needed to obtain these bounds.

Interference Assumption (IA): Any pair of concurrent accesses to the same object may potentially interfere with each other.

Retry Assumption (RA): A retry can be caused *only* by the completion of an operation on the same object. (Hence, the number of retries is at most the number of concurrent accesses to the object.)

Preemption Assumption (PA): Each operation spans at most two quanta and hence can be preempted at most once.

(PA) stems from the observation that lock-free loop iterations are typically very short relative to Q [3]. Determining whether this assumption holds is straightforward.

The remaining notations are defined as shown below.

- Let $I(T, \ell) = \mathsf{maxsum}_{M-1}\{\ A(U, \ell) \mid U \in \tau/\{T\}\ \}$. Informally, $I(T, \ell)$ is the maximum number of retries that T can experience in a single quantum without supertasking.
- Let $I(\mathcal{S}, \ell) = \mathsf{maxsum}_{M-1}\{\ A(\mathcal{T}, \ell) \mid \mathcal{T} \in \pi/\{\mathcal{S}\}\ \}$. Informally, $I(\mathcal{S}, \ell)$ is the maximum number of retries that $T \in \mathcal{S}$ can experience in a single quantum.

5.2.2 Analysis without Supertasks

In this subsection, we consider the conversion of phases when supertasks are not used.

Theorem 5.1 *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$ where ℓ is a lock-free object, $T^{[i]}$ is equivalent to*

$$\mathbf{C}(\emptyset, e_B(\ell) + (2 \cdot I(T, \ell) + 1) \cdot e_R(\ell))$$

when supertasking is not used.

Proof. By (PA), a single access to ℓ is preempted at most once before completion. In the worst case, this access experiences the maximum number of retries during the quantum preceding the preemption and during the quantum in which the access completes. By (RA), the worst-case number of retries in a single quantum is bounded by the number of concurrent accesses to ℓ within that quantum. Since there are at most $M - 1$ tasks executing in parallel with T's job and each concurrent task U makes at most $A(U, \ell)$ accesses to ℓ within the quantum, it follows that $I(T, \ell)$ is an upper bound on the number of retries that T will perform in each quantum due to parallel interference. (Note that (PA) implies that $I(T, \ell) \cdot e_R(\ell)$ is much smaller than Q.) Therefore, at most $2 \cdot I(T, \ell) + 1$ retries are performed before the access completes. (At most $I(T, \ell)$ retries are needed for each quantum in which T executes. In addition, if T is preempted during an attempt, then one additional retry may be needed due to accesses performed while T was not executing.) Therefore, $e_B(\ell) + (2 \cdot I(T, \ell) + 1) \cdot e_R(\ell)$ is an upper bound on the total processor time required to complete phase $T^{[i]}$. $\square$

5.2.3 Analysis with Supertasks

Supertasking can improve performance in the following two ways.

- A supertask can prevent potentially-interfering tasks from executing in parallel. By doing so, the number of retries needed to complete an operation can be reduced.
- By (PA), if all tasks that share an object are component tasks of the same supertask, then a retry is necessary only if the access is preempted. As a result, a simpler wait-free algorithm can often be used.

When supertasks are used, the worst-case number of interferences experienced in a single quantum changes from $I(T, \ell)$ to $I(\mathcal{S}, \ell)$, where $T \in \mathcal{S}$. In addition, algorithmic gains may be obtained by a reduction in $N(\ell)$, which impacts $e_B(\ell)$ and $e_R(\ell)$. As observed earlier, when $N(\ell) = 1$, ℓ can be implemented using a uniprocessor algorithm. We demonstrate the benefit of such an implementation later with a case study.

Theorem 5.2 *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$ where ℓ is a lock-free object, $T^{[i]}$ is equivalent to*

$$\mathbf{C}(\emptyset, e_B(\ell) + (2 \cdot I(\mathcal{S}, \ell) + 1) \cdot e_R(\ell))$$

when using quantum-based supertasks.

Proof. The proof is virtually identical to that of Theorem 5.1. However, since T can be scheduled in parallel with at most one component task from each of the other supertasks, the worst-case number of retries that can occur during each quantum is $I(\mathcal{S}, \ell)$. □

5.2.4 Examples

We now use the above theorems to derive weights for a sample task set in which all shared objects use lock-free implementations. Let $\mathcal{J}(T, \ell) = \left|\left\{ T^{[i]} \;\middle|\; T^{[i]} \in \mathcal{C}(\ell) \right\}\right|$. Informally, $\mathcal{J}(T, \ell)$ is the number of accesses to lock-free object ℓ made by each job of T.

To simplify the example, assume that all tasks are synchronous periodic tasks with periods equal to relative deadlines that satisfy the conditions given in Lemma 3.4. Also, let $Q = 1$. Under these assumptions, Lemma 3.4 can be used to compute weights for all tasks in the task set after converting the lock-free accesses. Consider the example four-processor task set shown in Figure 5.2(a)–(b). The value given in the e column is the total execution requirement of all execution phases that do *not* access lock-free objects. This example considers only two implementations for each object ℓ: a uniprocessor implementation (used when $N(\ell) = 1$) and

T	e	p, d	$\mathcal{A}(\ell_1)$	$A(\ell_1)$	$\mathcal{A}(\ell_2)$	$A(\ell_2)$
T_1	10	100	2	1	0	0
T_2	15	100	1	1	0	0
T_3	15	100	0	0	1	1
T_4	25	100	0	0	2	2
T_5	25	200	2	1	1	1
T_6	30	200	1	1	0	0
T_7	20	200	0	0	1	1
T_8	40	300	3	2	0	0
T_9	65	500	0	0	2	1
T_{10}	50	700	5	2	12	3

(a)

ℓ	$e_B^{[1]}(\ell)$	$e_R^{[1]}(\ell)$	$e_B^{[M]}(\ell)$	$e_R^{[M]}(\ell)$
ℓ_1	0.012	0.08	0.05	0.16
ℓ_2	0.005	0.075	0.017	0.11

(b)

T	$I(\ell_1)$	$I(\ell_2)$	$\lambda(\ell_1)$	$\lambda(\ell_2)$	$\Lambda(\ell_1)$	$\Lambda(\ell_2)$	w
T_1	5	6	1.81	1.447	3.62	0.0	14/100
T_2	5	6	1.81	1.447	1.81	0.0	17/100
T_3	5	6	1.81	1.447	0.0	1.447	17/100
T_4	5	5	1.81	1.227	0.0	2.454	28/100
T_5	5	6	1.81	1.447	3.62	1.447	30/200
T_6	5	6	1.81	1.447	1.81	0.0	32/200
T_7	5	6	1.81	1.447	0.0	1.447	22/200
T_8	4	6	1.49	1.447	4.47	0.0	45/300
T_9	5	6	1.81	1.447	0.0	2.894	68/500
T_{10}	4	4	1.49	1.007	7.45	12.084	70/700

(c)

Figure 5.2: **(a)** A sample task set τ on $M = 4$ processors with two lock-free objects ℓ_1 and ℓ_2. All parameters are expressed in units of quanta. **(b)** Parameters of ℓ_1 and ℓ_2. **(c)** Summary of all values computed when applying Theorem 5.1 and Lemma 3.4. The λ and Λ values are temporary values that represent the results of particular stages in the calculation.

a multiprocessor implementation (used when $N(\ell) > 1$). We use the above theorems to select task weights for this set below.

Without supertasks. Figure 5.2(c) summarizes the values that are computed when applying Theorem 5.1 and Lemma 3.4. We explain each column in turn by stepping through the computation of T_{10}'s weight. First, we must bound the number of retries for each of T_{10}'s object accesses. Consider ℓ_1. Applying the definition of $I(T_{10}, \ell_1)$ yields $I(T_{10}, \ell_1) = \mathsf{maxsum}_{M-1}\{\ A(U, \ell_1) \mid U \in \tau/\{T_{10}\}\ \} = \mathsf{maxsum}_3\{2, 1, 1, 1, 1, 0, 0, 0, 0\} = 4$. This is shown in the column labeled $I(\ell_1)$ in Figure 5.2(c). Next, we compute the worst-case execution cost of a single access to object ℓ_1 by T_{10}, labeled $\lambda(\ell_1)$ in the table. Since $N(\ell_1) = \min(4, 6) = 4$, $e_B(\ell_1) = e_B^{[M]}(\ell_1) = 0.05$ and $e_R(\ell_1) = e_R^{[M]}(\ell_1) = 0.16$. From these values and the expression $\lambda_{T_{10}}(\ell_1) = e_B(\ell_1) + (2 \cdot I(T_{10}, \ell_1) + 1) \cdot e_R(\ell_1)$, we get $\lambda_{T_{10}}(\ell_1) = 0.05 + (2 \cdot 4 + 1) \cdot 0.16 = 1.49$. We can now determine the total execution overhead of T_{10}'s accesses to ℓ_1 in a single job, labeled $\Lambda(\ell_1)$ in the table, by simply multiplying the value in the $\lambda(\ell_1)$ column by that in the $\mathcal{A}(\ell_1)$ column. Doing so yields $\Lambda_{T_{10}}(\ell_1) = 5 \cdot 1.49 = 7.45$. In a similar manner, it can be shown that $\Lambda_{T_{10}}(\ell_2) = 12.084$. Hence, the execution requirement of an entire job of T_{10} is upper-bounded by $50 + 7.45 + 12.084 = 69.534$. Applying Lemma 3.4 to T_{10} then yields the weight $\frac{\lceil 69.534 \rceil}{700} = \frac{70}{700}$, as shown in the last column of the table. Since the calculated weights of all tasks are upper-bounded by unity and their sum is upper-bounded by M, (2.4) implies that this task set is schedulable under any optimal Pfair scheduling policy.

With supertasks. Figure 5.3(a) shows one possible partitioning of the example task set from Figure 5.2(a). This partitioning assigns all tasks accessing ℓ_2 to $\mathcal{S}_2$ and then assigns all remaining tasks to $\mathcal{S}_1$. By doing so, we permit the use of the more efficient uniprocessor algorithm for ℓ_2 in place of the multiprocessor version. Therefore, $e_B(\ell_2) = e_B^{[1]}(\ell_2) = 0.005$ and $e_R(\ell_2) = e_R^{[1]}(\ell_1) = 0.075$. However, notice that the multiprocessor version of ℓ_1's algorithm must still be used. Hence, $e_B(\ell_1) = e_B^{[M]}(\ell_1) = 0.05$ and $e_R(\ell_1) = e_R^{[M]}(\ell_1) = 0.16$. Since the computations in Theorem 5.2 are similar to those already demonstrated, we do not explain them here. The values resulting from each step can be seen in Figure 5.3(b). Notice that

$\mathcal{S}$	**Components**	$A(\ell_1)$	$A(\ell_2)$	$I(\ell_1)$	$I(\ell_2)$
$\mathcal{S}_1$	T_1, T_2, T_6, T_8	2	0	2	3
$\mathcal{S}_2$	T_3, T_4, T_5, T_7, T_9, T_{10}	2	3	2	0

(a)

T	$I(\ell_1)$	$I(\ell_2)$	$\lambda(\ell_1)$	$\lambda(\ell_2)$	$\Lambda(\ell_1)$	$\Lambda(\ell_2)$	w
T_1	2	3	0.85	0.53	1.70	0.0	12/100
T_2	2	3	0.85	0.53	0.85	0.0	16/100
T_3	2	0	0.85	0.08	0.0	0.08	16/100
T_4	2	0	0.85	0.08	0.0	0.16	26/100
T_5	2	0	0.85	0.08	1.70	0.08	27/200
T_6	2	3	0.85	0.53	0.85	0.0	31/200
T_7	2	0	0.85	0.08	0.0	0.08	21/200
T_8	2	3	0.85	0.53	2.55	0.0	43/300
T_9	2	0	0.85	0.08	0.0	0.16	66/500
T_{10}	2	0	0.85	0.08	4.25	0.96	56/700

(b)

Figure 5.3: **(a)** Parameters for the partitioning $\pi = \{\{T_1, T_2, T_6, T_8\}, \{T_3, T_4, T_5, T_7, T_9, T_{10}\}\}$. **(b)** Summary of all values computed when applying Theorem 5.2 and Lemma 3.4.

many of the task weights are smaller than in the previous example. In addition, the total weight has reduced from 1.57 to 1.45. Unfortunately, as discussed earlier in Chapter 4, the use of supertasks produces some loss, which counters this improvement.

Remarks. In a later section, we present experimental results that suggest that supertasking often improves schedulability, even with reweighting overhead. However, in a system with relatively few processors, the benefit of supertasking is often negated by its cost. This is because of the limited parallelism in such systems. Recall that the most significant component of the retry overhead is caused by the parallel execution of tasks. Hence, this overhead scales with M and may be insignificant for small values of M. For this reason, it is always advisable to calculate the weights of task sets both with and without supertasks in order to determine whether supertasking is beneficial.

5.3 Case Study: Queues

In this section, we illustrate the algorithmic benefits that can be obtained by enabling the use of a uniprocessor object implementation instead of a multiprocessor implementation. This is accomplished by presenting two lock-free queue implementations and comparing the number of primitive operations performed by each.

5.3.1 Variable Tagging

Lock-free algorithms often use the *compare-and-swap* (`CAS`) primitive to update shared variables. `CAS` is similar to the `CAS2` primitive (used in Figure 5.1), but accesses only one memory location. Unlike `CAS2`, `CAS` (and similar primitives) are fairly common. The primary benefit of `CAS` is that it can avoid the *late-write* problem. This problem occurs when a task T's next instruction updates a shared variable, but another update on that variable occurs first. If T's update succeeds, then the result of the earlier update will be overwritten.

To illustrate why late writes can be undesirable, consider a simple shared counter that supports only an increment operation. To perform the increment, a task must make a local copy of the current value, increment that local value, and then overwrite the shared value with the local value. Now, suppose two tasks simultaneously invoke the increment operation and read the value 2 from the shared variable, resulting in a local value of 3. Once the first operation updates the counter, the second operation becomes invalid due to the fact that its update is based on an out-of-date counter value (*i.e.*, the increment is based on the counter value 2, but the current value is 3). Hence, the second update, which is called an *enabled late* *write*, must fail to ensure correctness.

By using `CAS`, we can ensure that the value being overwritten is equal to the value read by the task. Such a guarantee would be sufficient for the above example because the shared variable's value monotonically increases;[5] hence, each value is uniquely associated with only one successful increment operation. Unfortunately, when multiple operations may successfully write the same value, simply comparing the current value to that read earlier is not sufficient to determine whether other updates occurred after that read.

The `CAS` primitive can be used to detect modification by subdividing a word of memory into two sets of bits. One set stores the current value of the variable, while the second stores a counter like that discussed above. This counter field is called the variable's *tag*. Each time an update is attempted, an increment operation is applied to the tag. By the same reasoning given above, each value of the tag is uniquely associated with a single operation. Hence, updates of the variable can be detected by checking the tag. Since the tag and value comprise a single word of memory, a `CAS` operation can be used to atomically compare and update both values simultaneously. Both implementations presented here use tagged variables.

template *tagged*(T): **record** *tag*: **integer**; *value*: T

Our algorithms use the template definition shown above to define tagged variables. As stated above, each tagged variable is assumed to require only one memory word. Though accounting for the bounded range of the tags is an important issue, we ignore this complication here since such advanced issues are outside the scope of this discussion. Instead, we make the simplifying assumption that tag ranges are unbounded.

[5] We ignore the possibility of rollover.

5.3.2 Algorithms

Our multiprocessor and uniprocessor queue implementations are shown in Figures 5.5 and 5.6, respectively. (A correctness proof for the multiprocessor version can be found in Appendix A.) Both implementations are based on a linked list, in which each list node consists of a value field, *value*, and a tagged pointer to the next list node, *next*. In addition, the tagged pointers *Head* and *Tail* record the start and end, respectively, of the list. To allow safe concurrent enqueue and dequeue operations, the linked list contains a *dummy* head node. The structure of the queue is illustrated in Figure 5.4.

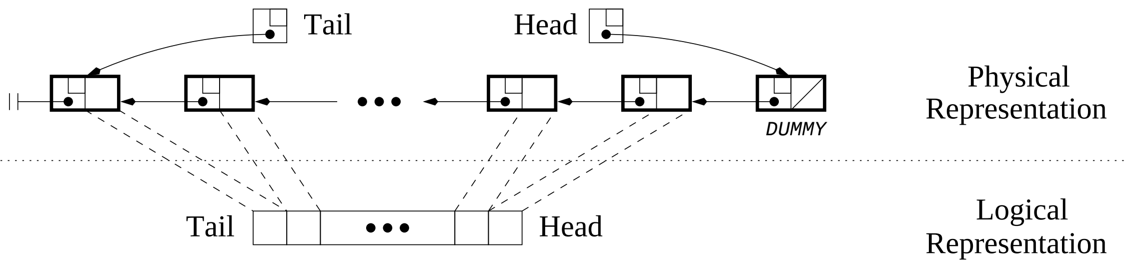


Figure 5.4: Physical and logical representation of the shared queue.

The *next* and *Tail* pointers are tagged to support multiple enqueuers (writers). For the single-enqueuer case, the enqueue procedure simplifies to `Enqueue_1W`, shown below `Enqueue_MW` in Figure 5.5, and these tag fields can be safely removed. Similarly, the tag field in *Head* supports multiple dequeuers (readers) and can be safely removed in the single-dequeuer case. In this case, `Dequeue_MR` simplifies to the procedure `Dequeue_1R` in Figure 5.5. `Enqueue_1W` and `Dequeue_1R` are common to both implementations, and thus are not repeated in Figure 5.6.

Multiprocessor algorithm. In the multiprocessor implementation, an enqueue operation invokes `Enqueue_MW` and passes a node (*in*) containing the value to be enqueued. The *next* field is initialized in lines 1–2 and the *done* flag cleared in line 3. The *next* field of the last

```
template tagged(T):
    record
        tag: integer;
        value: T

typedef node:
    record
        value: element;
        next: tagged(pointer to node)

private var
    x, h, t: tagged(pointer to node);
    done: boolean; out: element;
    in: pointer to node

procedure Enqueue_MW(in)
1:  x := *(in).next;
2:  *(in).next := (x.tag+1,nil);
    do
3:      done := false;
4:      t := Tail;
5:      x := *(t.value).next;
6:      if t = Tail then
7:          done := CAS(&(*(t.value).next),
                     (x.tag,nil),(x.tag+1,in));
8:          x := *(t.value).next;
9:          CAS(&Tail, t, (t.tag+1,x.value))
        fi
10: while $\neg$done

procedure Enqueue_1W(in)
11: x := *(in).next;
12: *(in).next := (x.tag+1,nil);
13: t := Tail;
14: x := *(t.value).next;
15: *(t.value).next := (x.tag+1,in);
16: Tail := (t.tag+1,in)
```

```
shared var
    Head, Tail: tagged(pointer to node)

procedure Dequeue_MR()
    returns pointer to node
    do
17:     done := false;
18:     h := Head;
19:     if h.value = Tail.value then
20:         if h = Head then
21:             return nil
            fi
        else
22:         x := *(h.value).next;
23:         if x.value $\neq$ nil then
24:             out := *(x.value).value;
25:             done := CAS(&Head, h,
                        (h.tag+1,x.value))
            fi
        fi
26: while $\neg$done;
27: *(h.value) := (out,(x.tag+1,nil));
28: return h.value

procedure Dequeue_1R()
    returns pointer to node
29: h := Head;
30: if h.value = Tail.value then
31:     return nil
    else
32:     x := *(h.value).next;
33:     out := *(x.value).value;
34:     Head := (h.tag+1,x.value);
35:     *(h.value) := (out,(x.tag+1,nil));
36:     return h.value
    fi
```

Figure 5.5: Multiprocessor (lock-free) shared queue.

node is then read in lines 4–5. If the comparison at line 6 fails, then another enqueue has completed and a retry occurs. Otherwise, an update of the *next* field is attempted (line 7). If successful, then the new node has been chained onto the end of the list and it remains only to update *Tail*. This is done in lines 8–9. If the `CAS` at line 7 is not successful, then another node, call it X, has already been chained onto the end of the list by another task, in which case the operation must be retried. Lines 8–9 ensure that *Tail* is correctly updated before the retry is performed. (The task enqueuing X may not have updated *Tail* yet.)

The `Dequeue_MR` procedure returns either a pointer to a node that contains the dequeued value or *nil* to signify an empty queue. Since *Head* always points to a dummy node, the node *following* the dummy node, if one exists, contains the value at the head of the queue. Thus, `Dequeue_MR` seeks to remove the dummy head node and return the data stored in the node *after* it. This latter node then becomes the new dummy head node. The operation begins by initializing the *done* flag at line 17 and then reading address of the dummy node at line 18. This address is compared to *Tail* at line 19. If equal, then either the list is empty or the node referenced by h has been dequeued and re-enqueued by other concurrent operations. In the latter case, $h \neq Head$ must hold due to tagging. Thus, line 20 correctly distinguishes between these possibilities. If the test at line 19 is not successful, then either an interference has occurred, or the list is non-empty. In either case, an attempt is made to dequeue the head node in lines 22-25. (If an interference has indeed occurred, then this dequeue attempt will fail. Attempting the operation is the simplest way to detect the interference.) If $x.value$ is *nil* at line 23, then an interference has occurred, and the operation is retried. In line 24, the data value in the node after the dummy node is read, as explained above. In line 25, an attempt is made to advance the *Head* pointer past the (old) dummy node. If this attempt fails, then the operation is retried. (If the `CAS` at line 25 succeeds, then no other enqueue could have completed between lines 18 and 25.) Line 27 simply stores the dequeued value into the removed node so that it can be returned at line 28.

Uniprocessor algorithm. The uniprocessor wait-free queue implementation is obtained

```
template tagged(T):
    record
        tag: integer;
        value: T

procedure Enqueue_MW(in)
1:  pm := false;
2:  x := *(in).next;
3:  *(in).next := (x.tag+1,nil);
4:  t := Tail;
5:  x := *(t.value).next;
6:  if x.value ≠ nil then
7:      if CAS(&Tail, t,
                (t.tag+1,x.value)) then
8:          t := (t.tag+1,x.value);
9:          x := *(t.value).next;
10:         pm := (x.value ≠ nil)
        else
11:         pm := true
        fi
    fi;
12: if ¬pm ∧ t = Tail then
13:     if CAS(&(*(t.value).next),x,
                (x.tag+1,in)) then
14:         CAS(&Tail, t, (t.tag+1,in))
        else
15:         pm := true
        fi
    fi;
16: if pm then
17:     t := Tail;
18:     x := *(t.value).next;
19:     if x.value ≠ nil then
20:         Tail := (t.tag+1,x.value);
21:         t := (t.tag+1,x.value);
22:         x := *(t.value).next
        fi;
23:     *(t.value).next := (x.tag+1,in);
24:     Tail := (t.tag+1,in)
    fi
```

```
typedef node:
    record
        value: element;
        next: tagged(pointer to node)

shared var
    Head, Tail: tagged(pointer to node)

private var
    x, h, t: tagged(pointer to node);
    in: pointer to node; out: element;
    pm: boolean

procedure Dequeue_MR()
    returns pointer to node
25: h := Head;
26: if h.value = Tail.value then
27:     if h = Head then
28:         return nil
        fi
    else
29:     x := *(h.value).next;
30:     if x.value ≠ nil then
31:         out := *(x.value).value;
32:         if CAS(&Head, h,
                    (h.tag+1,x.value)) then
33:             *(h.value) := (out,(x.tag+1,nil));
34:             return h.value
            fi
        fi
    fi;
35: h := Head;
36: if h.value = Tail.value then
37:     return nil
    else
38:     x := *(h.value).next;
39:     out := *(x.value).value;
40:     Head := (h.tag+1,x.value);
41:     *(h.value) := (out,(x.tag+1,nil));
42:     return h.value
    fi
```

Figure 5.6: Uniprocessor (wait-free) shared queue.

by unrolling the loop in the multiprocessor version a constant number of times and then simplifying the code listing. The result is shown in Figure 5.6.

The loop in `Enqueue_MW` (Figure 5.5) is unrolled three times to produce the procedure shown in Figure 5.6. The need for three loop iterations is illustrated by task B in Figure 5.7. In this scenario, task A begins an operation close to the slot boundary and is preempted between lines 8 and 9. As a result, it leaves its node linked to the tail of the queue (see the t_A state shown in the lower portion of Figure 5.7). Suppose no other operations occur until B initiates its operation. Because the *next* pointer in the node referenced by *Tail* is not *nil* (*i.e.*, it references A's node), the first `CAS` call in line 7 fails and line 9 updates *Tail* to reference A's node. B is then preempted immediately before line 7 during the second iteration, and some other task (C) performs an operation while B is preempted. In this case, B's second execution of line 7 also fails (when B resumes) because the information stored in x and t is out-of-date. Hence, a third iteration is performed. This last iteration is guaranteed to succeed. It follows that three loop iterations occur in the worst case.[6]

In Figure 5.6, the first loop iteration, which checks for a partially completed operation, produces lines 6–11. The second iteration, which attempts to perform the enqueue operation, produces lines 12–15. If preemption occurs during either of these two "iterations," then (PA) ensures that the next iteration is effective non-preemptable. Therefore, the third iteration can be simplified, as shown in lines 16–24.

By (PA), the multiprocessor version of `Dequeue_MR` iterates at most twice on a uniprocessor. The first loop iteration corresponds to lines 25–34 in Figure 5.6. If the operation is preempted, then a second iteration is performed, which corresponds to lines 35–42.

[6]Notice that only one of these iterations was caused by a preemption. Thus, although there are three loop iterations, there is only one retry. Hence, this scenario is not a violation of (PA).

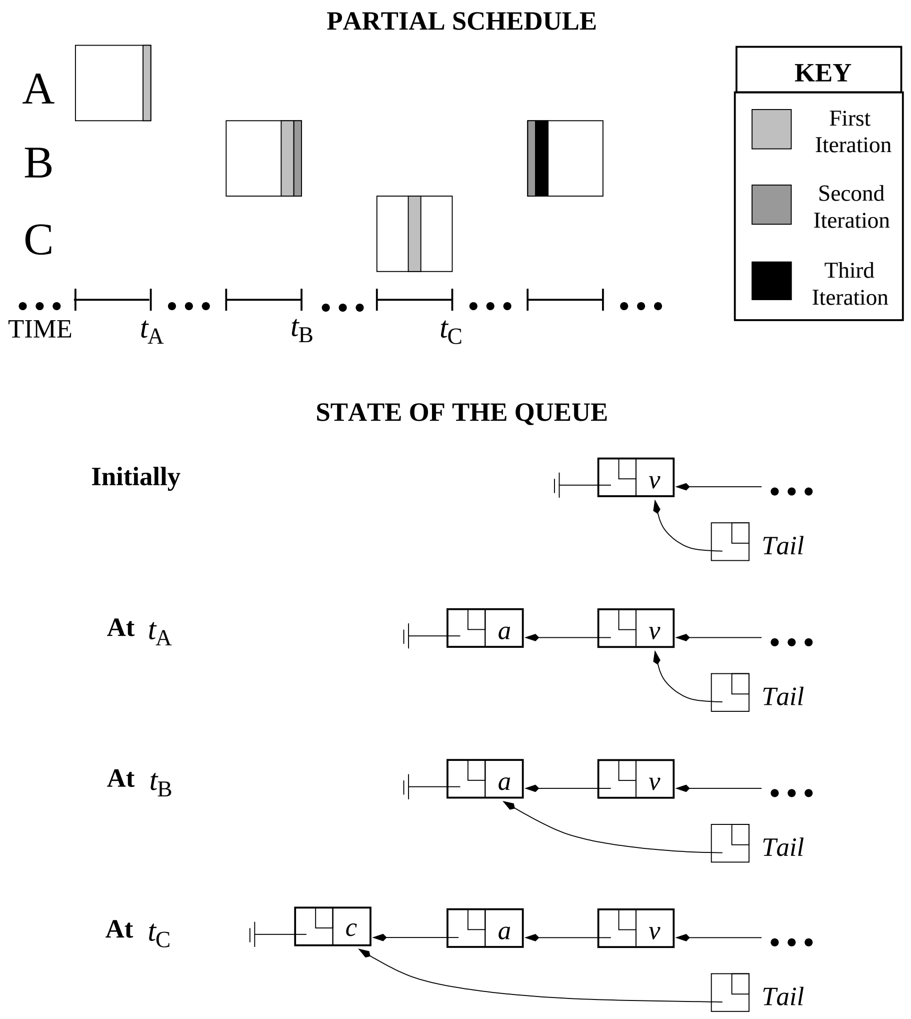


Figure 5.7: Illustration of the worst-case scenario when using the multiprocessor lock-free queue implementation on a quantum-based uniprocessor. This same worst case applies to the use of this implementation within a supertask.

5.3.3 Comparison

One simple method of comparing the relative efficiency of these algorithms is to count the number of shared-memory reads and writes (denoted R and W, respectively) and the number of `CAS` invocations (denoted CAS) that occur in the worst case. This comparison must necessarily be made on a uniprocessor. For the multiprocessor version of `Enqueue_MW`, the

M	Procedure	Path	R	W	CAS	$1\times R$	$1\times W$	$3.5\times CAS$	Total
> 1	`Enqueue_MW`		13	1	6	13	1	21	35
> 1	`Dequeue_MR`		10	2	2	10	2	7	19
1	`Enqueue_MW`	preempted	8	4	2	8	4	7	19
1	`Enqueue_MW`	not preempted	5	1	3	5	1	10.5	16.5
1	`Dequeue_MR`		8	3	1	8	3	3.5	14.5

Figure 5.8: Shared-memory instruction counts along worst-case code paths for both queue algorithms.

worst-case path under uniprocessor execution includes *three* loop iterations, with six `CAS` calls being made in lines 7 and 9. In addition, lines 1–2 contribute one read and write, and each loop iteration contributes four reads. Therefore, in the worst case, the multiprocessor `Enqueue_MW` algorithm has $R = 13$, $W = 1$, and $CAS = 6$.

A similar analysis can be applied the other procedures with the exception of `Enqueue_MW` in Figure 5.6. It is unclear which code path produces the worst-case behavior in this procedure. For this reason, we consider two paths through the code. Figure 5.8 summarizes the instruction counts along all considered paths.

The cost of synchronization primitives. Synchronization primitives usually require more cycles than uncached reads and writes. For example, LaMarca [39] noted that the *load-linked/store-conditional* instruction pair (which provides functionality similar to `CAS`) on a DEC 3000-400 with a 130 MHz Alpha 2 21064 CPU requires approximately 3.5 times the number of cycles as an uncached shared-memory read. Though it is not a modern processor, this and other older architectures are still being used today, particularly in embedded systems.

Comparison. An admittedly simple method for approximating the relative performance of these algorithms is to compare the weighted sums of the previous instruction counts based on LaMarca's observations. Though this comparison is far from exact, it nevertheless provides some insight into the relationship between these implementations. Figure 5.8 shows the resulting weighted sums produced by the worst-case code paths. These sums suggest that using the uniprocessor algorithm provides an improvement in `Enqueue_MW` and `Dequeue_MR` of around 84% ($\frac{35}{19} \approx 1.84$) and 31% ($\frac{19}{14.5} \approx 1.31$), respectively.

Improvement trends. For simple lock-free operations that update only one variable, a uniprocessor implementation may provide only a marginal improvement. However, for more complex operations that update multiple variables, significant improvement is likely. This is demonstrated by `Enqueue_MW`, which shows significant improvement despite the fact that it updates only two variables.

Operations that perform multiple updates can be greatly simplified by using a *multi-word compare-and-swap* (`MWCAS`). This primitive generalizes both `CAS` and `CAS2` by making the number of words involved in the operation an argument. Though it is impractical to provide a `MWCAS` primitive in hardware, it *can* be efficiently implemented in software on a uniprocessor [3]. In contrast, no efficient implementation is known for multiprocessors. For this reason, the class of objects that have efficient uniprocessor lock-free implementations is far larger than the class that can be efficiently implemented on multiprocessors.

5.4 Assigning Tasks to Supertasks

In this section, we present a simple heuristic for assigning tasks to quantum-based supertasks in order to reduce lock-free overhead. In the next section, we present the results of an experimental study that compares the overhead experienced under several approaches, including the use of the heuristic presented here. The purpose of this section and the experimental study is to demonstrate that supertasking can be an effective means of reducing lock-free overhead, even when reweighting overhead is considered. Since reducing lock-free overhead is but one benefit of supertasking, the heuristic considered here is too simplistic for most systems. Indeed, provisioning supertasks so that total overhead is minimized is one of the prominent unsolved optimization problems in this area.

5.4.1 The Heuristic

Since supertask assignment is a variation of partitioning, we consider the use of a heuristic. (Recall from Chapter 1 that the partitioning problem is NP-hard in the strong sense.) A pseudo-code version of our heuristic is shown in Figure 5.9. Tasks are prioritized based upon the number of accesses made to each object and the degree of contention for that object. We define the *weighted contention* for object ℓ by the value $e_R^{[M]}(\ell) \cdot \sum_{T \in \tau} \frac{\mathcal{J}(T,\ell)}{T.p}$. In this expression, $\sum_{T \in \tau} \frac{\mathcal{J}(T,\ell)}{T.p}$ gives the frequency of accesses to ℓ by all tasks in τ. $e_R^{[M]}(\ell)$ then represents the interference penalty. All tasks that access the object ℓ with the highest weighted contention are assigned to supertasks first. Since each task T's interference depends on $A(T,\ell)$, tasks are assigned in non-increasing order by $A(T,\ell)$. After these tasks are assigned, the remaining objects are considered in non-increasing order by weighted contention.

Assignment rules. The act of assigning selected tasks to supertasks is delegated to an assignment rule. We consider the use of two rules, which are taken from prior work on partitioning [19, 23]. The *Next Fit* (NF) rule begins by assigning tasks to the first (lowest-indexed) supertask. Whenever a supertask is unable to accept a task, the rule moves to the supertask with the next higher index and continues. Once a supertask fails to accept a task, no further attempts are made to assign to that supertask. This rule intuitively matches the goal of the heuristic due to the fact that tasks that are consecutively assigned are likely to be

```
Create $|\tau|$ empty supertasks and add them to $\pi$;
$\tau' := \tau$;
$\Gamma' := \Gamma$;
while $|\Gamma'| > 0 \wedge |\tau'| > 0$ do
    Select $\ell \in \Gamma'$ with largest $e_R^{[M]}(\ell) \cdot \sum_{T \in \tau} \frac{\mathcal{I}(T,\ell)}{T.p}$ value;
    $\Gamma' := \Gamma'/\{\ell\}$;
    while there exists $T \in \tau'$ with $A(T,\ell) > 0$ do
        Select $T \in \tau'$ with largest $A(T,\ell)$;
        $\tau' := \tau'/\{T\}$;
        Assign $T$ to a non-full supertask using the assignment rule
    od
od;
Remove all empty supertasks from $\pi$
```

Figure 5.9: Heuristic algorithm for assigning tasks to supertasks.

assigned to the same supertask.

The *First Fit* (FF) rule, on the other hand, always assigns a task to the lowest-indexed supertask that can accept the task. The advantage of this rule is that it may use fewer supertasks on average, which results in less reweighting overhead. However, the fact that supertasks are not considered in sequence, as is done under the NF rule, implies that the FF rule will be less effective at preventing object sharing across multiple supertasks. Hence, using the FF rule may reduce reweighting overhead, but is also likely to increase lock-free overhead (relative to that experienced when using the NF rule).

5.4.2 Implementation

Unfortunately, implementing this heuristic is a non-trivial task. The problem that arises is that the lock-free overhead depends on the assignment of tasks to supertasks. Hence, the ultimate weight of each task is not known at the time of the assignment. Effectively, each task's weight may expand or contract after being assigned to a supertask. If the total weight of a component task set after these weight changes exceeds unity, then the task assignments are not valid. In such a case, another round of assignments must be performed.

We address the dependency between task weights and supertask assignments by applying the heuristic iteratively. Specifically, an initial round of assignments is conducted using "ideal" weights[7] and an acceptance test of the form

$$\sum_{T \in \mathcal{S}} T.w \leq \alpha,$$

where α is set to unity. Once assignments are made, the weights of the component tasks are updated and each supertask is checked to ensure that the cumulative weight does not exceed unity. If this requirement is satisfied, then the assignment phase ends and the supertasks are reweighted. Otherwise, the supertask assignments are nullified, α is decremented by 0.01, and another round of assignments are conducted using the task weights computed at the end of the previous round.

The benefit of decreasing α (and hence increasingly underestimating the maximum capacity of supertasks) each time a new round begins is that a fraction $1 - \alpha$ of each supertask's maximum capacity is effectively reserved to account for the inflation of task weights. The disadvantage of such compensation is that lowering α can result in an unnecessarily high number of supertasks, thereby increasing reweighting overhead. For this reason, we use a relatively small step size (*i.e.*, 0.01) when decreasing α. We found that termination typically occurs after only two or three rounds when using this step size.

[7]Ideal weights are based on the assumption that no retries occur.

5.5 Experimental Results

In this section, we present the results of two experimental studies that measure the inflation experienced when lock-free synchronization is used. Since we are unable to reasonably estimate the impact of algorithmic improvements obtained through the use of supertasks, such improvements were not considered in either of these studies.

The goal of the first study was to identify the approach that is likely to produce the lowest inflation. Both the QB-EPDF and QB-EDF forms of supertasking (see Chapter 4) were considered; each form was tested using the heuristic proposed in the previous section with each of the FF and NF rules. The resulting inflation was compared to that experienced when no supertasks are used.

The second study focused on the relationship between inflation and the $A(T, \ell)$ values. Specifically, the upper limit imposed on $A(T, \ell)$ values was systematically increased, thereby increasing the worst-case contention for the lock-free objects. The goal of this study was to determine how the performance of each approach scales as contention increases.

5.5.1 Study 1: Retry Overhead

In this section, we present the details and results of the first study. This study was designed to evaluate the efficacy of using supertasks to reduce retry overhead. Though the use of supertasks can reduce worst-case contention, it also introduces reweighting overhead.

Sample space. For this study, the sample space was defined as follows:

- M is in the range $2, \ldots, 16$;
- $|\tau|$ is in the range $5 \cdot \log_2 M, \ldots, 30 \cdot \log_2 M$;

- $\tau.u$ is in the range $0.2 \cdot M, \ldots, 0.8 \cdot M$;
- $|\Gamma|$ is in the range $\log_2 M, \ldots, 10 \cdot \log_2 M$;
- $T.p$ is in the range $100, \ldots, 5000$;
- $A(T, \ell)$ is in the range $1, \ldots, 3$;
- $\mathcal{J}(T, \ell)$ is in the range $A(T, \ell), \ldots, 8$;
- $e_B(\ell)$ is in the range $0.001, \ldots, 0.05$;
- $e_R(\ell)$ is in the range $0.001, \ldots, 0.025$.

These ranges were selected rather arbitrarily so that values that seemed likely to occur in practice would be included in the experiments. For instance, the 0.025 upper bound on $e_R(\ell)$ reflects our expectation that retry loops should seldom, if ever, require more than 2.5% of a quantum. (If retry loops are too lengthy, then the use of lock-free synchronization will provide no advantage over the use of lock-based synchronization.) Similarly, we expect retry loops to require at least 0.1% of a quantum. These expectations are based on the prior work of Ramamurthy [56]. We scale $|\tau|$ and $|\Gamma|$ logarithmically relative to M because contention (and hence inflation) tends to increases quickly as M is increased.

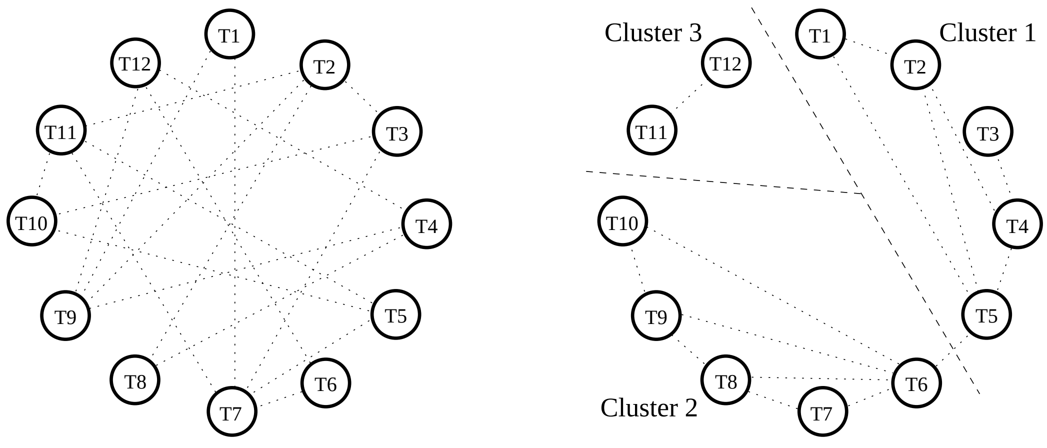
Sampling. To provide fairly uniform coverage, samples were collected as follows:

- M was set to each value in the set $\{\ 2^x \mid 1 \leq x \leq 4\ \}$;
- $|\tau|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 5 \leq x \leq 30\ \}$;
- $\tau.u$ was set to each value in the set $\{\ 0.025x \cdot M \mid 8 \leq x \leq 32\ \}$;
- $|\Gamma|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 1 \leq x \leq 10\ \}$.

For each combination of the above parameter assignments, ten valid component task sets were generated and evaluated. Recall that a valid task set is one that is schedulable under every approach considered by the experiment. As discussed in Chapter 4, we estimate the fraction of the sample population represented by each computed mean by the fraction of generated samples that were determined to be valid. (As explained earlier, we refer to this latter fraction as the *sampling validity*.)

Relevance. As discussed earlier, the performance of random task sets is a questionable indicator of actual performance due to the fact that the relationship between random task sets and real task sets is unknown. However, such experiments can provide insight into performance trends. Unfortunately, random task-set experiments are even more difficult to utilize when studying complex behaviors, such as suspensions and resource sharing. This follows from the fact that the patterns of behavior within a task set significantly impact performance. Hence, to estimate performance reliably, it is necessary to accurately recreate the expected behavior. However, without knowledge of which patterns of behavior are likely to occur, this is not possible.

For simplicity, these experiments generated resource accesses in a uniform fashion. Specifically, when randomly generating a resource access, the resource being accessed was chosen from a uniform distribution of all shared resources. Consequently, the task sets considered in these experiments tended to exhibit widespread object sharing, *i.e.*, if sharing dependencies between tasks were represented using a graph in which nodes represent tasks, then the graph would resemble a large web containing all nodes. Such a scenario may reflect cases in which a multiprocessor is dedicated to a single large-scale, multithreaded application. However, it does not reflect cases in which several applications are multiplexed onto a single multiproces-



(a) Widespread Sharing **(b) Clustered Sharing**

Figure 5.10: Graphs of resource-sharing dependencies between a set of twelve tasks. Scenarios include **(a)** widespread sharing, and **(b)** clustered sharing.

sor. In such a situation, resource sharing would likely be localized to clusters of tasks, each of which represents a single application or some part of an application. Figure 5.10 illustrates these two scenarios. Unfortunately, as with studies of sharing behavior in general, a proper study of clustered sharing requires some prior knowledge of the characteristics of real clusters. Hence, we leave experiments that focus on clustered sharing as future work.

Measurement. The following measurements were taken during the experimental runs.

Ideal: the sum of the ideal weights[8] of all tasks (a baseline measurement); this measurement reflects mapping overhead alone.

No Supertasks: the total weight of all tasks when object accesses are considered and supertasks are not used; this measurement reflects mapping and retry overhead.

Ideal QB (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic (a baseline measurement); this measurement reflects mapping

[8]A task's ideal weight is based on the assumption that no retries are needed for object accesses.

and retry overhead.

Ideal QB (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic (a baseline measurement); this measurement reflects mapping and retry overhead.

QB-EPDF (FF): the sum of the assigned weights of all supertasks created by the FF form of the assignment heuristic under the QB-EPDF scenario; this measurement reflects mapping, retry, and reweighting overhead.

QB-EPDF (NF): the sum of the assigned weights of all supertasks created by the NF form of the assignment heuristic under the QB-EPDF scenario; this measurement reflects mapping, retry, and reweighting overhead.

QB-EDF (FF): the sum of the assigned weights of all supertasks created by the FF form of the assignment heuristic under the QB-EDF scenario; this measurement reflects mapping, retry, and reweighting overhead.

QB-EDF (NF): the sum of the assigned weights of all supertasks created by the NF form of the assignment heuristic under the QB-EDF scenario; this measurement reflects mapping, retry, and reweighting overhead.

To ensure termination when reweighting, $\mathcal{S}.w_\phi + 10^{-4}$ was passed as $w_{\min}$ when invoking `Reweight` (presented earlier in Figure 4.7). When discussing the supertasking approaches, trends in the reweighting overhead will not be noted. (A discussion of reweighting overhead can be found in Chapter 4.) The sampling validity was consistently around unity for the two- and four-processor cases. For this reason, we omit graphs of sampling validity for those cases.

Overall Performance. The tables shown in Figures 5.11–5.14 summarize the overall results of this study. Separate summaries are given for each value of M. As in Chapter 4, these tables consist of five columns. The first (leftmost) column identifies the approach associated with each row. The second ("Mean") column shows the mean inflation experienced under each approach. The third ("Standard Deviation") column gives the standard deviation of the samples used to compute that mean. The fourth ("Halflength") column gives the halflength of a 99% confidence interval. A confidence interval is defined to be $[m - h, m + h]$, where m denotes the mean and h denotes the halflength. Finally, the fifth column reports what fraction of the sample sets favored the use of each approach. In this comparison, attention was restricted to only a few of the tested approaches. *N/A* is used to signify which approaches were omitted from the comparison. A more detailed pairwise comparison of approaches was also conducted; the results can be found in Appendix B.

Notice that the fifth column includes a "QB-EDF (Both)" measurement that was not described above. This case refers to samples for which QB-EDF is the favored approach and both the NF and FF rules produced identical results. For low processor counts, the NF and FF rules often produce identical results, which implies that the choice of assignment rule is of little importance. However, as the processor count increases, more supertasks are needed on average, and the likelihood that these rules provide identical performance drops. We include this special measurement primarily to gauge the importance of the assignment rule.

Consider the $M = 2$ results first, which are shown in Figure 5.11. In this case, the QB-EDF approach produced the lowest inflation on average. (Recall that the Ideal entries reflect baseline measurements and not actual approaches.) In addition, the QB-EPDF approach performs much worse than not using supertasks. This relationship is the result of two factors.

$M = 2$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.010632	0.006481	0.00006548	*N/A*
No Supertasks	0.022861	0.015684	0.00015847	2.791%
Ideal QB (FF)	0.013820	0.009000	0.00009093	*N/A*
Ideal QB (NF)	0.013897	0.009055	0.00009149	*N/A*
QB-EPDF (FF)	0.042780	0.060778	0.00061408	*N/A*
QB-EPDF (NF)	0.038047	0.056036	0.00056616	*N/A*
QB-EDF (FF)	0.014029	0.009000	0.00009093	7.517%
QB-EDF (NF)	0.014088	0.009055	0.00009149	3.903%
QB-EDF (Both)	*N/A*	*N/A*	*N/A*	85.789%

Figure 5.11: Summary of the results of the first lock-free experiment for the $M = 2$ case.

First, the QB-EPDF approach is the most costly with respect to reweighting overhead (according to the studies presented in Chapter 4). Second, worst-case interference is determined by considering $M - 1$ interfering tasks. Consequently, for small values of M, such as $M = 2$, interference is already limited. As a result, the reduction in retry overhead obtained by using supertasks is often outweighed by reweighting overhead.

The variance of samples does not appear to significantly differ from that observed in the reweighting experiments presented in Chapter 4. Again, QB-EPDF shows significantly more variance than the other approaches. Indeed, the fact that the performance of the supertasking approaches strongly resembles that observed earlier in the reweighting experiments suggests that the retry overhead did not significantly impact this case. (An obvious exception is the "No Supertasks" mean.)

The rightmost column relates the fraction of the valid task sets that favored each approach. Because QB-EPDF cannot improve upon QB-EDF, we considered only the "No Supertasks" and QB-EDF approaches in this comparison. As shown, 2.791% of the sets showed no benefit to using either form of the QB-EDF approach. For 85.789% of the task sets, both forms of the QB-EDF approach performed indentically and provided some improvement. Finally, the

FF (respectively, NF) form of QB-EDF provided the lowest inflation for 7.517% (respectively, 3.903%) of the sample sets. This last result suggests that neither the NF nor FF assignment rules can be assumed to always produce better results than the other. However, this result does suggest that the FF rule will likely be the better choice. Notice, however, that the FF rule appears to produce worse results than the NF rule when used with QB-EPDF.

Next, consider the $M = 4$ results, which are shown in Figure 5.12. Due to the increased parallelism, the worst-case interference increases as well, relative to the previous case. As a result, both QB-EPDF and QB-EDF improved upon "No Supertasks" in this case. (This suggests that improvement is likely to occur for more than four processors as well.) Indeed, the fraction of task sets that favor not using supertasks has dropped below 0.1%, as shown in the last column. As before, QB-EDF provides a significant improvement over QB-EPDF. In addition, the relationship between the NF and FF rules observed for the $M = 2$ case continues to hold here, *i.e.*, the FF rule likely provides the lowest inflation, though the NF rule may provide better results for a significant fraction of the task sets. Again, the use of the FF rule with QB-EPDF appears to be an exception. In this and the remaining cases, the variance of samples is unremarkable, *i.e.*, the standard deviation appears to track the sample means relatively closely for each approach. Because of this, we do not remark on the sample variance beyond this point. Finally, notice that (according to the fifth column) the choice of assignment rule more significantly impacts results in this case. Whereas around 85% of the tested sets showed no preference for either rule in the $M = 2$ case, only 47% showed no preference in this case. Instead, the FF rule appears to be the prefered approach.

The results of the $M = 8$ case are shown in Figure 5.13. These results show basically the same trend as the $M = 4$ case. Specifically, QB-EDF continued to perform better than QB-

$M = 4$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.026062	0.014900	0.00015054	*N/A*
No Supertasks	0.134587	0.101213	0.00102261	0.029%
Ideal QB (FF)	0.057995	0.047864	0.00048360	*N/A*
Ideal QB (NF)	0.059094	0.049122	0.00049631	*N/A*
QB-EPDF (FF)	0.093600	0.084012	0.00084882	*N/A*
QB-EPDF (NF)	0.090789	0.085212	0.00086094	*N/A*
QB-EDF (FF)	0.058327	0.047875	0.00048370	41.088%
QB-EDF (NF)	0.059407	0.049143	0.00049651	11.615%
QB-EDF (Both)	*N/A*	*N/A*	*N/A*	47.268%

Figure 5.12: Summary of the results of the first lock-free experiment for the $M = 4$ case.

$M = 8$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.046311	0.026363	0.00026636	*N/A*
No Supertasks	0.531052	0.423021	0.00427401	0.009%
Ideal QB (FF)	0.219781	0.203902	0.00206013	*N/A*
Ideal QB (NF)	0.227022	0.212299	0.00214497	*N/A*
QB-EPDF (FF)	0.295368	0.241692	0.00244194	*N/A*
QB-EPDF (NF)	0.319000	0.288635	0.00291623	*N/A*
QB-EDF (FF)	0.220359	0.203941	0.00206053	72.594%
QB-EDF (NF)	0.227624	0.212325	0.00214523	15.111%
QB-EDF (Both)	*N/A*	*N/A*	*N/A*	12.286%

Figure 5.13: Summary of the results of the first lock-free experiment for the $M = 8$ case.

EPDF, while the performance gap between the "No Supertasks" and supertask approaches widened further. In addition, the FF rule is increasingly favored over the NF rule when using QB-EDF. However, one aspect of these results that differs from those of previous cases is that the QB-EPDF now favors the FF rule rather than the NF rule. The $M = 16$ case, summarized in Figure 5.14, appears to simply continue these trends.

Impact of the task count. We now consider how inflation varied with the task count, task set utilization, object count, and weighted contention. Consider the task count first. Figures 5.15, 5.16, 5.17, and 5.18 show how inflation varies with the task count on two, four, eight, and sixteen processors, respectively. In all graphs, the "Ideal QB (FF)" and "QB-EDF

$M = 16$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.069839	0.038872	0.00039274	*N/A*
No Supertasks	1.466354	1.180997	0.01193224	0.211%
Ideal QB (FF)	0.648293	0.579701	0.00585702	*N/A*
Ideal QB (NF)	0.678701	0.612468	0.00618809	*N/A*
QB-EPDF (FF)	0.905877	0.657126	0.00663929	*N/A*
QB-EPDF (NF)	1.079902	0.860122	0.00869027	*N/A*
QB-EDF (FF)	0.649586	0.579636	0.00585637	85.772%
QB-EDF (NF)	0.680260	0.612297	0.00618636	12.894%
QB-EDF (Both)	*N/A*	*N/A*	*N/A*	1.123%

Figure 5.14: Summary of the results of the first lock-free experiment for the $M = 16$ case.

(FF)" lines are all virtually co-linear. This is also true of the "Ideal QB (NF)" and "QB-EDF (NF)" curves. Graphs of the sampling validity for the eight- and sixteen-processor cases are shown in Figure 5.19. We summarize the relationships suggested by these graphs below.

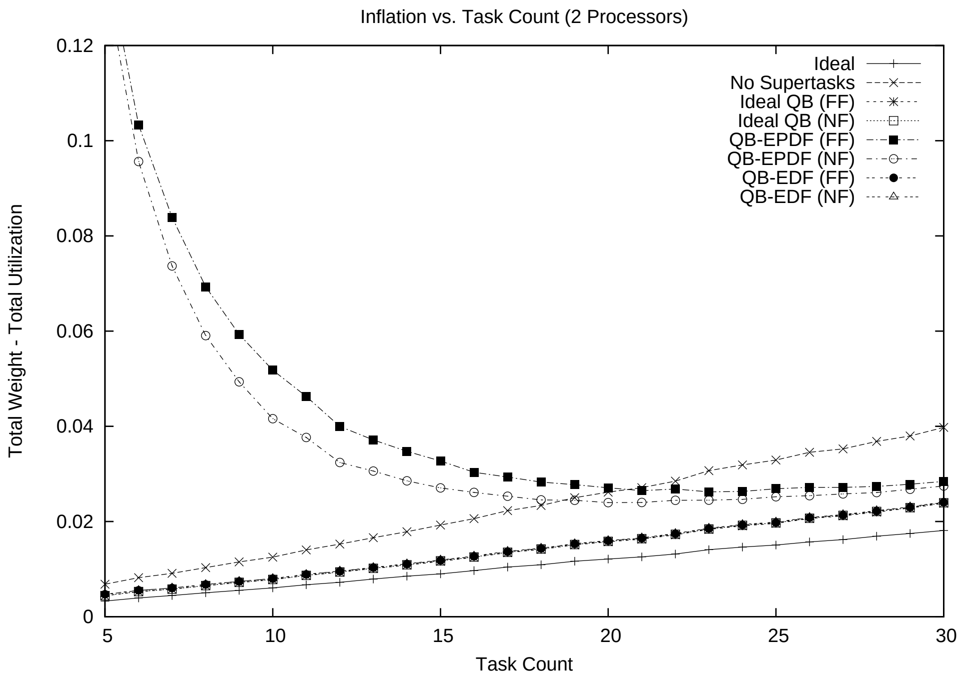
Observation 5.1 *The QB-EPDF approach provides no benefit on two, four, eight, and sixteen processors when task counts are approximately less than 20, 20, 30, and 40, respectively.*

Explanation: Lower task counts tend to produce low component task counts within the supertasks. As shown in the experiments in Chapter 4, low component task counts produce more reweighting overhead, particularly under QB-EPDF.

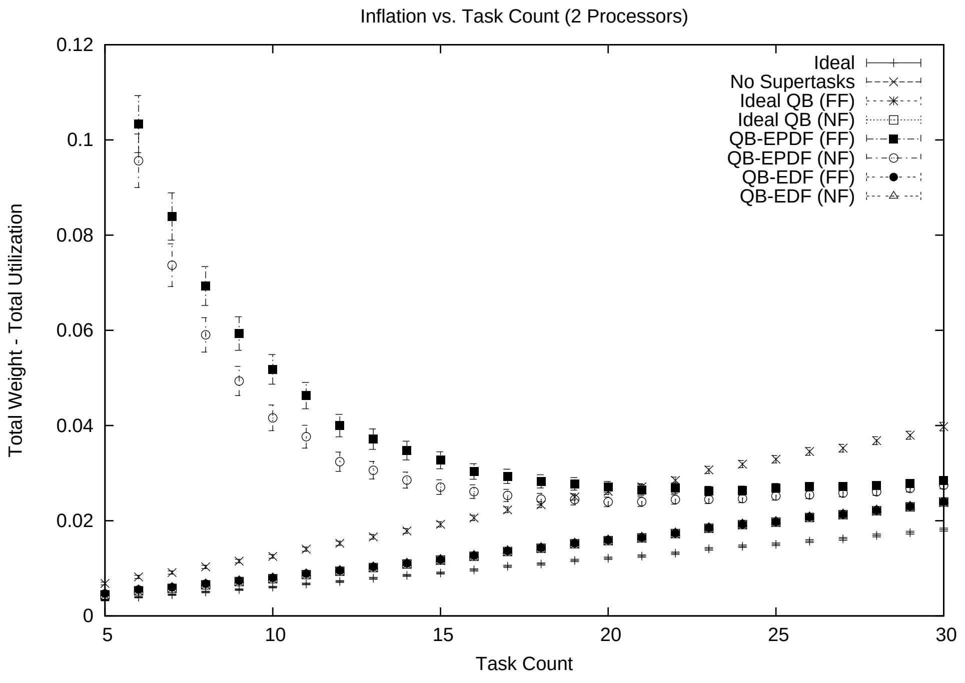
Observation 5.2 *Retry overhead increases linearly with the task count.*

Explanation: This relationship is most apparent by comparing the "Ideal," "No Supertasks," and "Ideal QB" curves. Although all curves are linear, the slopes of the "No Supertasks" and "Ideal QB" curves are steeper than that of "Ideal." This implies that the retry overhead is increasing linearly.

This behavior is the result of the random generation process. The number of objects accessed by each task is randomly chosen from the range $1, \ldots, |\Gamma|$. As a result, an increase in

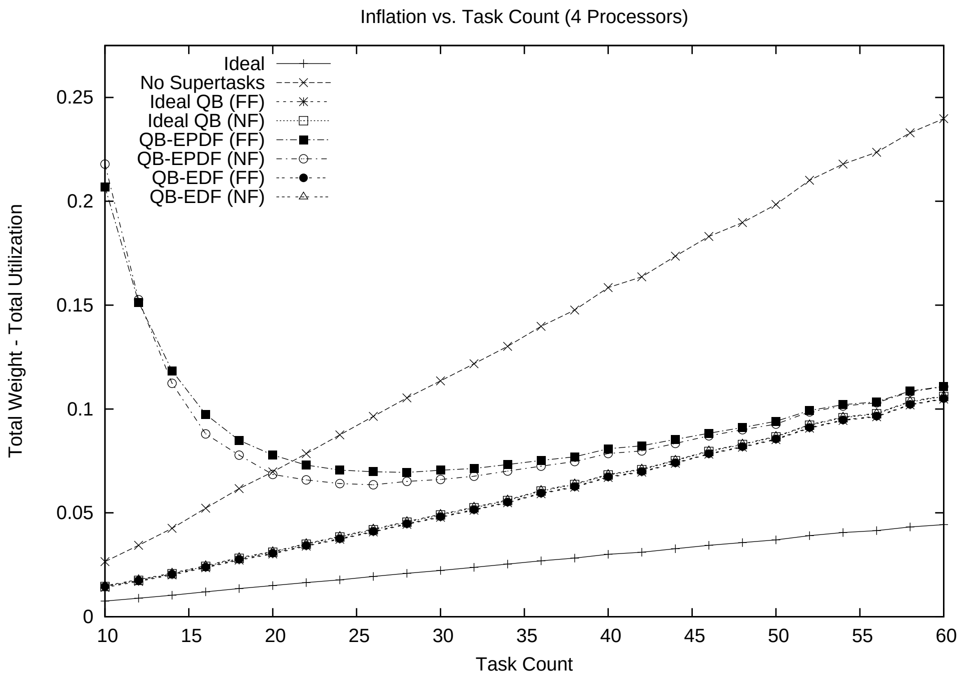


(a) Sample Means Only

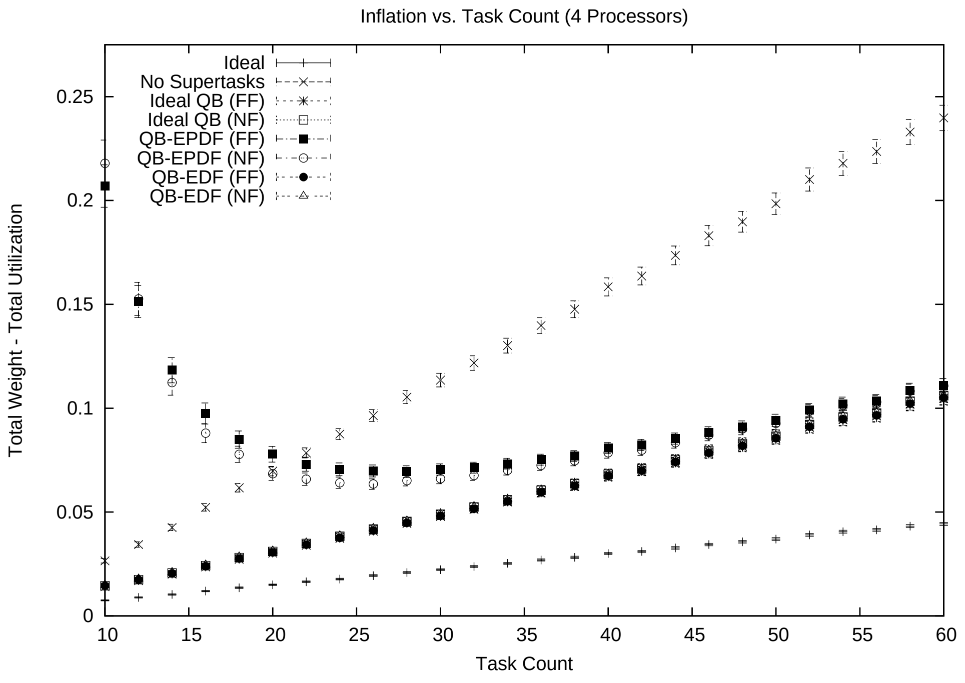


(b) 99% Confidence Interval

Figure 5.15: Plots show how inflation varies as the task count is increased when using lock-free synchronization on two processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

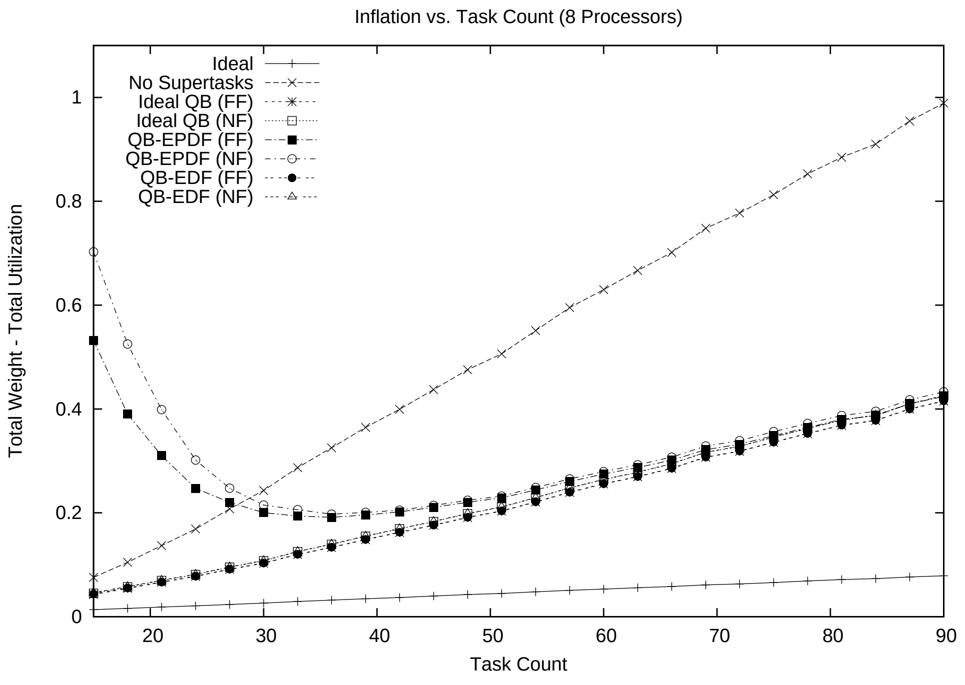


(a) Sample Means Only

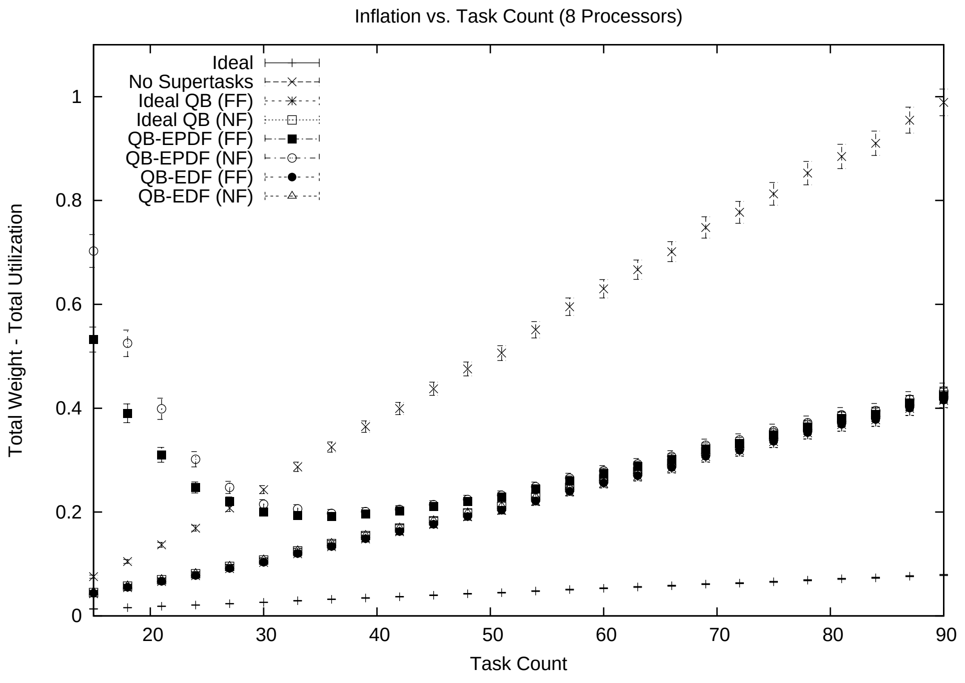


(b) 99% Confidence Interval

Figure 5.16: Plots show how inflation varies as the task count is increased when using lock-free synchronization on four processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

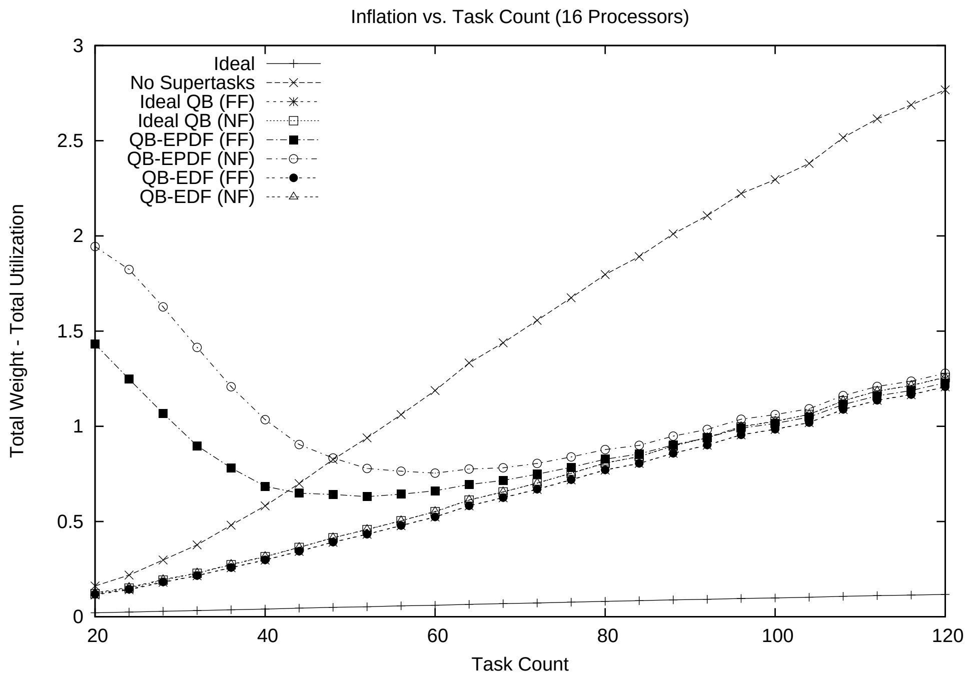


(a) Sample Means Only

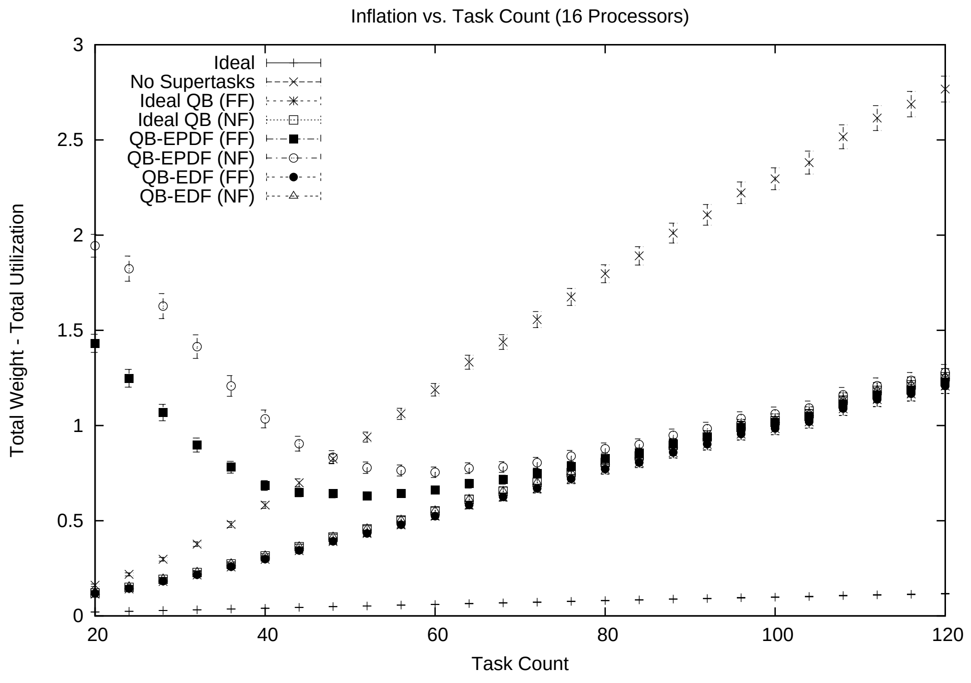


(b) 99% Confidence Interval

Figure 5.17: Plots show how inflation varies as the task count is increased when using lock-free synchronization on eight processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

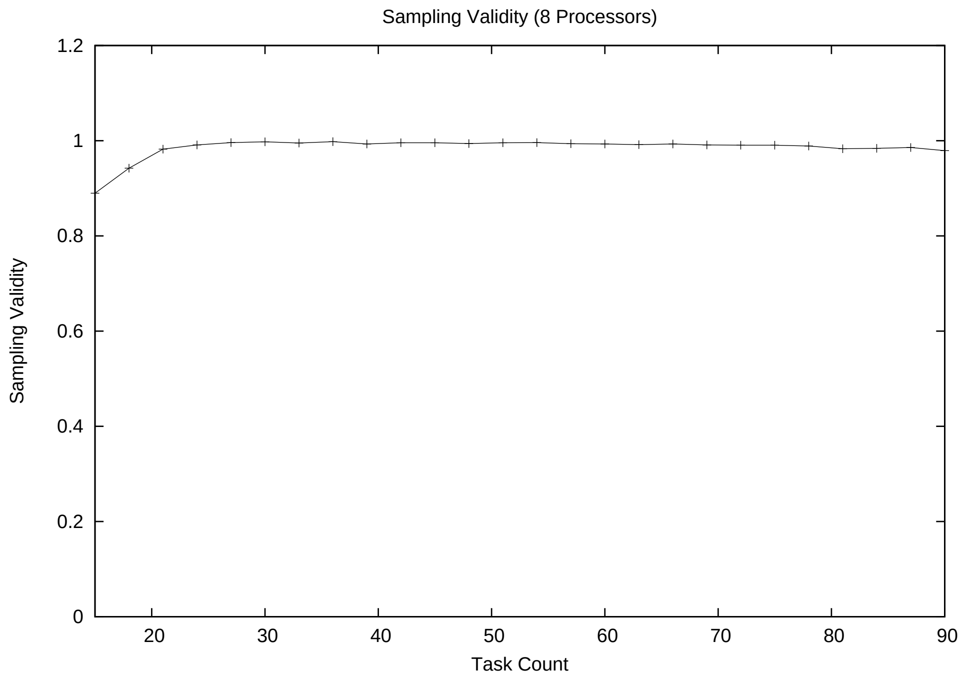


(a) Sample Means Only

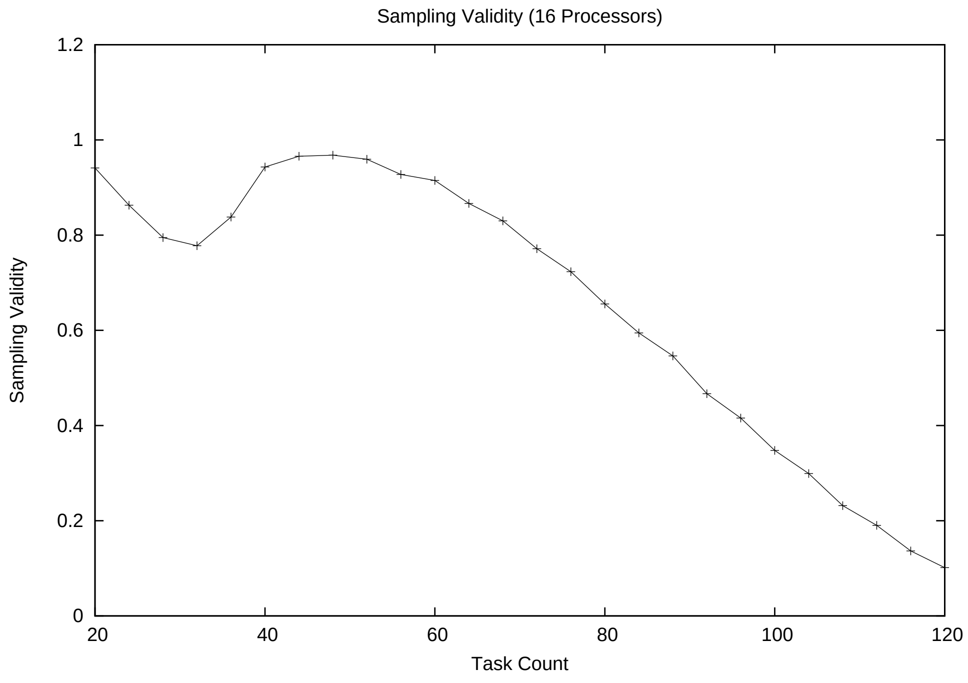


(b) 99% Confidence Interval

Figure 5.18: Plots show how inflation varies as the task count is increased when using lock-free synchronization on sixteen processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) $M = 8$



(b) $M = 16$

Figure 5.19: Plots show the sampling validity for the graphs showing inflation plotted against the task count. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases. The validity was approximately unity throughout the $M = 2$ and $M = 4$ cases. Hence, graphs of those cases are not shown.

either the task count or the object count will produce more sharing, and hence more inflation.

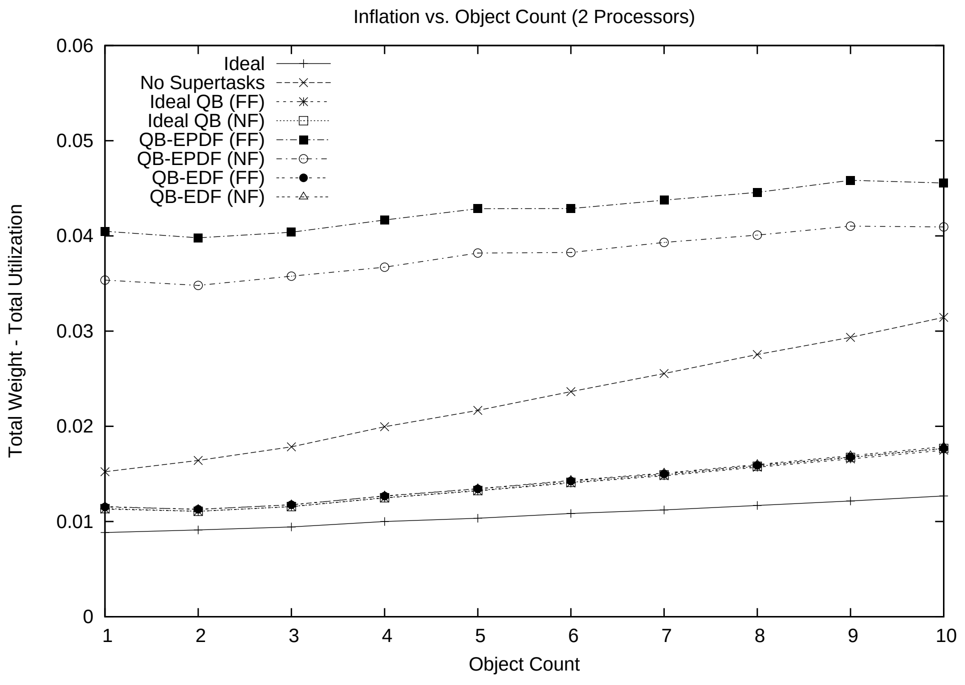
Notice that the relative performance of the FF and NF forms of the QB-EPDF approach is inconsistent. For lower M values, the FF rule performs worse than the NF rule. However, for larger M values, the FF rule performs better. The cause of this behavior is not clear.

Also, notice that the sampling validity curve for the $M = 16$ case, shown in Figure 5.19(b), has an unusual shape. This shape is easily explained by considering the graph of inflation given in Figure 5.18. Specifically, the first drop in sampling validity, which occurs when the task count is in the range $[20, 40]$, is caused by the reweighting overhead of the QB-EPDF approach. As the task count increases, this overhead reduces, which results in increasing sampling validity. However, the sampling validity drops off again for task counts above 50 due to the increasing retry overhead experienced under the "No Supertasks" approach.

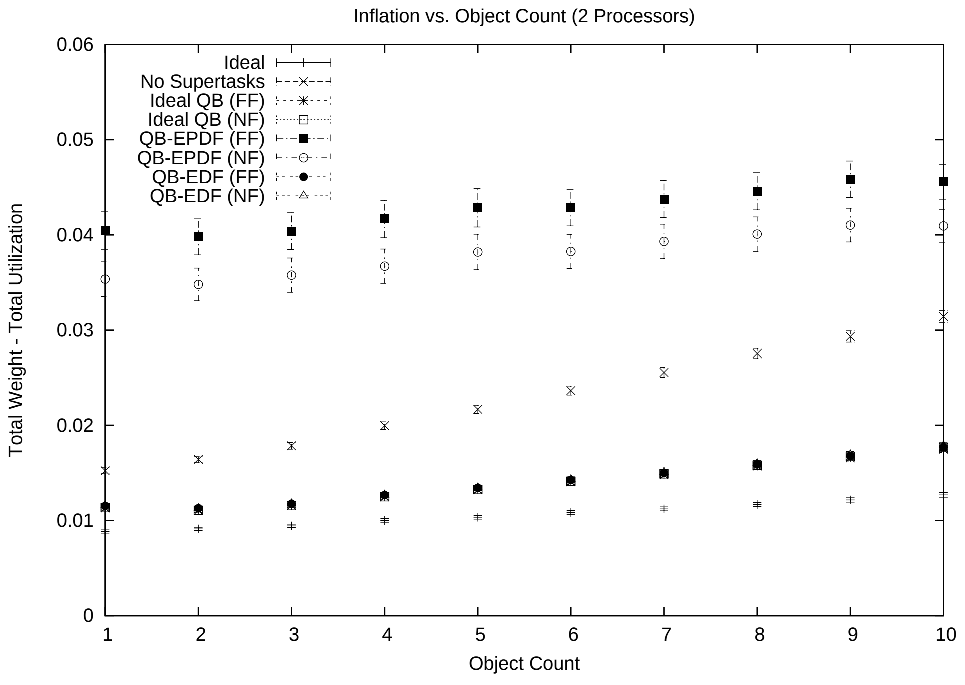
Impact of the object count. Figures 5.20, 5.21, 5.22, and 5.23 show how inflation varies with the object count on two, four, eight, and sixteen processors, respectively. Again, the "Ideal QB" and QB-EDF lines are all virtually co-linear. The sampling validity for the eight- and sixteen-processor cases is plotted in Figure 5.24. As shown, the sampling validity is close to unity for the eight-processor case, but decreases approximately linearly in the sixteen-processor case. Figure 5.23 suggests that this decline is due to the high retry overhead experienced under the "No Supertasks" approach. We summarize other relationships suggested by these graphs below.

Observation 5.3 *Retry overhead increases linearly with the object count.*

Explanation: As explained above, this behavior is the result of the random generation process. Specifically, each task is randomly chosen to access between 1 and $|\Gamma|$ objects.

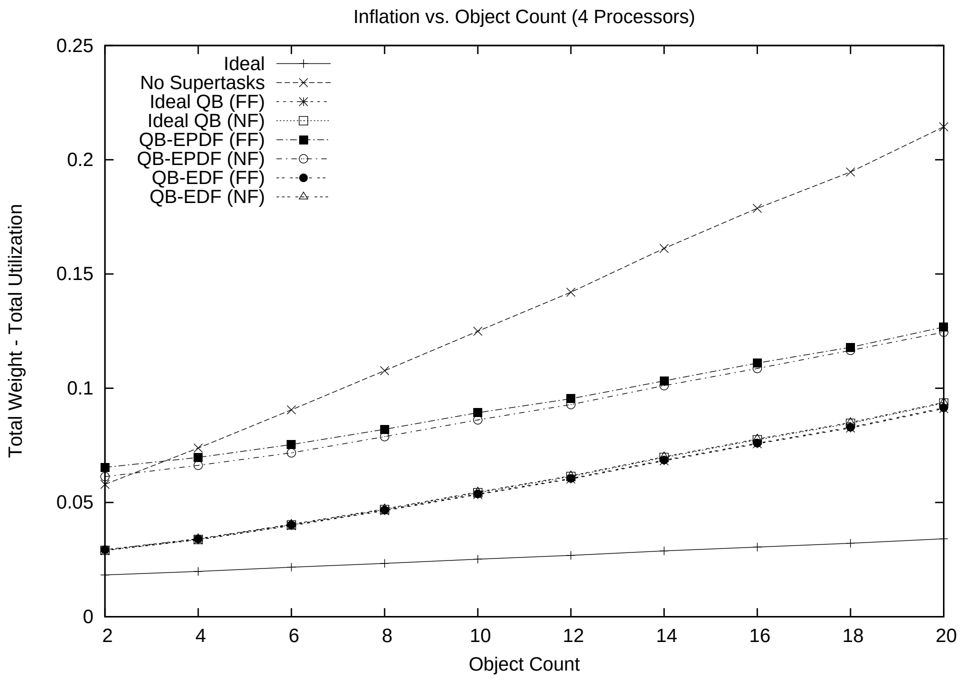


(a) Sample Means Only

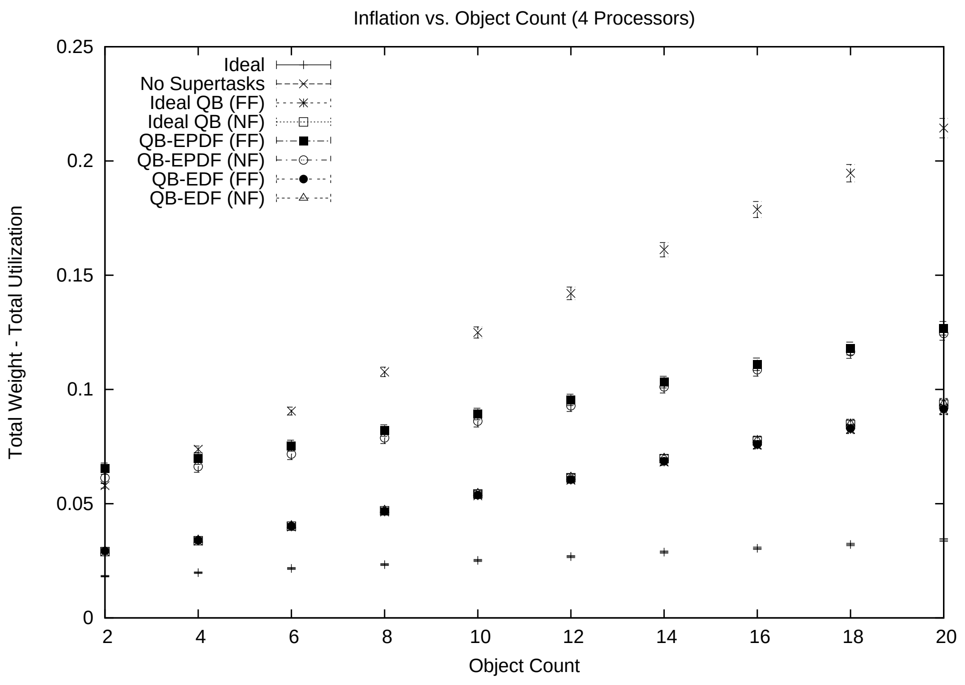


(b) 99% Confidence Interval

Figure 5.20: Plots show how inflation varies as the object count is increased when using lock-free synchronization on two processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

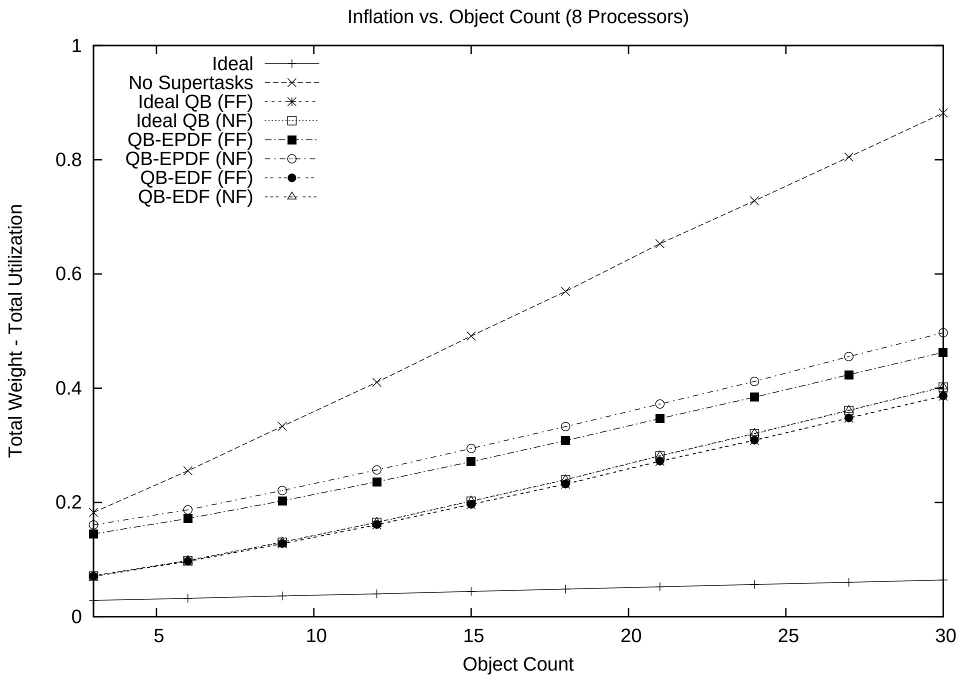


(a) Sample Means Only

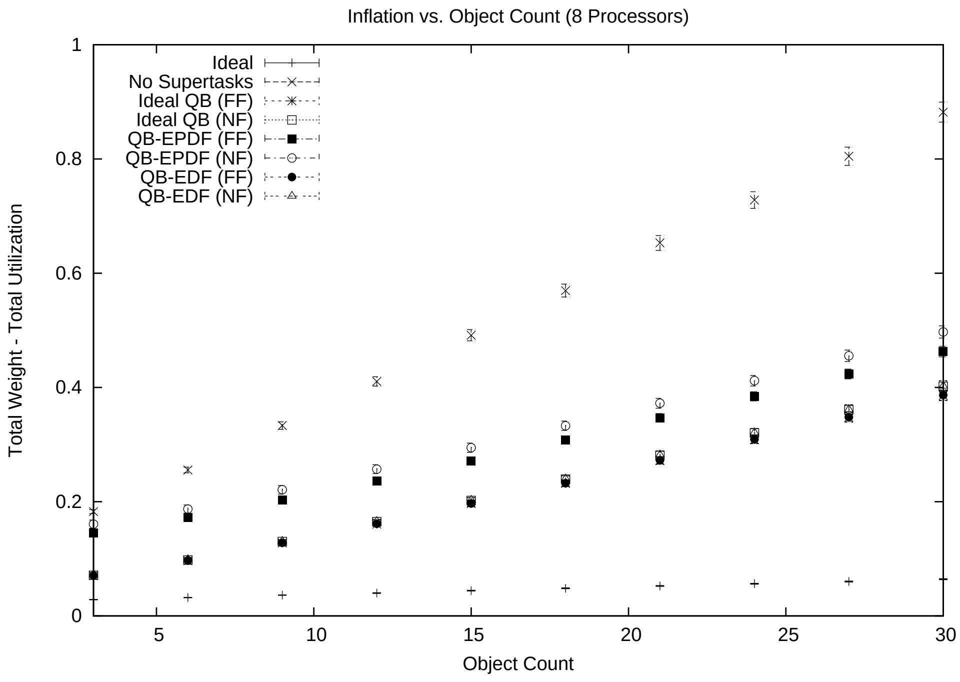


(b) 99% Confidence Interval

Figure 5.21: Plots show how inflation varies as the object count is increased when using lock-free synchronization on four processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

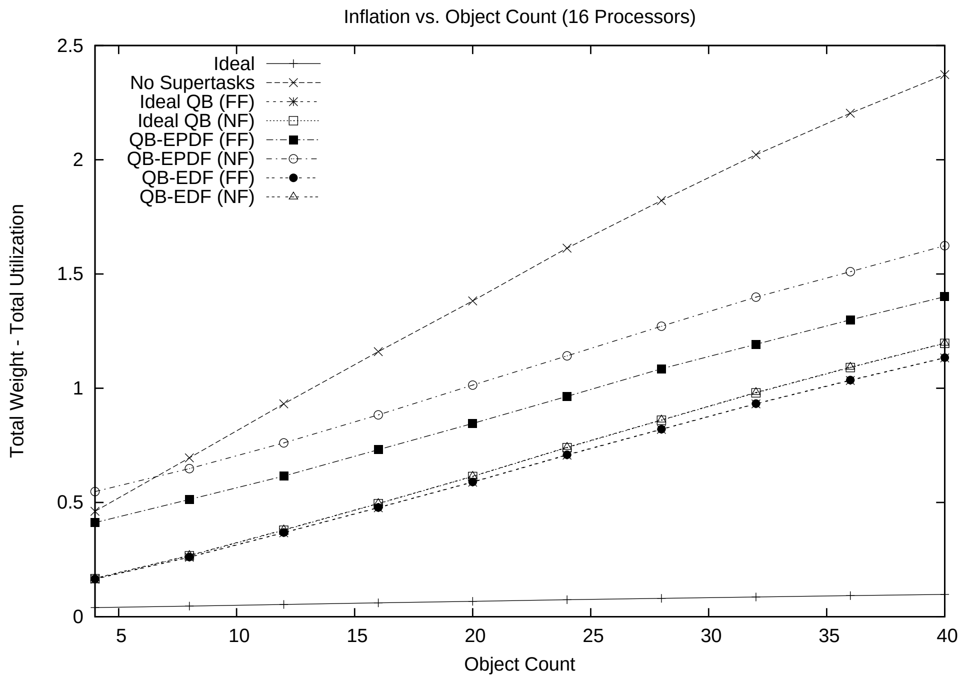


(a) Sample Means Only

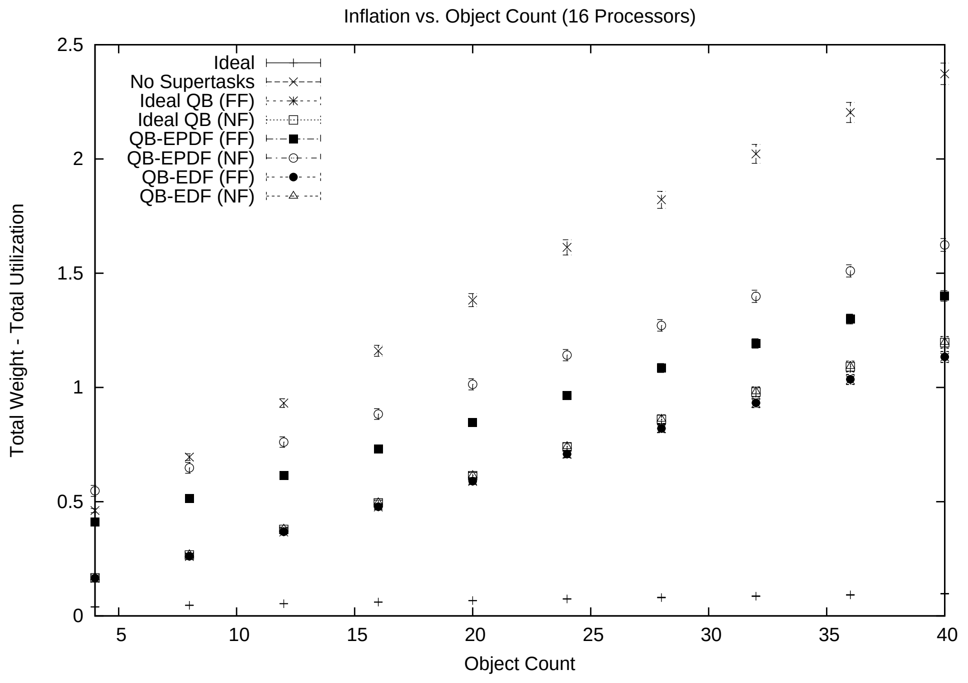


(b) 99% Confidence Interval

Figure 5.22: Plots show how inflation varies as the object count is increased when using lock-free synchronization on eight processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

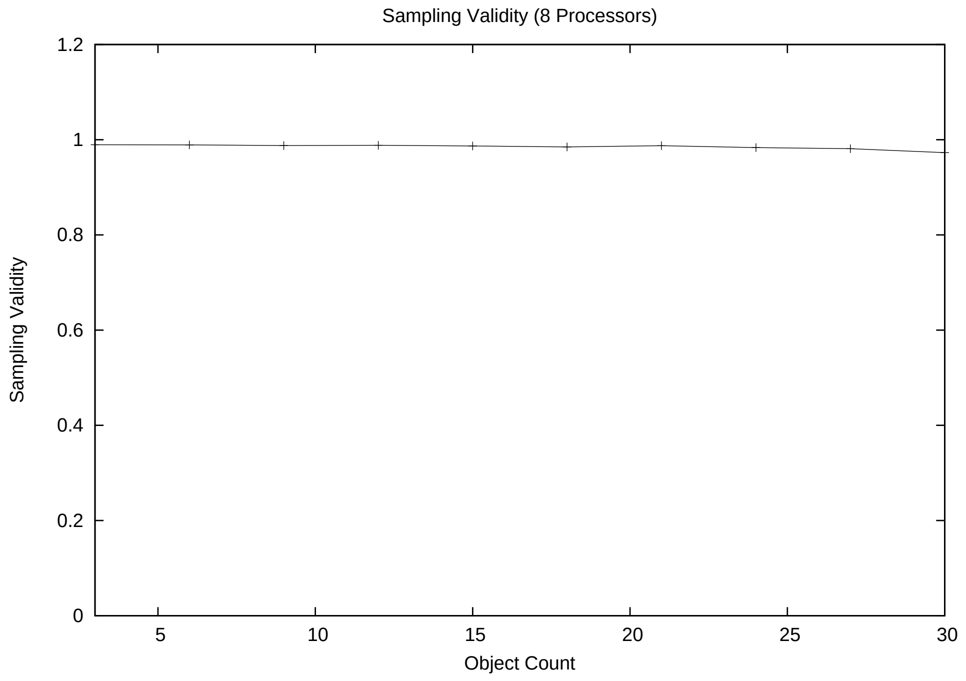


(a) Sample Means Only

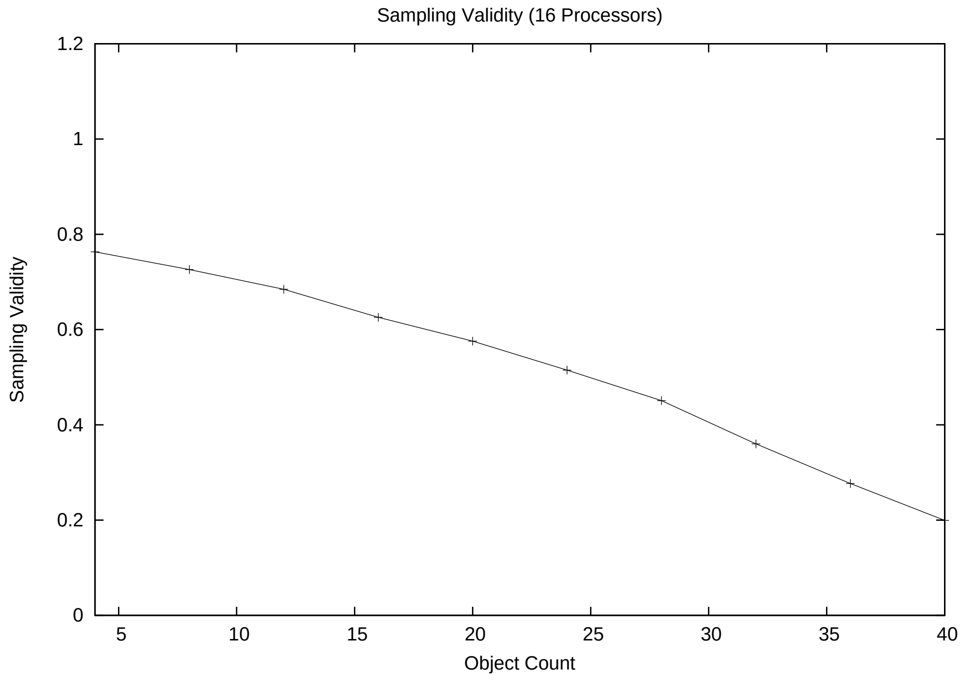


(b) 99% Confidence Interval

Figure 5.23: Plots show how inflation varies as the object count is increased when using lock-free synchronization on sixteen processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) $M = 8$



(b) $M = 16$

Figure 5.24: Plots show the sampling validity for the graphs showing inflation plotted against the lock count. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases. The validity was approximately unity throughout the $M = 2$ and $M = 4$ cases. Hence, graphs of those cases are not shown.

Hence, as $|\Gamma|$ increases, contention and inflation increases as well.

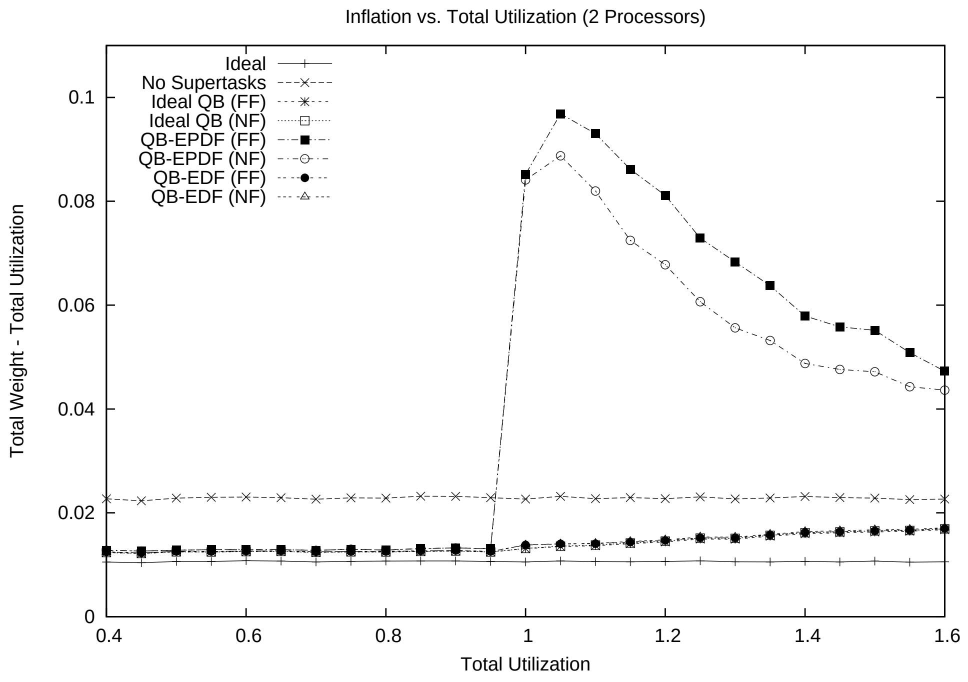
Observation 5.4 *Performance of the FF and the NF rules diverges as* $|\Gamma|$ *and* M *increase. The NF rule slightly improves on the FF rule under QB-EDF, but this relationship is inverted for the QB-EPDF approach.*

Explanation: There is no obvious reason for either of these trends. This behavior warrants further study.

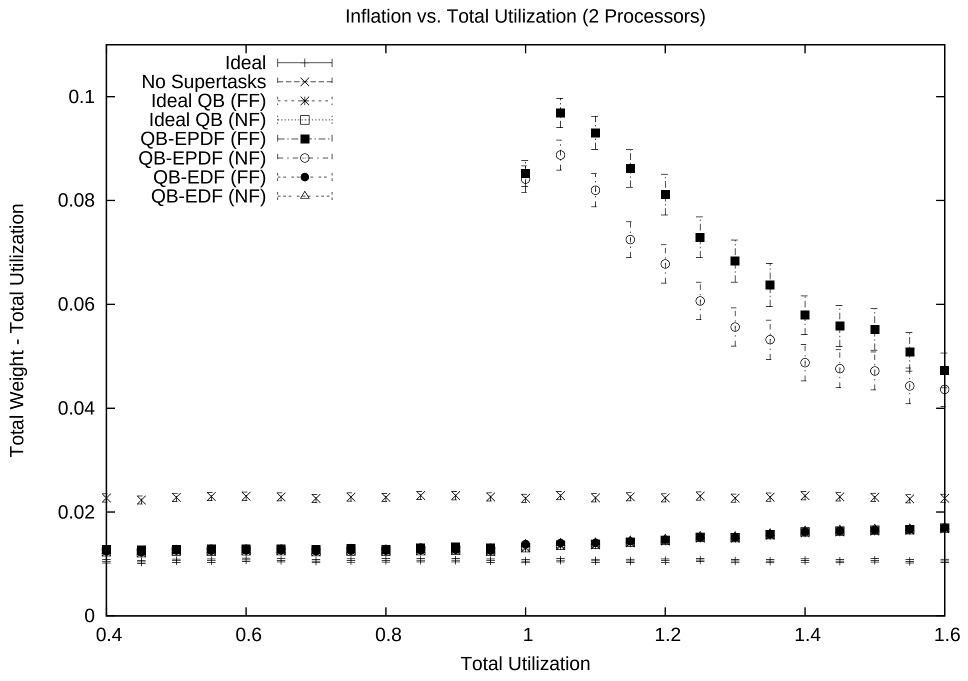
The only other trend of interest is the poor performance of QB-EPDF in the $M = 2$ case. This trend was already discussed earlier on page 204.

Impact of utilization. Figures 5.25, 5.26, 5.27, and 5.28 show how inflation varies with the task set utilization on two, four, eight, and sixteen processors, respectively. Again, the "Ideal QB (FF)" and "QB-EDF (FF)" lines (respectively, the "Ideal QB (NF)" and "QB-EDF (NF)" lines) are virtually co-linear. The sampling validity for the eight- and sixteen-processor cases is plotted in Figure 5.29. As shown in both graphs, the sampling validity is fairly consistent (though not ideal in the $M = 16$ case) at low utilizations, but drops off steeply around the high end of the range. This is not surprising since task sets with higher utilizations leave less spare processor capacity to be consumed by retry overhead. This decrease in sampling validity is the cause of the skew that can be seen at these high utilizations in Figures 5.27 and 5.28

The most obvious trend in these graphs is the seemingly erratic behavior of QB-EPDF. Comparing these curves to the "Ideal QB" curves implies that reweighting overhead is the cause of this behavior. Recall that an unexpected dependency between utilization and reweighting overhead was previously noted earlier in Chapter 4. We believe that the behavior

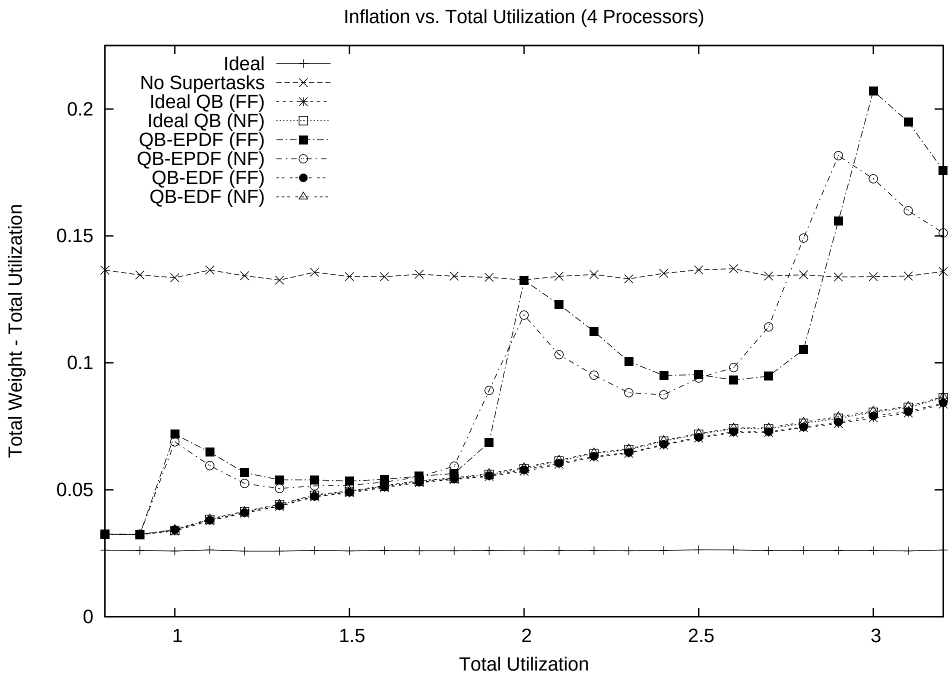


(a) Sample Means Only

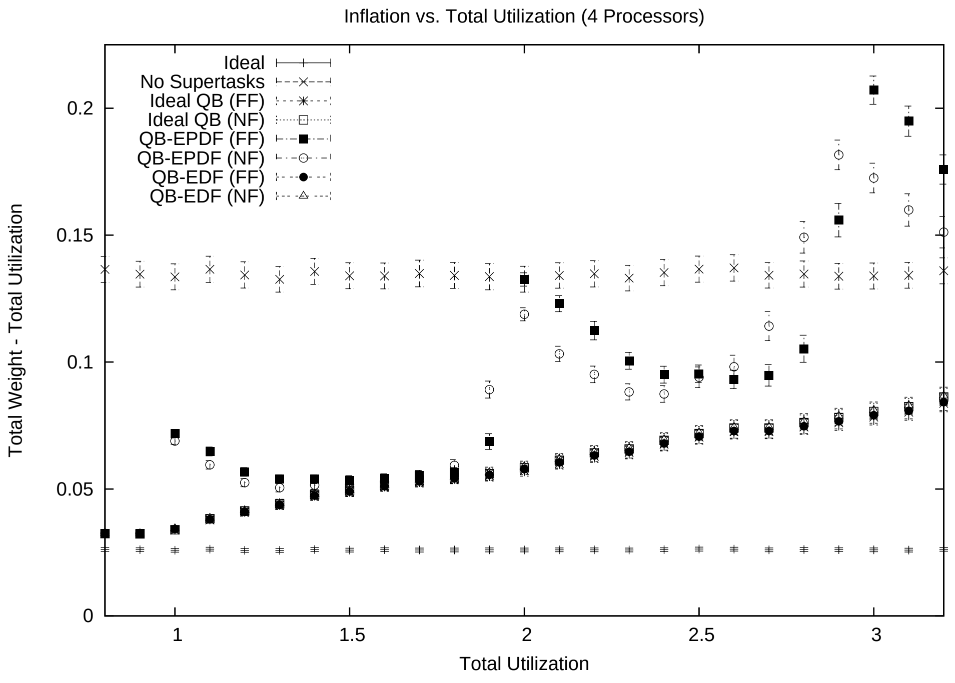


(b) 99% Confidence Interval

Figure 5.25: Plots show how inflation varies as the task set utilization is increased when using lock-free synchronization on two processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

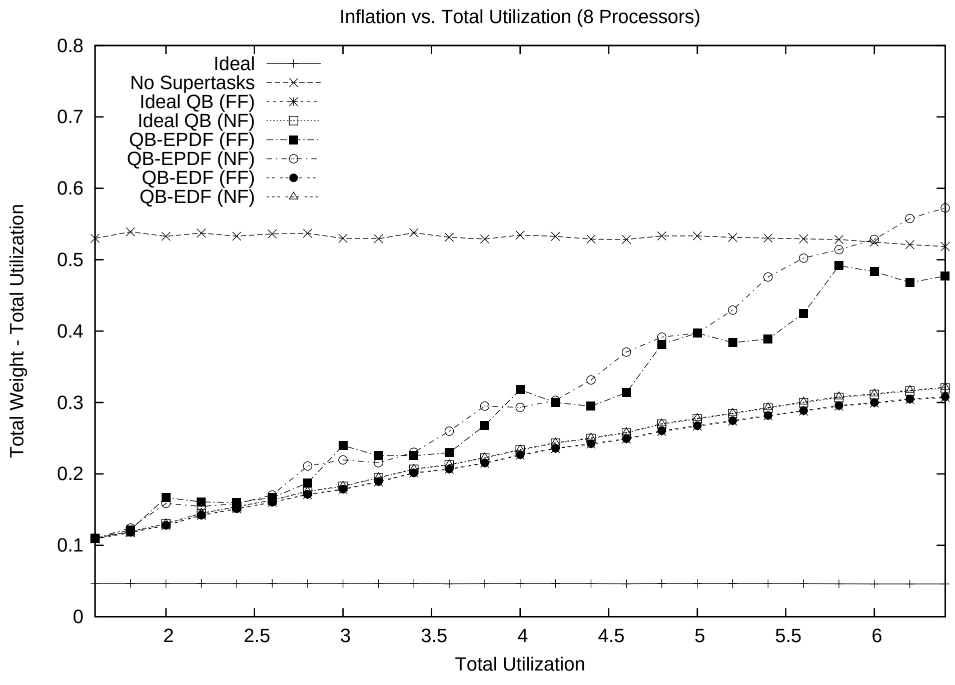


(a) Sample Means Only

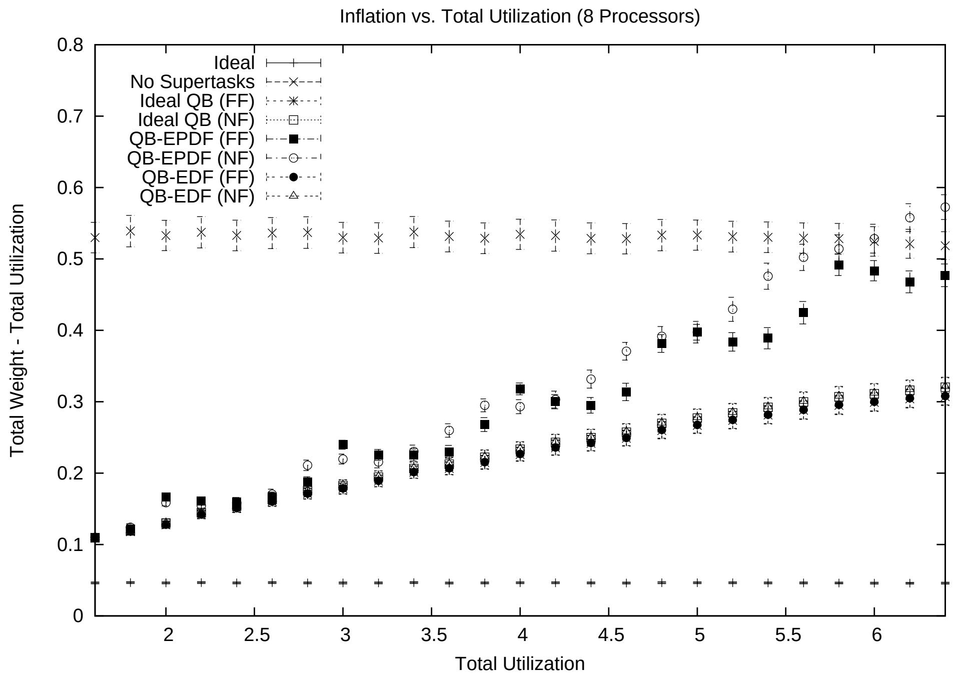


(b) 99% Confidence Interval

Figure 5.26: Plots show how inflation varies as the task set utilization is increased when using lock-free synchronization on four processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

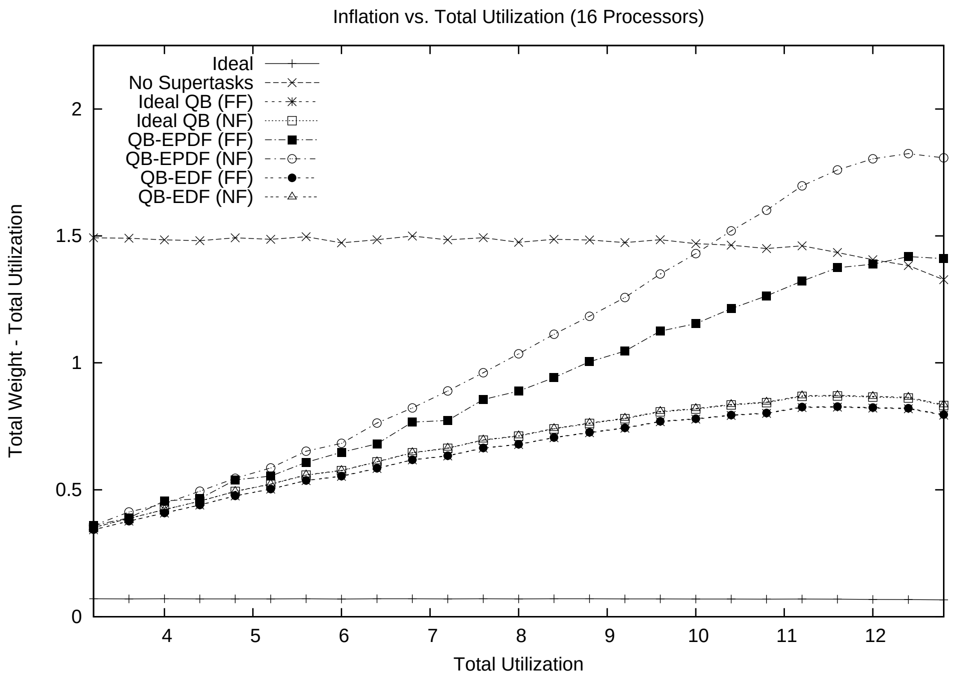


(a) Sample Means Only

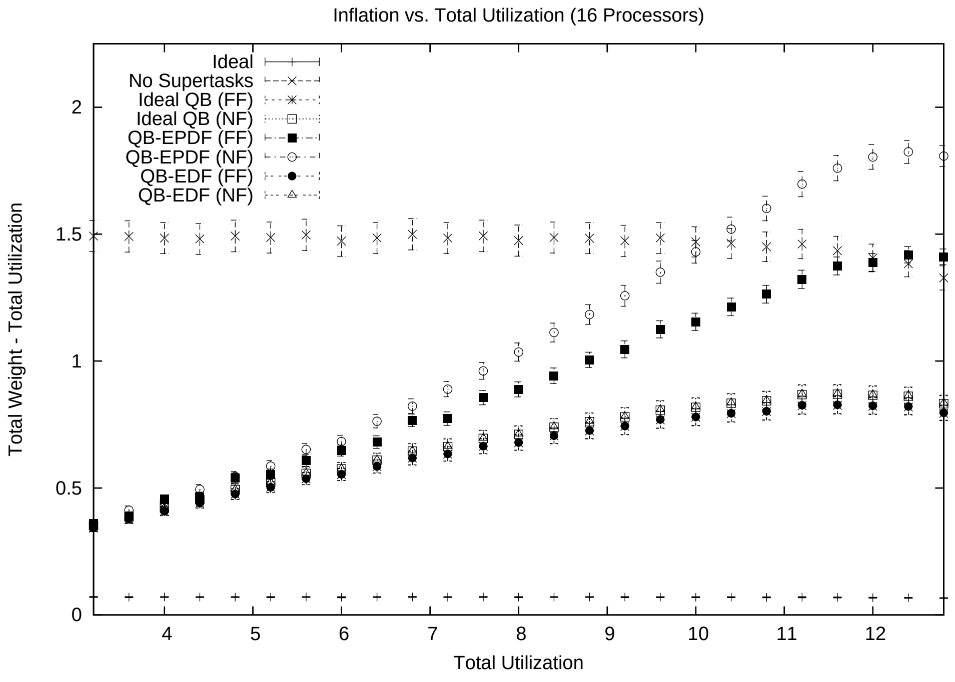


(b) 99% Confidence Interval

Figure 5.27: Plots show how inflation varies as the task set utilization is increased when using lock-free synchronization on eight processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

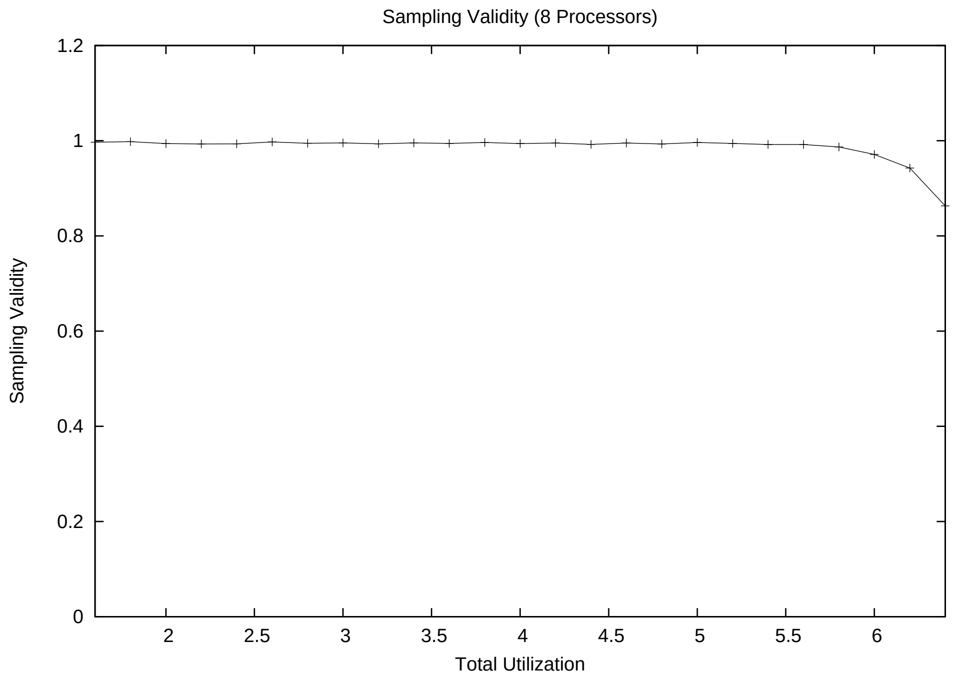


(a) Sample Means Only

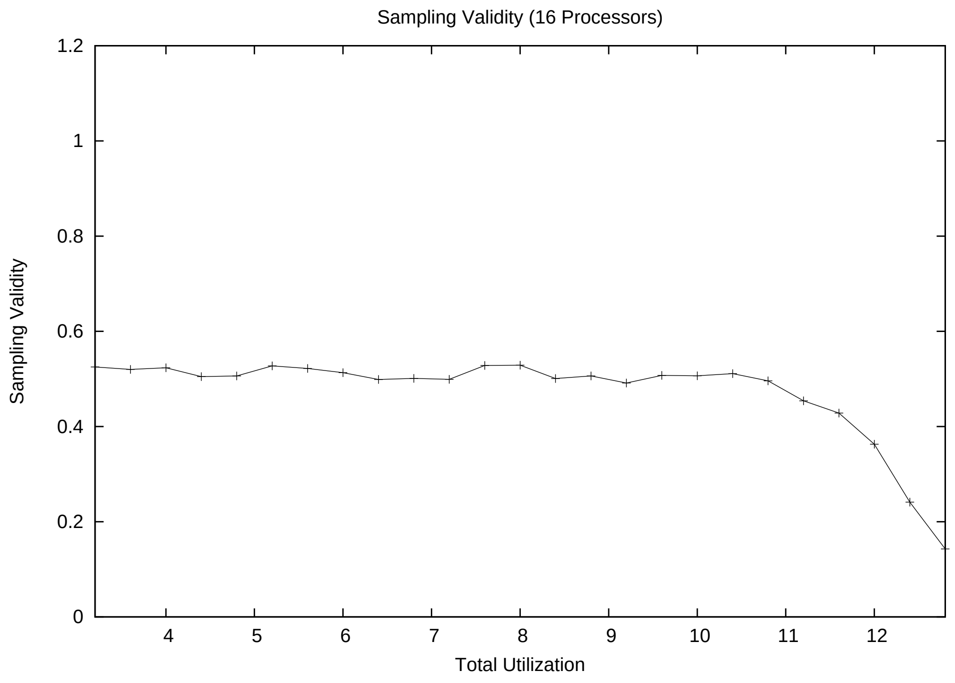


(b) 99% Confidence Interval

Figure 5.28: Plots show how inflation varies as the task set utilization is increased when using lock-free synchronization on sixteen processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) $M = 8$



(b) $M = 16$

Figure 5.29: Plots show the sampling validity for the graphs showing inflation plotted against the task set utilization. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases. The validity was approximately unity throughout the $M = 2$ and $M = 4$ cases. Hence, graphs of those cases are not shown.

observed here is a by-product of the same dependency. We summarize other relationships suggested by these graphs below.

Observation 5.5 *Retry overhead is independent of the task set utilization when not using supertasks. However, this overhead is dependent on the utilization when using supertasks.*

Explanation: The independence property follows simply from the fact that the utilization does not impact the formulas used to bound retry overhead. A dependency is introduced when using supertasks because the utilization dictates the number of supertasks needed to service the task set. Larger numbers of supertasks tend to result in increased worst-case interference, and hence more inflation.

Observation 5.6 *Retry overhead is worse under the NF rule than the FF rule for high utilizations.*

Explanation: For higher total utilizations, individual task utilizations and the range of task utilizations tend to be larger. Because the NF rule abandons a supertask on the first assignment failure, it tends to create more supertasks than the FF rule. To understand why this leads to more overhead, suppose that each component task set has a total utilization of 0.92 (on average) under the FF rule, but a total utilization of only 0.8 under the NF rule. When the total task set utilization is 16, the FF rule can be expected to create around $\left\lceil \frac{16}{0.92} \right\rceil = 18$ supertasks. On the other hand, the NF rule should produce around $\left\lceil \frac{16}{0.8} \right\rceil = 20$ supertasks. Because, the NF rule produces more supertasks on average, it incurs more reweighting overhead. Furthermore, as task weights increase, the mean utilization of supertasks decreases (on average) more rapidly under the NF rule than under the FF rule. Hence, higher task utilizations favor the use of the FF rule.

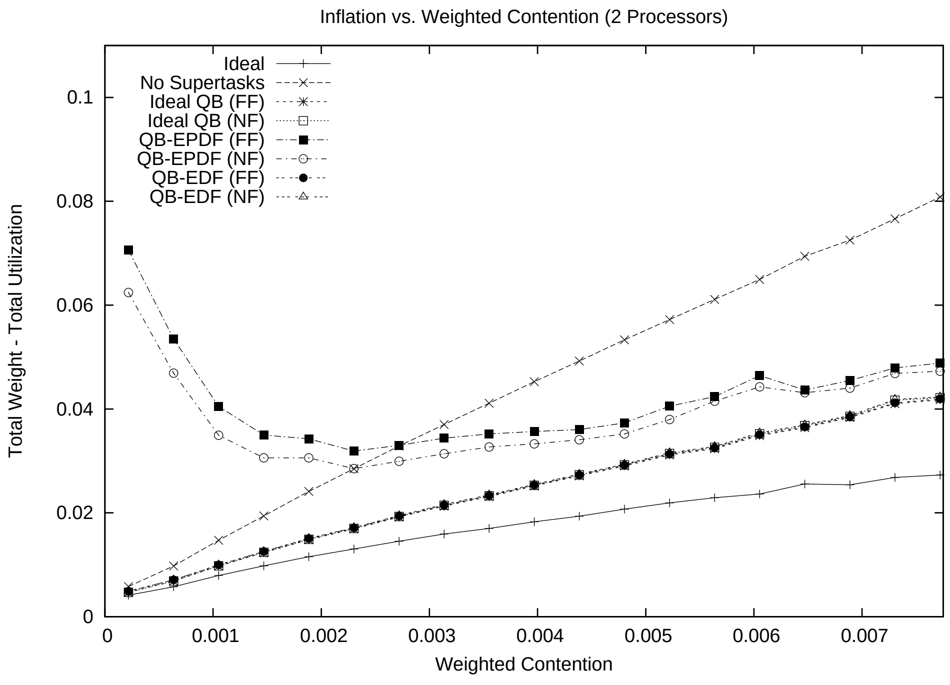
Impact of weighted contention. In the last set of graphs from the first study, inflation is plotted against the weighted object contention. Recall that weighted object contention is a weighted measure of the access frequency to the shared objects. Hence, larger values suggest heavier usage of the shared objects. Consequently, retry overhead is expected to increase with weighted object contention. Unlike the previous graphs, the sample means in these graphs do *not* represent uniform coverage of the sample space. In addition, the number of samples used to produce each mean varies across the graph. (Each mean is based on at least 100 samples.)

Figures 5.30, 5.31, 5.32, and 5.33 show the graphs for the two-, four-, eight-, and sixteen-processor cases, respectively. As before, the "Ideal QB (FF)" and "QB-EDF (FF)" lines (respectively, the "Ideal QB (NF)" and "QB-EDF (NF)" lines) are virtually co-linear. The sampling validity for the eight- and sixteen-processor cases is plotted in Figure 5.34.

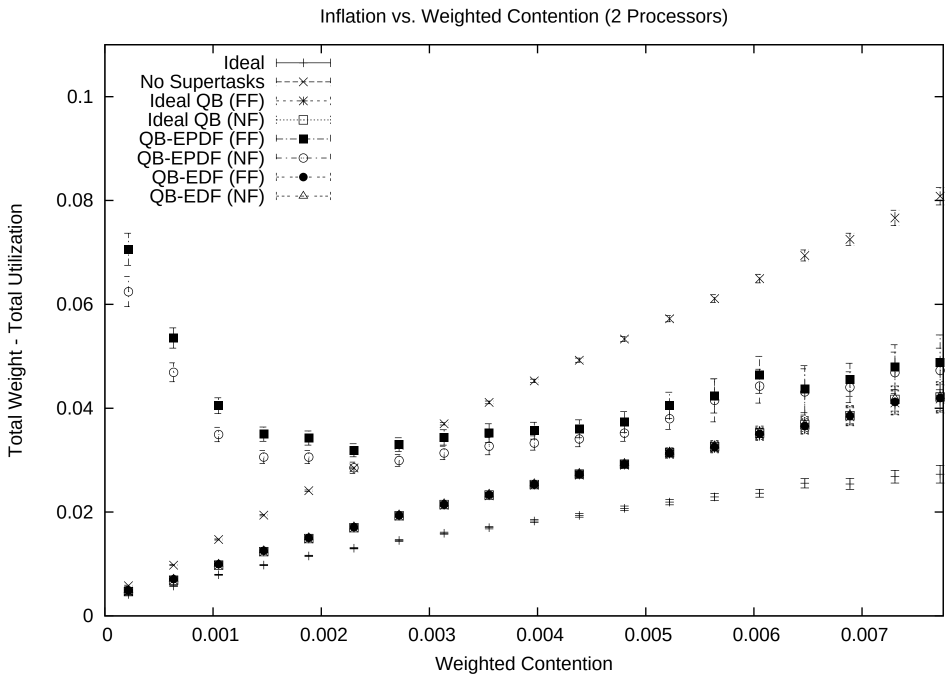
Although the sampling validity is consistent and relatively close to unity in the $M = 8$ case, neither property holds in the $M = 16$ case. In addition, the shape of the sampling validity curve in the $M = 16$ case does not appear to be consistent with the plot of the sample means in Figure 5.33 either. For instance, there is no obvious reason for the dip in sampling validity that occurs around $x = 0.038$. The cause of this inconsistency is not clear and requires further study. (The mean in question is based on approximately 1800 samples; hence, inadequate sampling does not appear to be the cause of this inconsistency.)

Observation 5.7 *Reweighting overhead under the QB-EPDF approach is high when weighted contention is low, but drops off quickly as weighted contention increases.*

Explanation: This is an artifact of non-uniform sampling. Low weighted contention more frequently occurs in small task sets. (This tendency likely holds in practice as well.) Hence, the behavior shown in these graphs is actually the same as that shown earlier in the graphs

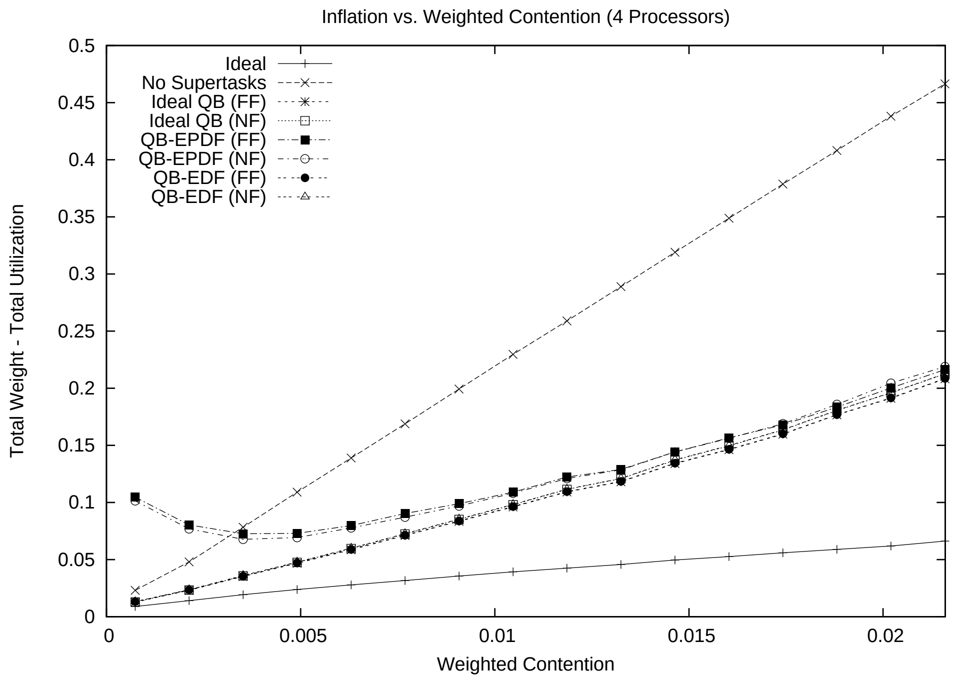


(a) Sample Means Only

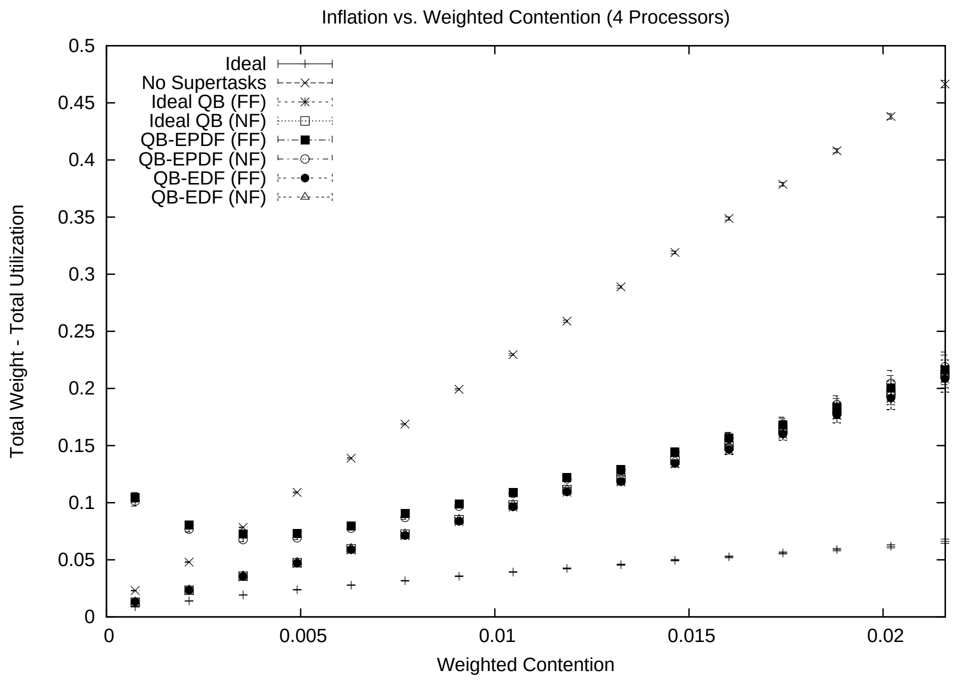


(b) 99% Confidence Interval

Figure 5.30: Plots show how inflation varies as the weighted contention is increased when using lock-free synchronization on two processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

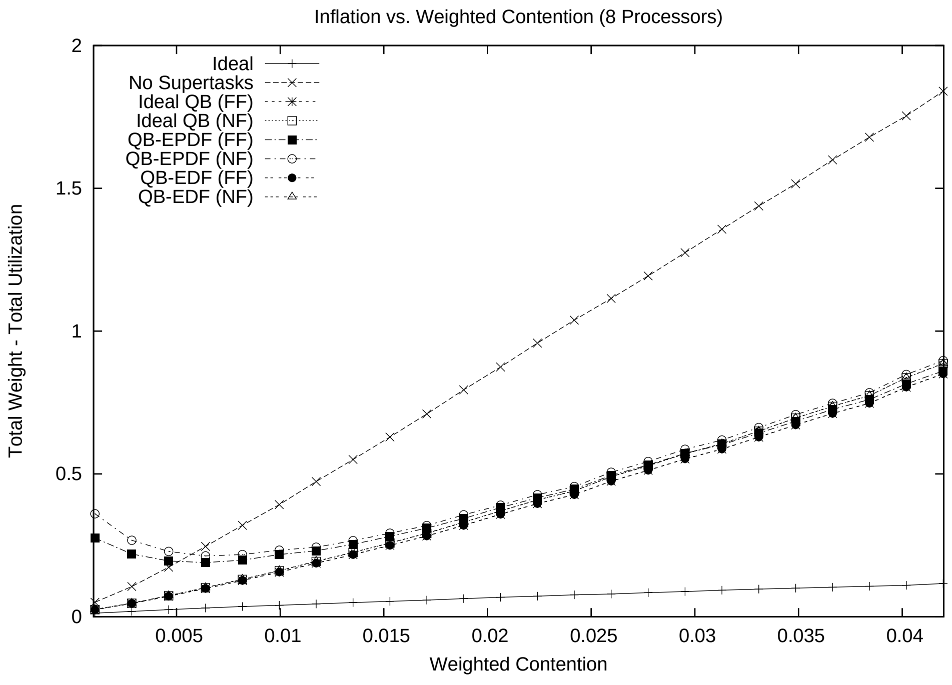


(a) Sample Means Only

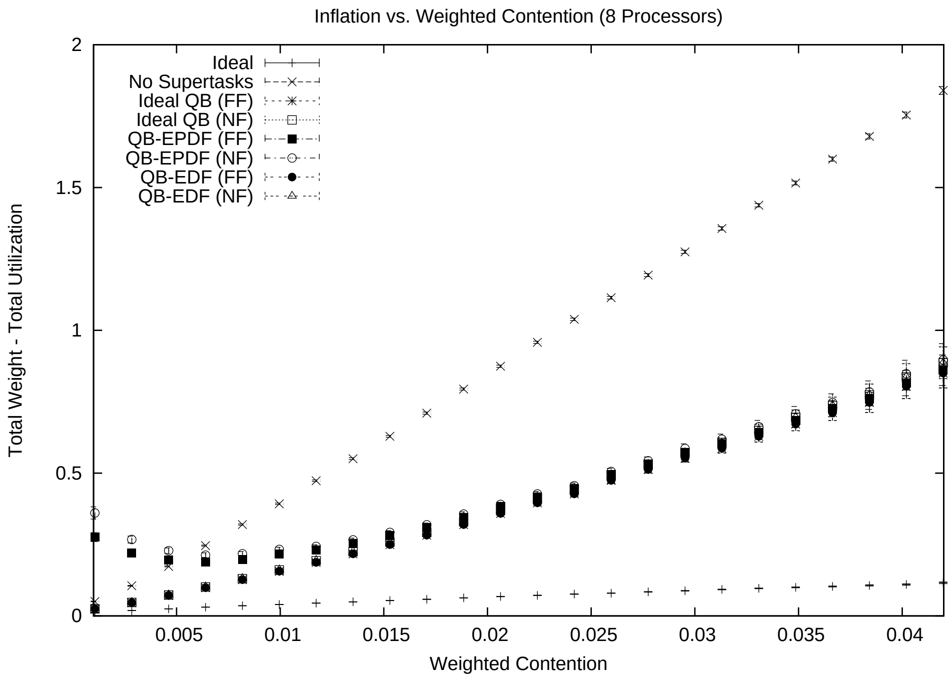


(b) 99% Confidence Interval

Figure 5.31: Plots show how inflation varies as the weighted contention is increased when using lock-free synchronization on four processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

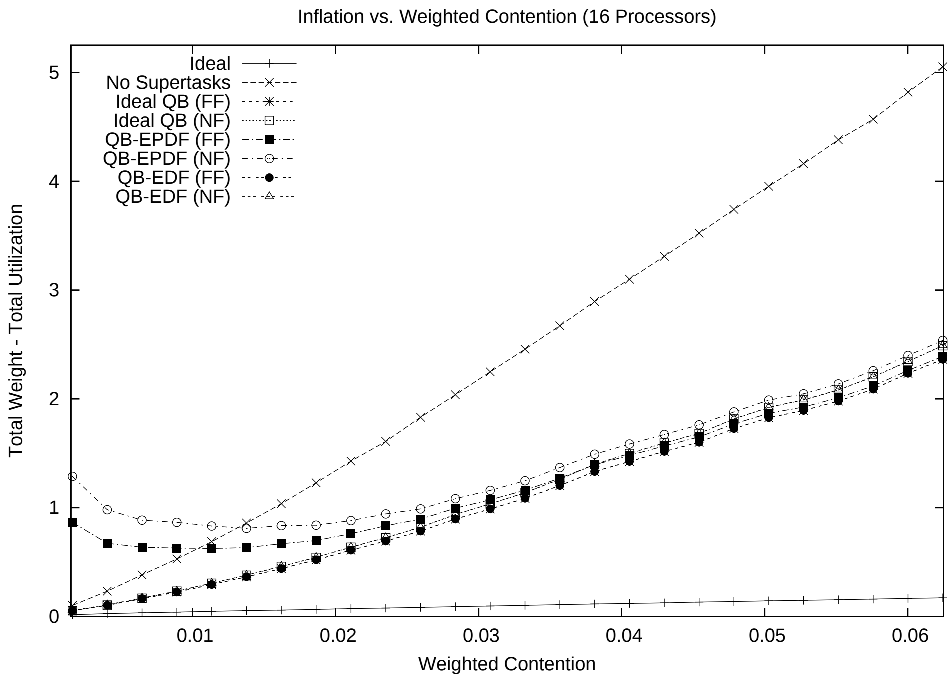


(a) Sample Means Only

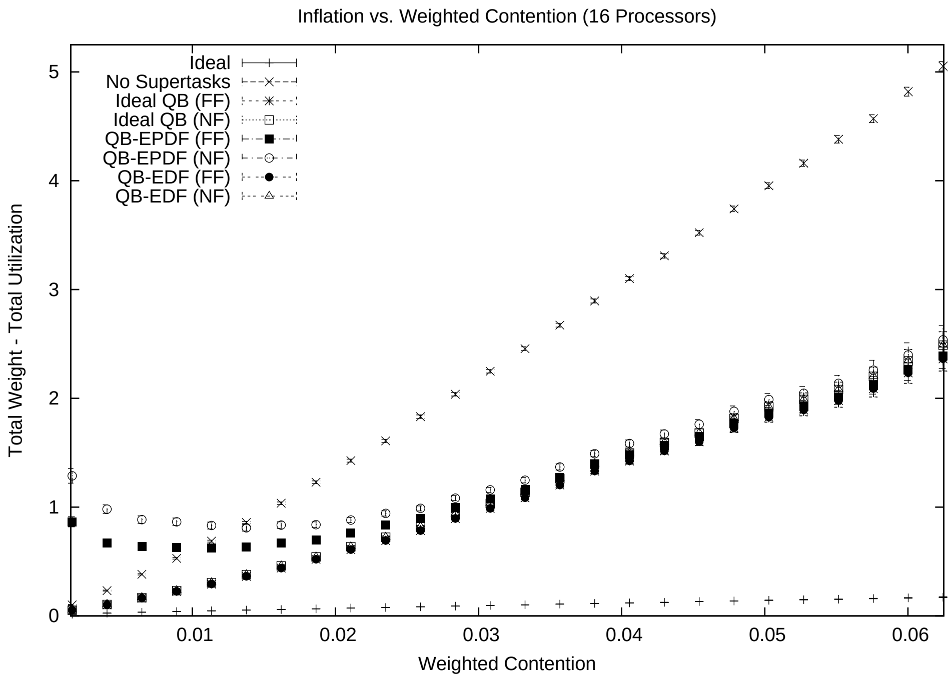


(b) 99% Confidence Interval

Figure 5.32: Plots show how inflation varies as the weighted contention is increased when using lock-free synchronization on eight processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

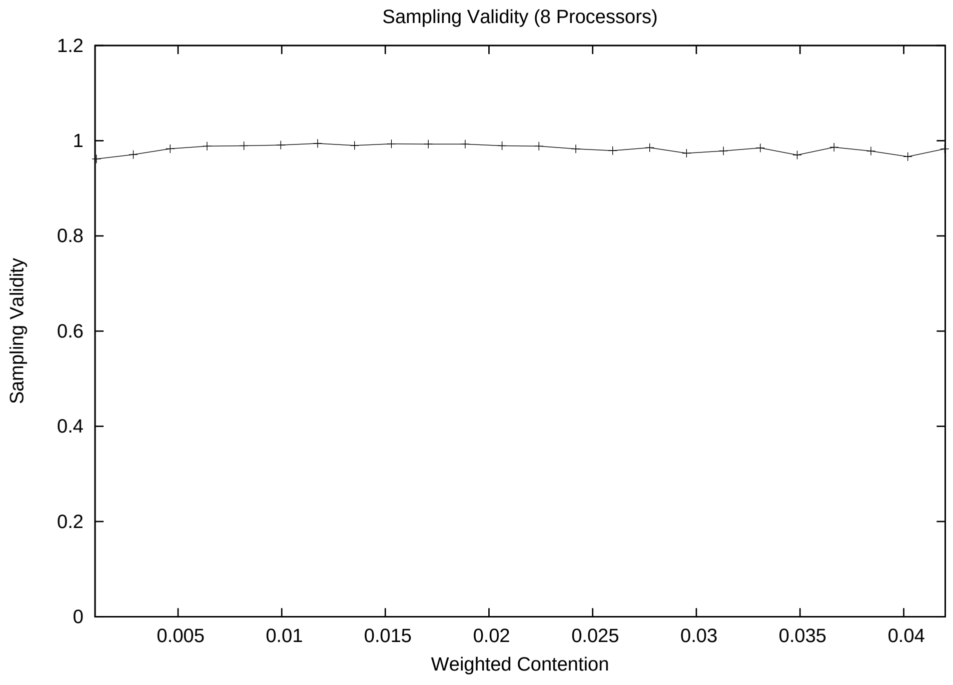


(a) Sample Means Only

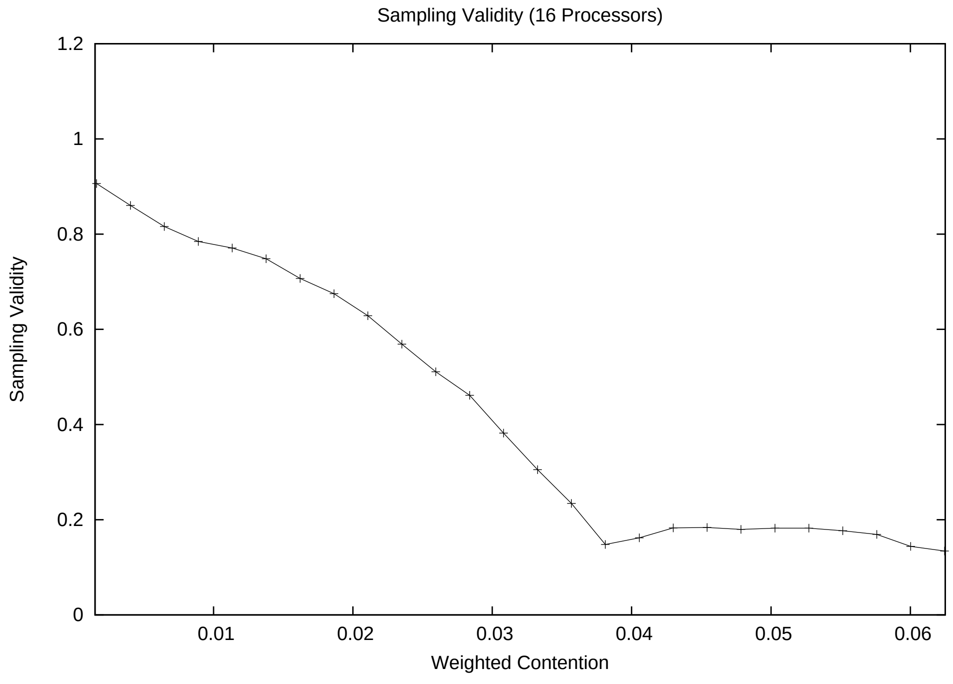


(b) 99% Confidence Interval

Figure 5.33: Plots show how inflation varies as the weighted contention is increased when using lock-free synchronization on sixteen processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) $M = 8$



(b) $M = 16$

Figure 5.34: Plots show the sampling validity for the graphs showing inflation plotted against the weighted contention. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases. The validity was approximately unity throughout the $M = 2$ and $M = 4$ cases. Hence, graphs of those cases are not shown.

of inflation plotted against the task count.

Observation 5.8 *As weighted contention increases, the performance of all supertasking approaches appears to converge.*

Explanation: As weighted contention increases, reweighting overhead becomes insignificant relative to the retry overhead. In addition, the scale of the retry overhead makes the difference between the supertasking approaches appear insignificant by comparison. For instance, in the $M = 16$ graph, the inflation of the supertasking approaches at $x = 0.05$ spans a range of approximately 0.2. On the other hand, the difference between the inflation when using supertasks and not using supertasks is around 2, which is an order of magnitude larger.

5.5.2 Study 2: Per-Quantum Access Count

In this section, we present the details and results of the second study. This study was designed to show how the performance of each approach scales as the $A(T, \ell)$ values increase. To accomplish this, the upper bound imposed on these values, denoted B, was systematically increased and measurements were taken as in the previous study.

Sample space. Because this study allowed for higher contention than the previous one, a more constrained sample space was considered. This was necessary to ensure that the sampling validity of the computed means was sufficiently high to make the results interesting. The sample space was defined as follows:

- $|\tau|$ is in the range $6 \cdot \log_2 M, \ldots, 15 \cdot \log_2 M$;
- $\tau.u$ is in the range $0.2 \cdot M, \ldots, 0.6 \cdot M$;
- $|\Gamma|$ is in the range $\log_2 M, \ldots, 5 \cdot \log_2 M$;

- B is in the range $1, \ldots, 8$;
- $A(T, \ell)$ is in the range $1, \ldots, B$.

All other aspects of the sample space matched those in the previous study. The above ranges were selected through trial and error so that approximately unit sampling validity could be achieved by at least two of the means computed for the $M = 16$ case.

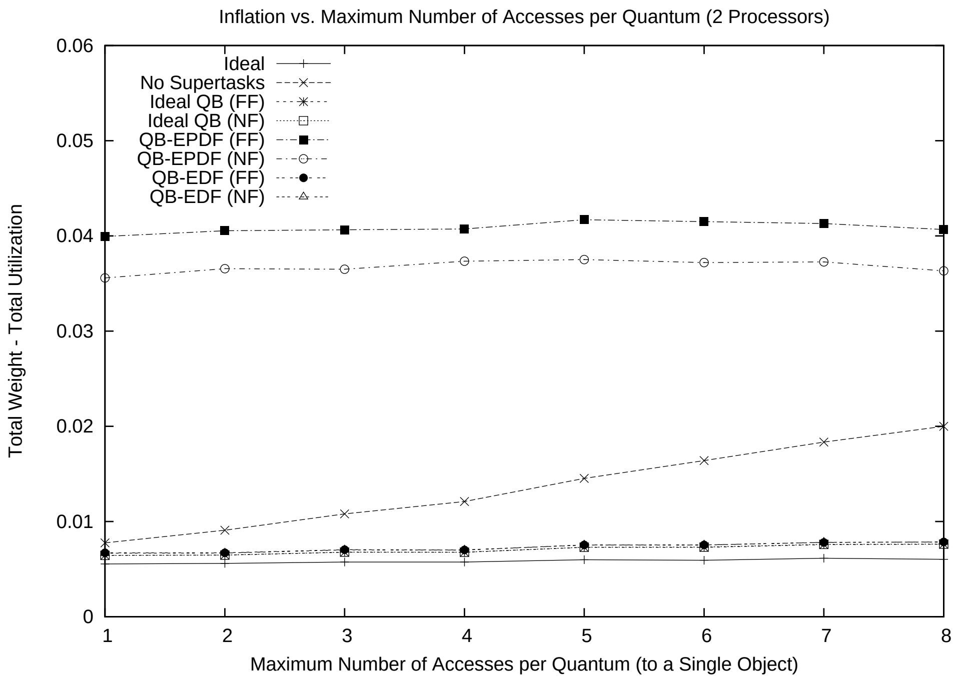
Sampling. Samples were collected as follows:

- B was set to each value in the range $1, \ldots, 8$;
- $|\tau|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 6 \leq x \leq 15\ \}$;
- $\tau.u$ was set to each value in the set $\{\ 0.025x \cdot M \mid 8 \leq x \leq 24\ \}$;
- $|\Gamma|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 1 \leq x \leq 5\ \}$.

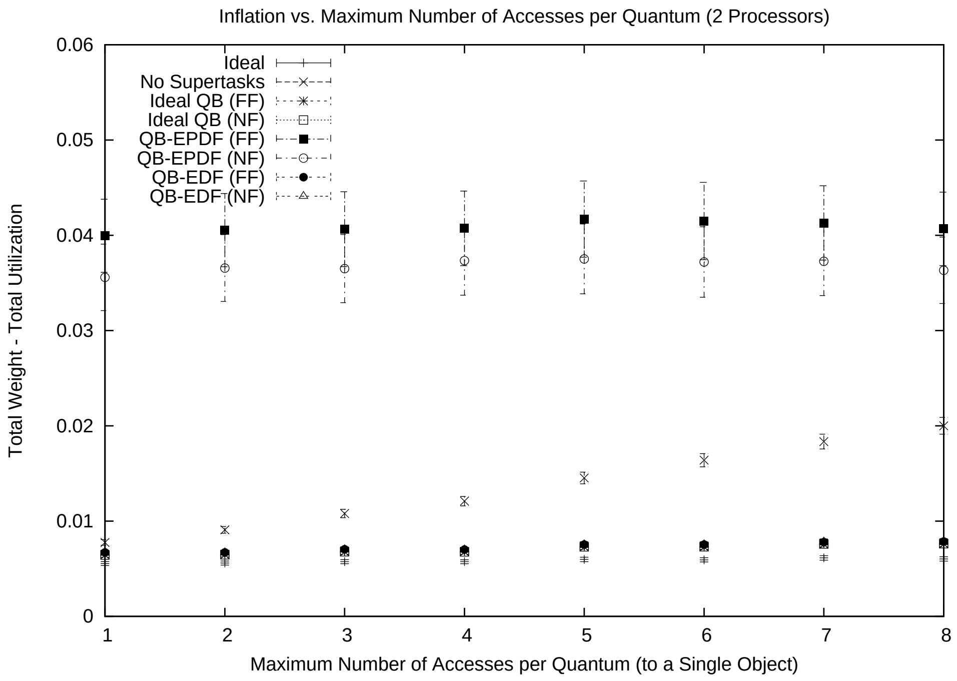
For each combination of the above parameter assignments, two valid component task sets were generated and evaluated. Measurement was performed as in the previous study.

Results. Figures 5.35, 5.36, 5.37, and 5.38 show the inflation plotted against B for two, four, eight, and sixteen processors, respectively. As expected, the "Ideal QB (FF)" and "QB-EDF (FF)" lines (respectively, the "Ideal QB (NF)" and "QB-EDF (NF)" lines) are again virtually co-linear. The sampling validity for the eight- and sixteen-processor cases is shown in Figure 5.39. As shown, sampling validity drops off after $B = 6$ when $M = 8$ and after $B = 3$ when $M = 16$. The resulting skew is apparent in Figures 5.37 and 5.38. Overall, these results agree with those of the first study.

Observation 5.9 *When $M = 2$, the impact of B on the supertasking approaches is negligible.*

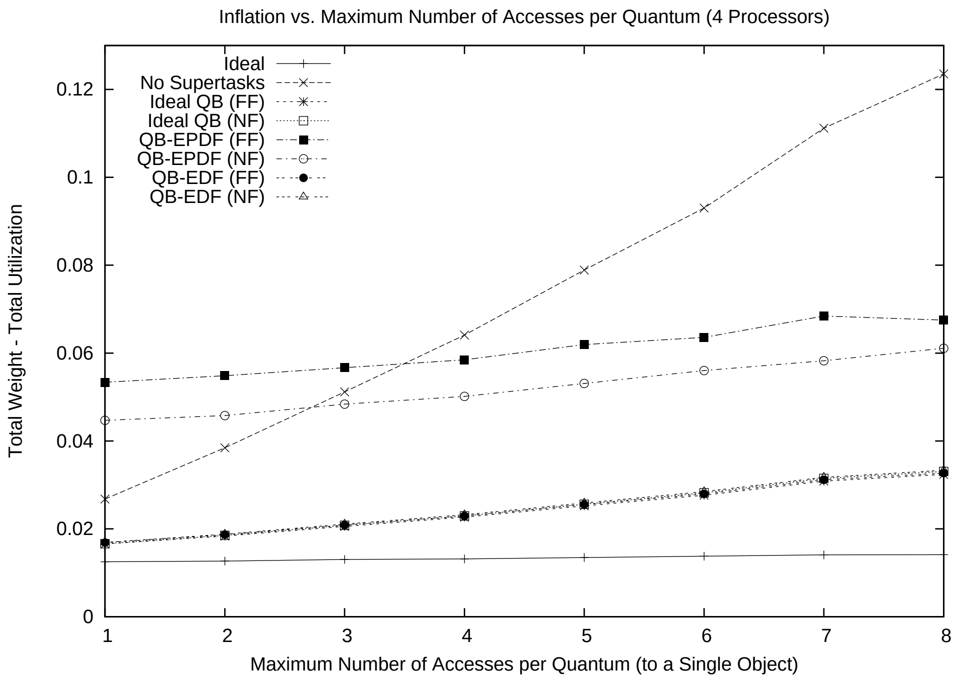


(a) Sample Means Only

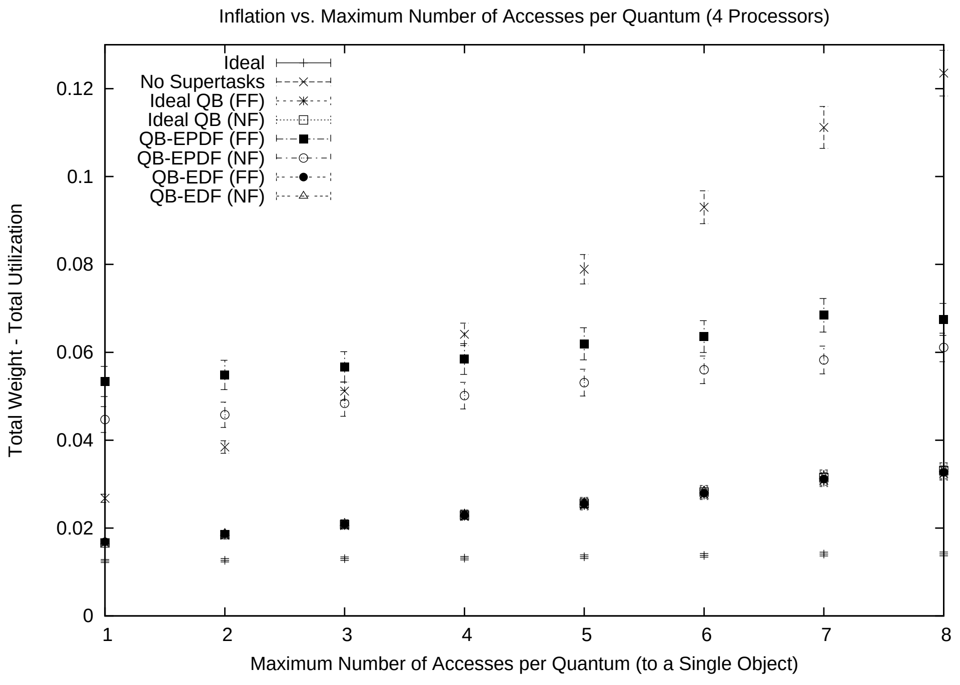


(b) 99% Confidence Interval

Figure 5.35: Plots show how inflation varies as the bound on the per-quantum number of object accesses is increased when using lock-free synchronization on two processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

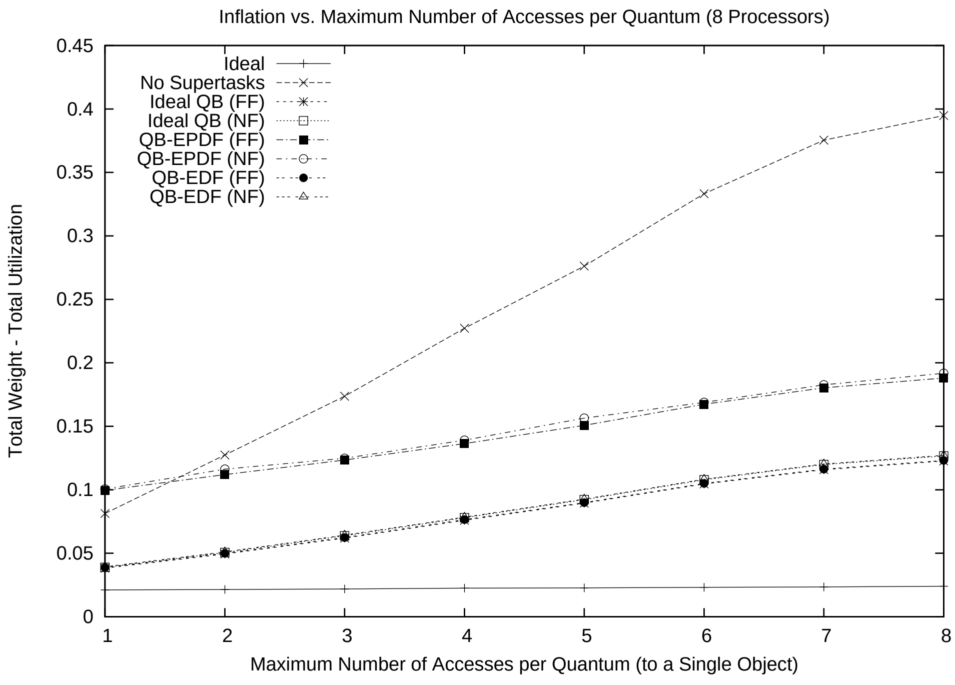


(a) Sample Means Only

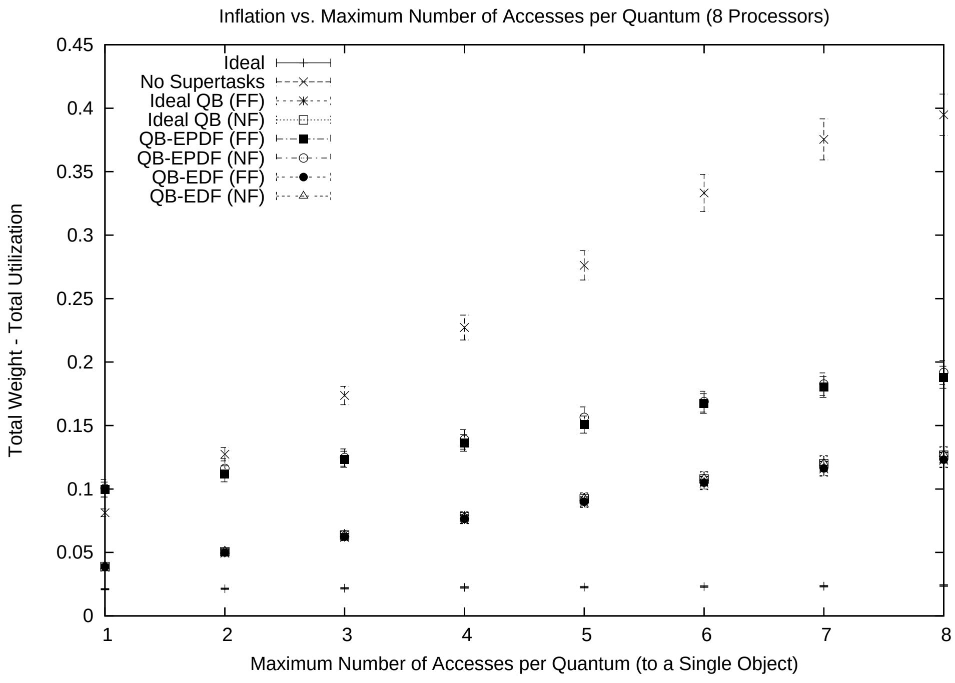


(b) 99% Confidence Interval

Figure 5.36: Plots show how inflation varies as the bound on the per-quantum number of object accesses is increased when using lock-free synchronization on four processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

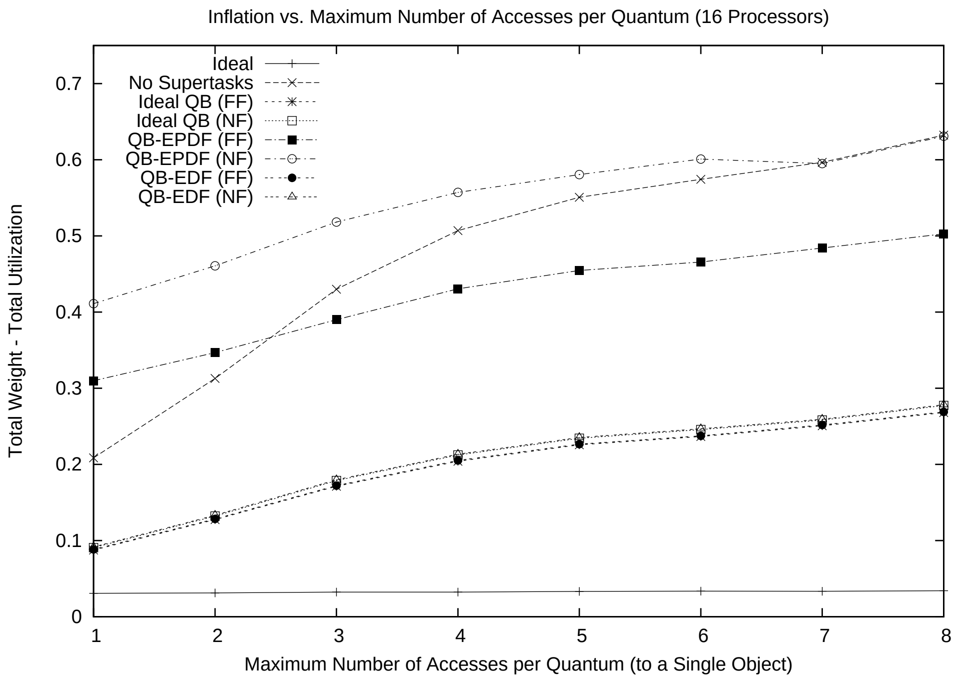


(a) Sample Means Only

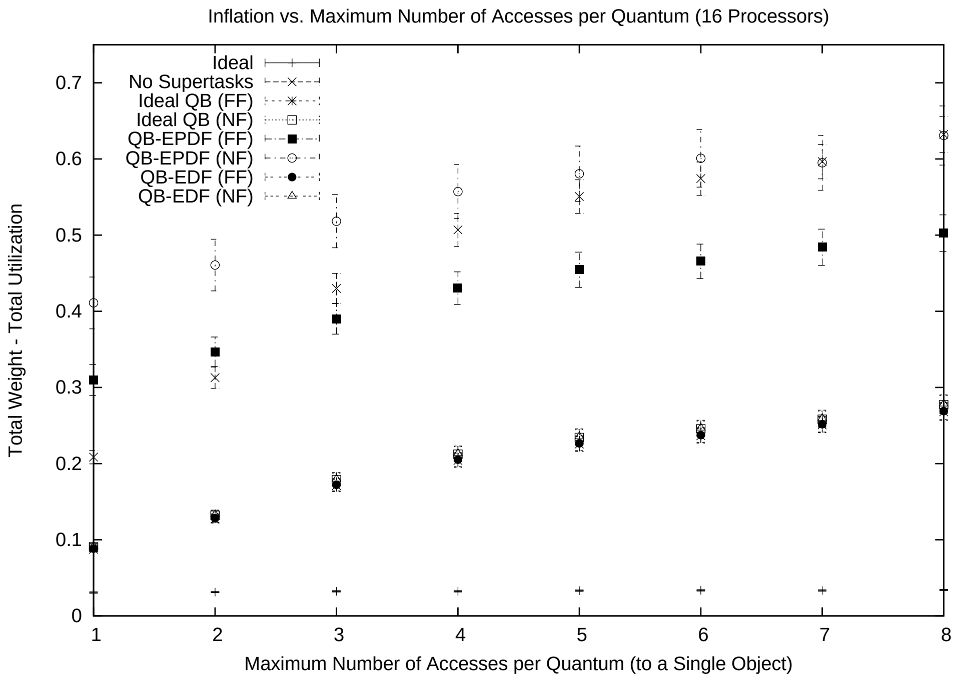


(b) 99% Confidence Interval

Figure 5.37: Plots show how inflation varies as the bound on the per-quantum number of object accesses is increased when using lock-free synchronization on eight processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

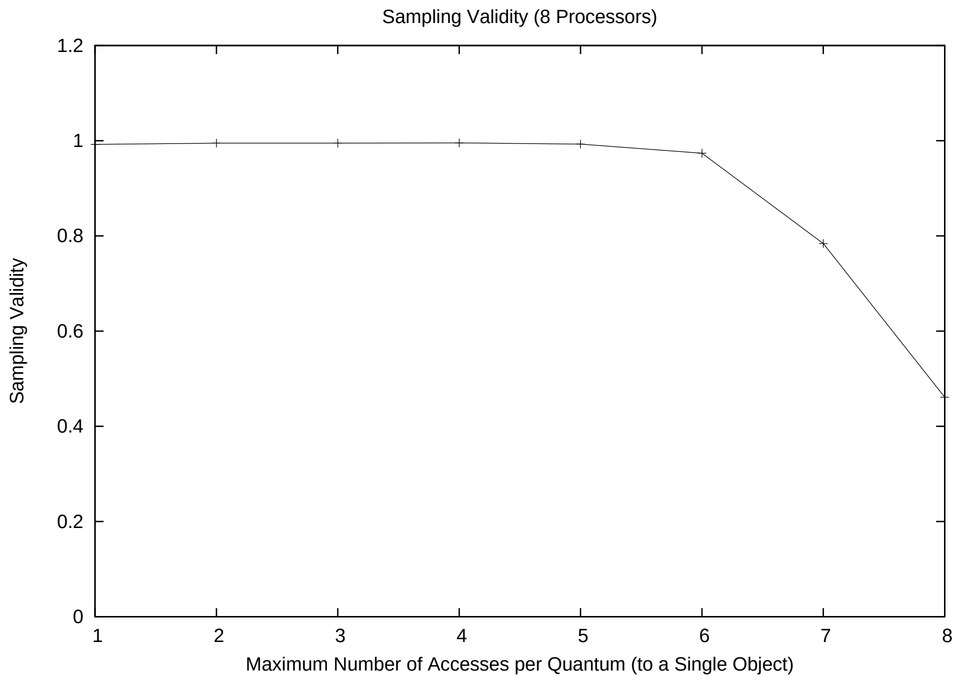


(a) Sample Means Only

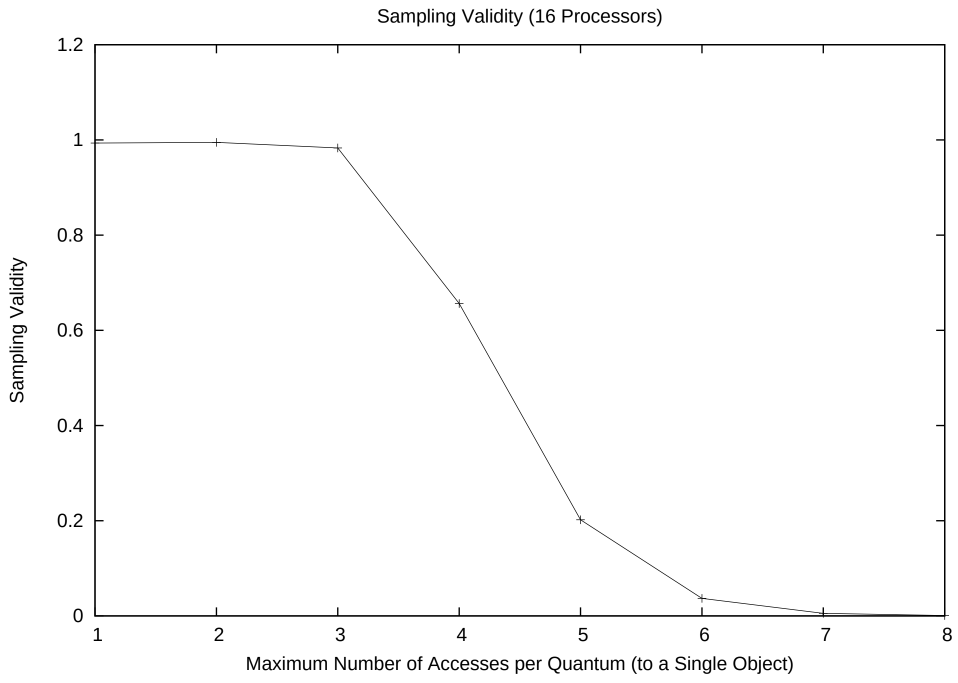


(b) 99% Confidence Interval

Figure 5.38: Plots show how inflation varies as the bound on the per-quantum number of object accesses is increased when using lock-free synchronization on sixteen processors. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) $M = 8$



(b) $M = 16$

Figure 5.39: Plots show the sampling validity for the graphs showing inflation plotted against the bound on the per-quantum number of object accesses. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases. The validity was approximately unity throughout the $M = 2$ and $M = 4$ cases. Hence, graphs of those cases are not shown.

Explanation: On two processors, typically only one or two supertasks are needed. Assigning tasks in any reasonable fashion results in little to no sharing between supertasks. As a result, retry overhead is insignificant relative to other overheads.

Observation 5.10 *When $B < 3$, the reweighting overhead experienced under QB-EPDF tends to outweigh the reduction in retry overhead.*

Explanation: As explained earlier, QB-EPDF introduces the most reweighting overhead of the approaches considered here. The benefit of using supertasks is determined by the relative magnitudes of the retry overhead and the reweighting overhead introduced by the supertasks. For small B, retry overhead is fairly low. As a result, the reweighting overhead introduced by QB-EPDF exceeds the retry overhead avoided by the use of supertasks.

5.5.3 Conclusions

As for the relative performance of the tested approaches, both studies clearly indicate that QB-EDF supertasking is the best choice for reducing inflation. On average, the reweighting overhead introduced by this approach appears negligible compared to the reduction in retry overhead. These studies also suggest that this is not necessarily the case for QB-EPDF supertasking, particularly when the component task counts are low.

As for the assignment heuristic, these studies do not clearly indicate whether using an FF assignment rule or an NF assignment rule will produce the best results. Indeed, the presented results suggest that each performs best for a significant fraction of the sample space. However, the fraction for which the FF rule provides better performance is consistently larger throughout the study. In addition, the FF rule's performance appears to scale better as M increases. Based on these observations, the FF rule appears to be the better overall choice, though the NF rule may outperform it in some cases.

5.6 Summary

In this chapter, we considered the use of lock-free shared objects in a Pfair-scheduled system. After explaining how lock-free synchronization works, we presented a straightforward analysis that uses the structure of Pfair systems to yield tighter bounds on retry overhead than can typically be obtained on multiprocessors. We then explained how quantum-based supertasking can further reduce the retry overhead while also enabling the use of simpler algorithms. To demonstrate the benefits of supertasking, we considered the implementation of a lock-free queue as a case study and conducted an experimental evaluation of a simple assignment heuristic intended to reduce lock-free overhead. As expected, the experimental evaluation suggests that the use of quantum-based supertasking can significantly reduce the inflation experienced by task sets.

CHAPTER 6

Locking Synchronization[1]

In this chapter, we complete the discussion of task synchronization by considering lock-based (or locking) synchronization. Under this form of synchronization, access to each shared resource (or group of shared resources) is controlled by a *lock*. Specifically, in order to access a resource, a task is required to obtain the associated lock and maintain ownership of that lock for the duration of the access. Recall that operations that must access a shared resource are called critical sections. In this chapter, we consider only the use of *mutex semaphores* (or *mutexes*). At most one task can own a mutex at any time. Hence, mutexes ensure *mutual exclusion*[2] among all tasks, *i.e.*, a task that owns the mutex is ensured exclusive access to the resources guarded by that mutex. Consideration of other types of locks is left as future work.

Locking synchronization complicates scheduling because it leads to blocking, *i.e.*, a state in which a task is unable to make progress because it is waiting to access a shared resource. In part, this added complexity stems from the fact that blocking occurs reactively in response to changes in the runtime environment and hence is difficult to characterize off-line. The impact

[1]This chapter is derived from work previously published in the following papers:
[32] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 111-120, 2002.
[33] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23nd IEEE Real-time Systems Symposium*, pages 149–158, 2002.

[2]Indeed, the term "mutex" is an abbreviation of "mutual exclusion."

of blocking is worse under rate-based scheduling than under many other approaches because blocking disrupts the execution rates of tasks. Effectively, blocking forces the execution rate of a task to zero for the duration of the blocking. Hence, the primary focus of this chapter is bounding the duration of blocking and compensating for the rate reductions caused by blocking. We refer to weight inflation stemming from blocking as *blocking overhead*.

This chapter is organized as follows. After summarizing our assumptions in Section 6.1, we briefly discuss the need for locking protocols in rate-based systems in Section 6.2. We then consider the viability of using locking protocols based on inheritance[3] in Section 6.3. Such protocols (*e.g.*, the PIP and its variants [54, 15]) are very effective on uniprocessors. However, as we later explain, they are problematic under Pfair scheduling because their use imposes a dependency between task weights and blocking overhead. In Section 6.4, we present two protocols designed for short critical sections. For the more general case, we present a third protocol in Section 6.5 that places no restrictions on the critical-section lengths. We then present an experimental comparison of these protocols in Section 6.6.

6.1 Assumptions

A *lock-requesting* task T *issues* a request for a lock ℓ by invoking `RequestLock`. Until ℓ is granted, T's request is *pending*. Pending requests for each lock ℓ are assumed to be prioritized using a FIFO policy.[4] Once ℓ is granted to T, we refer to T as the *lock-holding* task until it completes the corresponding call to `ReleaseLock`. If a task U has a pending request for

[3]Recall that "inheritance" occurs when a task temporarily adopts characteristics of tasks that it blocks.

[4]Locking protocols typically prioritize requests by the scheduling priorities of the requesting tasks. However, this is not practical under Pfair scheduling because priorities are not fixed during critical sections. We consider a FIFO prioritization because it facilitates analysis and avoids request starvation. Consideration of other request prioritizations is left as future work.

ℓ while T holds ℓ, then T is said to *block* U. Similarly, if tasks T and U both have pending requests for ℓ, then these requests are *competing* and T and U are *competitors*. Finally, we assume that the overhead of the `RequestLock` and `ReleaseLock` calls are factored into the execution requirements of the appropriate phases of each task.

6.2 Blocking in Rate-based Systems

To motivate the need for locking protocols under Pfair scheduling, consider a task T with a long critical section that always executes at its normal rate, *i.e.*, no attempt is made to speed its critical section. Let C denote the amount of processor time required by the critical section. Since T's average execution rate is given by $Q \cdot T.w$, the execution of its critical section will make the associated lock unavailable to other tasks for approximately $\frac{C}{Q \cdot T.w}$ time units. Hence, tasks with low weights can potentially make locks unavailable for *very* long durations, which suggests a need for techniques that speed the execution of critical sections.

6.3 Viability of Inheritance-based Protocols

The most common approach to designing locking protocols is to rely on *inheritance* to speed the execution of lock-holding tasks [9, 15, 20, 53, 54, 55, 59]. Under such an approach, a lock-holding task takes on (*i.e.*, "inherits") the characteristics of tasks that it either blocks or has the potential to block. The lock-holding task then executes with these characteristics until releasing the associated lock. Unfortunately, the characteristics of Pfair scheduling complicate the use of such techniques (for reasons explained below). Consequently, the protocols presented in later sections intentionally avoid the use of inheritance. In this section, we discuss both the different forms of inheritance that are available under Pfair scheduling and what problems can arise from their use.

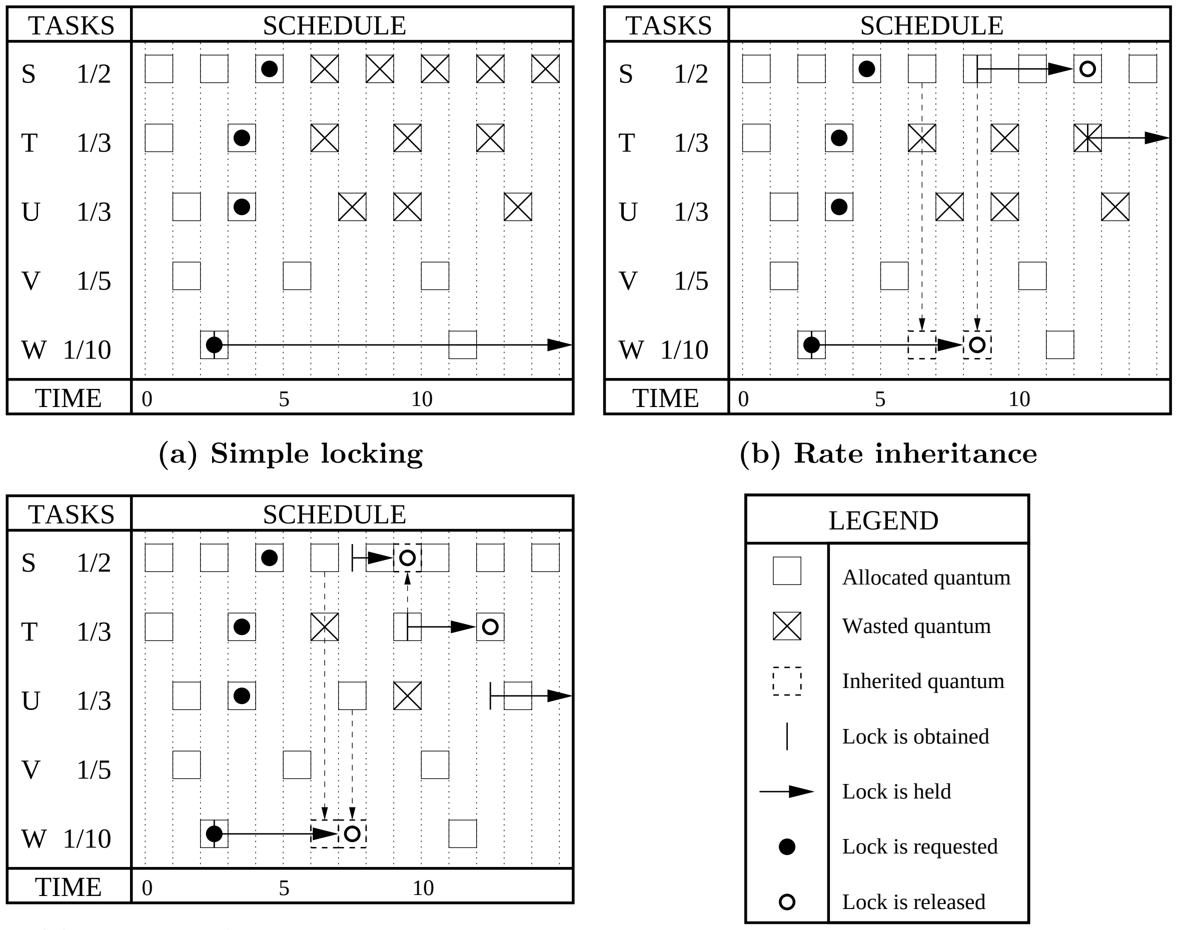
6.3.1 Inheritance in Rate-based Systems

Although conventional inheritance protocols are not directly applicable in Pfair systems, many of the concepts underlying them are applicable. Since tasks are characterized only by weights, two obvious classes of inheritance-based protocols exist. The first class consists of *static-weight* inheritance protocols, under which weight changes are not permitted. Scheduling disruptions caused by blocking can be ameliorated under this restriction by decoupling the choice of which task *executes* in a slot, and which task is actually *charged* for the quantum. More specifically, critical sections can be executed more quickly by temporarily re-routing quanta from a blocked task to the task that is causing the blocking. The second class consists of *dynamic-weight* inheritance protocols, under which weight changes are permitted. Below, we describe some of the more obvious inheritance protocols and the characteristics of each.

Simple locking. Figure 6.1(a) provides a concrete example of how low-weight tasks can cause significant disruptions in Pfair systems. During slot 2, task W obtains a lock that is needed to execute a critical section requiring two quanta. When S, T, and U request this lock in slots 3–4, they are forced to wait until the lock is released by W. Due to W's low weight, this does not happen until sometime after slot 14. In the meantime, all quanta allocated to S, T, and U are wasted (because they are waiting for the lock).[5] We will use this same scenario to illustrate the protocols described below.

Rate inheritance. Rate inheritance is based on priority inheritance [59]. Under this protocol, a lock-holding task T inherits the highest execution rate (*i.e.*, weight) of the tasks

[5]In practice, IS delays would be used to prevent scheduling of the blocked tasks. This is not done in the examples given in this section so that the magnitude of disruptiveness will be apparent.



(a) Simple locking

(b) Rate inheritance

(c) Deadline/allocation inheritance

Figure 6.1: A series of two-processor Pfair schedules for task set consisting of five tasks with weights $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{5}$, and $\frac{1}{10}$, respectively, that may request locks. The insets illustrate the following static-weight locking scenarios: **(a)** simple locking, **(b)** rate inheritance, and **(c)** deadline/allocation inheritance.

that it blocks (if that weight is higher than its normal weight). This protocol is easily implemented by swapping the association of weights to tasks, *i.e.*, the lock-holding task and the blocked task with the highest weight temporarily swap identities (scheduling parameters).

Figure 6.1(b) illustrates rate inheritance. In slot 3, W swaps with T and then with S in slot 4. As a result, W executes within S's quanta in slots 6 and 8.

One problem with rate inheritance is that the task with the highest weight may not have the highest priority at a given instant. For instance, in Figure 6.1(b), the priority of task U

is higher than that of task S in slot 7. The next form of inheritance addresses this concern.

Deadline inheritance. Under deadline inheritance, a lock-holding task swaps identities with whichever blocked task has the highest priority at each time instant. This strategy improves upon rate inheritance, but at the expense of more swapping overhead. Specifically, both approaches may perform a swap when a task is initially blocked, but deadline inheritance may perform additional swaps each time a lock-holding task consumes a quantum.

Figure 6.1(c) illustrates deadline inheritance. In slot 3, W swaps with T. However, this swap is undone in slot 4 when W swaps with S. When W is executed in slot 6 (within S's quantum), U becomes the highest priority task, causing W to take on U's identity.

Allocation inheritance. An alternative form of inheritance that achieves the same result as deadline inheritance is allocation inheritance (which closely resembles bandwidth inheritance, as described in [40]). To avoid the additional swapping overhead produced by deadline inheritance, the quanta allocated to *all* blocked tasks can be redirected to the lock-holding task for the duration of its critical section, *i.e.*, multiple sets of scheduling parameters may be mapped to the same task at a given instant. Although this approach reduces swapping overhead, it requires support for many-to-one mappings from parameter sets to tasks. In addition to the overhead required to maintain such a mapping, care must also be taken to ensure that each lock-holding task utilizes only one quantum in each slot.

Figure 6.1(c) also illustrates the schedule as it would appear under allocation inheritance. In slot 3, the allocations of both U and V are redirected to W. S's allocation is then redirected to W in slot 4. Hence, at the start of slot 6, W is actually associated with four different sets of parameters (*i.e.*, those normally associated with S, U, V, and W). In slot 6, two of these sets are scheduled. However, W can only utilize one of the two quanta.

The potential for parallel allocation of quanta is a fundamental shortcoming of all forms of static-weight inheritance. To avoid this problem, subtask releases can be selectively postponed using IS delays. Unfortunately, such a reactive approach is difficult to analyze. Alternatively, a dynamic-weight scheme like that described next could be used.

Weight inheritance. Under weight inheritance, the lock-holding task adds to its weight the weight of each task that it blocks. (Weight inheritance also bears resemblance to bandwidth inheritance.) By changing its weight, the lock-holding task is able to effectively serialize its quanta in that it avoids the simultaneous allocation of multiple quanta. Weight inheritance is not illustrated in Figure 6.1 for reasons explained in the next section.

6.3.2 Problems of Inheritance-based Protocols

There are two primary problems that limit the effectiveness of inheritance-based schemes in Pfair systems. First, as explained in the previous section, dynamic-weight inheritance is the most effective form of inheritance in concept, due to the ability to serialize inherited processor time. Unfortunately, under Pfair scheduling (and likely under any other real-time rate-based approach), instantaneous weight changes must be *forbidden* to ensure schedulability [62]. Although improved reweighting techniques have been proposed [2, 63], these approaches either lack worst-case predictability [2] or place additional restrictions on the system [63]. Accounting for delayed weight changes complicates worst-case analysis and necessitates the use of overly conservative weights.

Unfortunately, even static-weight schemes are somewhat impractical due to the dependency between blocking overhead and task weights. Consider two tasks, T and U, that share a common lock. To account for the possibility that T blocks U, $U.w$ must be increased based on the worst-case blocking that can occur, which is determined by $T.w$. However, changing $U.w$ changes the worst-case duration for which U can block T, which necessitates a compensatory change in $T.w$. Hence, assigning weights under an inheritance protocol is an optimization problem. Finding an optimal solution is almost certainly an intractable problem.

Despite these issues, inheritance-based schemes remain an interesting avenue for future study, particularly for soft real-time systems. In addition, recent work [13] on a "fast" reweighting approach for Pfair scheduling, called *PROfairness*, shows considerable improvement over existing techniques. Unfortunately, due to significant differences between the system model considered in [13] and that considered here, it is unclear whether PROfairness (or the concepts underlying it) can be used to produce an effective dynamic-weight locking protocol for hard real-time systems.

6.4 Supporting Short Critical Sections

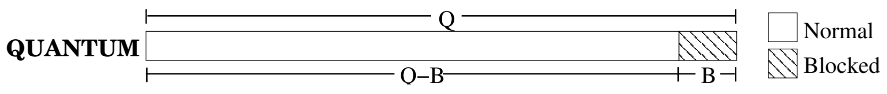
In this section, we present locking protocols for use with locks that guard short (relative to Q) critical sections. In experiments conducted by Ramamurthy [56] on a 66 MHz processor, critical-section durations for a variety of common objects (*e.g.*, queues, linked lists, *etc.*) were found to be in the range of tens of microseconds. On modern processors, these operations will likely require no more than a few microseconds. Since quantum sizes typically range from hundreds of microseconds to milliseconds, we expect the protocols described in this section to be widely applicable. In the next section, we consider support for longer critical sections.

6.4.1 Concept

When some task T is granted a lock ℓ, each task U with a pending request for ℓ is necessarily delayed for the duration of T's critical section. However, this unavoidable delay may be amplified drastically if T is preempted. Any such preemption effectively *lengthens* T's critical section. As explained earlier, this is particularly problematic under Pfair scheduling because each task is executed at an approximately uniform rate. Hence, if T has a low weight and its critical section is preempted before completing, then a lengthy delay is likely.

Fortunately, when using quantum-based scheduling with a sufficiently long quantum, the preemption of lock-holding tasks is avoidable. In particular, when T's critical-section duration is shorter than Q, its execution may span at most two quanta, which implies that T can be preempted at most once before relinquishing the lock. On the other hand, if T's critical section always begins *sufficiently early* within a quantum, then T will always release the lock before the next slot boundary. In practice, critical sections can be *forced* to start "sufficiently early" within a quantum by automatically blocking lock requests for critical sections that are at risk of being preempted. Whereas a task can only be blocked due to the unavailability of the requested lock in a traditional system, a task may be blocked due to either unavailability or the timing of the request under this approach.

To illustrate this approach, consider a lock request made by some task T, for which the associated critical section has worst-case duration e $(< Q)$. The proposed approach introduces an interval of *automatic blocking* (of the lock request in question) at the end of each quantum, as illustrated below. We refer to this interval as a *blocking zone* and denote its length by B.



T's request is *safe* if the quantum is *not* within a blocking zone of that request, and *unsafe* otherwise. Additionally, T's request is said to be *active* if T is currently scheduled, and *inactive* otherwise. As long as $B \geq e$, T's critical section can be guaranteed to execute completely within a single quantum by simply granting the requested lock only if T's request is both safe and active. In this way, the nonpreemptivity inherent under quantum-based scheduling can be exploited to support short critical sections. Indeed, this approach is applicable under any scheduling policy that uses a quantum-based model.

6.4.2 Zone Placement

There are several ways to apply the previous concept when designing a locking protocol. We refer to all protocols based on this concept as *zone-based* protocols. In this section, we briefly discuss some of the different rules for determining the placement of zones.

Request-based zones. The most aggressive approach is to provide supplementary information with each issued lock request. Whether the request is within its blocking zone at a given time can then be determined based upon this information and the state of the system. This approach introduces additional overhead into the lock-management algorithms, but also provides the most flexibility at runtime and ensures that no critical section is delayed longer than necessary to ensure safety. For these latter reasons, we focus on this approach when performing analysis later.

Group-based zones. Rather than associating a unique blocking zone with each request, each blocking zone be associated with a group of critical sections. Under such an approach, B must be sufficiently long to prevent the preemption of each critical section in the group. The primary benefit of using such groupings is to simplify runtime accounting and to reduce the memory overhead of the protocol. To illustrate the former benefit, consider two of the most obvious groupings: group according to the requesting task or group according to the requested lock. In both cases, it is possible to mark the start of a blocking zone by simply setting a bounded number of interrupts to occur within each quantum, as explained below.

First, consider using a single blocking zone for all critical sections of a given task. When a task is scheduled, an interrupt[6] that marks the start of that task's blocking zone can also be scheduled to occur on the assigned processor. Accounting for the overhead of the blocking zone is simplified in this case since each quantum requires only one such interrupt.

Now, consider using a single blocking zone for all critical sections guarded by a given lock.[7] Under this approach, up to $|\Gamma|$ interrupts are needed in each quantum to signal the start of each lock's blocking zone.[8] Although the use of more interrupts produces more overhead, the degree of unnecessary automatic blocking will likely be lower than when using task-based groups. This follows from the fact that locks are often used to synchronize operations of comparable complexity (and hence duration). On the other hand, the durations of critical sections of a given task may vary considerably.

Impact of timer precision. Real timers typically cannot generate interrupts at arbitrary times. Instead, expiration times (or delays) must be given as multiples of a timer-specific time unit. This unit defines the *precision*[9] of the timer.

Due to this practical restriction, it may be necessary to start blocking zones prematurely.

[6]We do not consider whether a hardware or software interrupt (or some other form of signaling mechanism) is most appropriate here, since this will likely depend on the target architecture.

[7]We previously considered this approach in [33].

[8]Only locks that may be requested by the scheduled task need to be considered. Unnecessary interruptions can be avoided by providing the operating system with a list of locks that each task may request.

[9]This is also called the *resolution* or *granularity* of the timer.

Specifically, each zone that is initiated by an interrupt should be started at the latest possible timer expiration time that occurs before the ideal starting time of the zone. This conservative placement is always guaranteed to be within u of the ideal placement, where u denotes the timer's minimum time unit. Despite this, when u reflects a significant fraction of a quantum, timer error may be prohibitively high.

Indeed, there are several more issues relating to the use of realistic timers, including jitter[10] and the handling of interrupts. Because we primarily focus on the use of request-based zones in the remainder of the chapter, we forgo further discussion of these issues and leave them as topics for future work.

6.4.3 Requirements and Additional Notation

In the sections that follow, we present two protocols based on the notion of blocking zones. These protocols differ only in their handling of delayed requests found at slot boundaries. The first protocol maintains these requests, while the second protocol discards them, forcing the requesting tasks to re-issue the requests later. Hence, inactive requests may exist under the first protocol, but not under the second protocol. To demonstrate that these protocols can be efficiently implemented, we present a sample implementation (in pseudocode) for each protocol. In this section, we begin by describing the base requirements of these protocols and by defining notational conventions used in the analysis presented later.

Requirements. The following requirements must be satisfied by a zone-based protocol.

(R1) A requesting task must eventually be granted the lock.

(R2) Each critical section must execute completely within a single quantum.

[10]The impact of jitter on the timing of an interrupt was discussed earlier in Chapter 3, in the context of the quantum interrupt. This issue can be similarly addressed here.

(R2) requires that the zone associated with each request be at least the maximum duration of the request's critical section.

Notation. First, let $T^{[i]}.B$ denote the length of the blocking zone associated with the request of a locking phase $T^{[i]}$. This length is assumed to be sufficiently long to satisfy (R2). The following shorthand notations bound the worst-case blocking caused by each task and each supertask within a single quantum.

- Let $T.\hat{e}(\ell) = \max\left\{ T^{[i]}.e \;\middle|\; T^{[i]}.R = \ell \right\}$. $T.\hat{e}(\ell)$ is the duration of the longest critical section of T requiring ℓ. This value is also an upper bound on the duration of blocking caused by T's requests for ℓ in a single quantum.

- Let $\mathcal{S}.\hat{e}(\ell) = \max\{ T.\hat{e}(\ell) \mid T \in \mathcal{S} \}$. $\mathcal{S}.\hat{e}(\ell)$ is the duration of the longest critical section of any task T in supertask $\mathcal{S}$ requiring ℓ. This value is also an upper bound on the duration of blocking caused by requests for ℓ issued from within $\mathcal{S}$ in a single quantum.

- Let $Q^m(T,\ell) = \mathsf{maxsum}_{m(M-1)}\{ U.\hat{e}(\ell) \mid U \neq T \}$. $Q^m(T,\ell)$ is an upper bound on the amount of time required to service competing requests of a single request of task T across m quanta while respecting a FIFO prioritization.

- Let $Q^m(\mathcal{S},\ell) = \mathsf{maxsum}_{m(M-1)}\{ T.\hat{e}(\ell) \mid T \notin \mathcal{S} \}$.[11] $Q^m(\mathcal{S},\ell)$ replaces $Q^m(T,\ell)$ when quantum-based supertasks are used.

- Let $I(T,\ell) = \sum_{U \neq T} U.\hat{e}(\ell)$. $I(T,\ell)$ is an upper bound on the amount of time required to service all competing requests of a single request of task T while respecting a FIFO

[11]Owing to an editing error, the definition of $Q^m(\mathcal{S},\ell)$ given in [33] is incorrect. However, this error does not impact the results presented in that paper since the formula is stated but never used.

prioritization. Notice that $\lim_{m \to \infty} Q^m(T, \ell) = I(T, \ell)$.

- Let $I(\mathcal{S}, \ell) = \sum_{T \notin \mathcal{S}} T.\hat{e}(\ell)$. $I(\mathcal{S}, \ell)$ replaces $I(T, \ell)$ when quantum-based supertasks are used. Again, notice that $\lim_{m \to \infty} Q^m(\mathcal{S}, \ell) = I(\mathcal{S}, \ell)$.

The definition given above for $Q^m(\mathcal{S}, \ell)$ is actually conservative, owing to the fact that quantum-based supertasking imposes the following two restrictions:

1. Each supertask can have at most one active interfering request in each quantum.

2. Tasks in a supertask cannot interfere with each other.

Although the above bound does consider Restriction 2, Restriction 1 is ignored because its consideration would not produce a simple formula. Alternatively, a polynomial-time algorithm like that sketched in Figure 6.2 can be used to compute the exact values of $Q^m(\mathcal{S}, \ell)$.

In the analysis that follows, a lock-requesting task T may be blocked due to two sources. First, T is *actively blocked* when one of its competitors is granted the lock before T and executes its critical section while T's request is both safe and active. A lock cannot be granted during a blocking zone (*i.e.*, when T is unsafe), even if no competing requests are present. Blocking caused solely by blocking zones is called *automatic blocking*.

6.4.4 Skip Protocol: Description and Analysis

Under the Skip Protocol (SP), delayed requests for locks are retained across slot boundaries, thereby ensuring that the FIFO prioritization is respected across slots. Due to this latter fact, the SP ensures that each competing task can block a request at most once. The primary cost of this approach is that the scheduler must either save and later restore the state of delayed requests at slot boundaries (*i.e.*, as the set of executing tasks changes), or simply ignore (*i.e.*,

```
**for each** $\mathcal{T} \neq \mathcal{S}$ **do**
    Create an empty list for $\mathcal{T}$;
    **for each** $T \in \mathcal{T}$ such that $T.\hat{e}(\ell) > 0$ **do**
        Add $T.\hat{e}(\ell)$ to $\mathcal{T}$'s list
    **od**;
    Sort $\mathcal{T}$'s list so that values are in non-increasing order
**od**;
$k := 1$;
$q := 0$;
**while** non-empty lists exist **do**
    $h := 0$;
    Let $L$ denote the number of non-empty lists;
    Select the $\min(M-1, L)$ non-empty lists with largest head values;
    **for each** selected list **do**
        Add head value to $h$;
        Remove head value from the list
    **od**;
    $q := q + h$;
    $Q^k(\mathcal{S}, \ell) := q$;
    $k := k + 1$
**do**;
Set $Q^m(\mathcal{S}, \ell) = q$ for all $m \geq k$
```

Figure 6.2: Algorithm for computing $Q^m(\mathcal{S}, \ell)$.

skip over) inactive requests when deciding which task will be granted a lock. In either case, this extra work translates into increased runtime overhead.

Due to FIFO prioritization and the fact that locks are not held across slot boundaries, request starvation is not possible. Hence, (R1) holds under the SP as long as $T^{[i]}.e < Q$ also holds for each locking phase $T^{[i]}$. To ensure that (R2) is satisfied, $T^{[i]}.B > T^{[i]}.e$ must also hold for each such locking phase to ensure that each critical section completes by the slot boundary that follows its start. Hence, we assume that the following relationship holds for each such locking phase $T^{[i]}$.

$$Q > T^{[i]}.B > T^{[i]}.e \tag{6.1}$$

(The $Q > T^{[i]}.B$ inequality is a trivial requirement that is needed to ensure that $T^{[i]}$ is considered safe at the start of each quantum.)

Basic analysis. As in the previous chapter, we present theorems and proofs that equate locking phases, assumed to execute under specified conditions, to non-locking phases, which can then be used with Lemmas 3.4–3.7. The next theorem handles the use of the SP.

Theorem 6.1 *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$, where ℓ is managed by the Skip Protocol and all blocking zones of ℓ satisfy* (6.1)*, phase $T^{[i]}$ is equivalent to*

$$\mathbf{C}\left(\emptyset, T^{[i]}.e + I(T,\ell) + \left(\left\lfloor \frac{I(T,\ell)}{Q - T^{[i]}.B} \right\rfloor + 1\right) \cdot T^{[i]}.B\right)$$

in a system that does not include supertasks.

Proof. Due to the FIFO prioritization being respected across slots, $I(T,\ell)$ is a trivial upper bound on the active blocking experienced by $T^{[i]}$. In the worst case, competitors of T execute for at least $Q - T^{[i]}.B$ time units within each quantum in which T is scheduled, and hence hold the lock until at least the start of T's blocking zone. In this case, which is depicted in Figure 6.3, T is preempted no more than $\left\lfloor \frac{I(T,\ell)}{Q-T^{[i]}.B} \right\rfloor$ times after issuing the request before ℓ is granted. (Since the blocking zone in the slot in which T is granted the lock is not counted, we use a floor expression rather than a ceiling.) In addition, if T's request is initiated at the start of a blocking zone, then T is blocked by an additional blocking zone. Hence, the duration of phase $T^{[i]}$ is at most the sum of the durations of **(i)** $T^{[i]}$'s critical section, **(ii)** $I(T,\ell)$ units of active blocking, and **(iii)** $\left(\left\lfloor \frac{I(T,\ell)}{Q-T^{[i]}.B} \right\rfloor + 1\right) \cdot T^{[i]}.B$ units of automatic blocking. $\square$

Improved analysis. Stronger restrictions, like (R3) given below, produce less overhead.

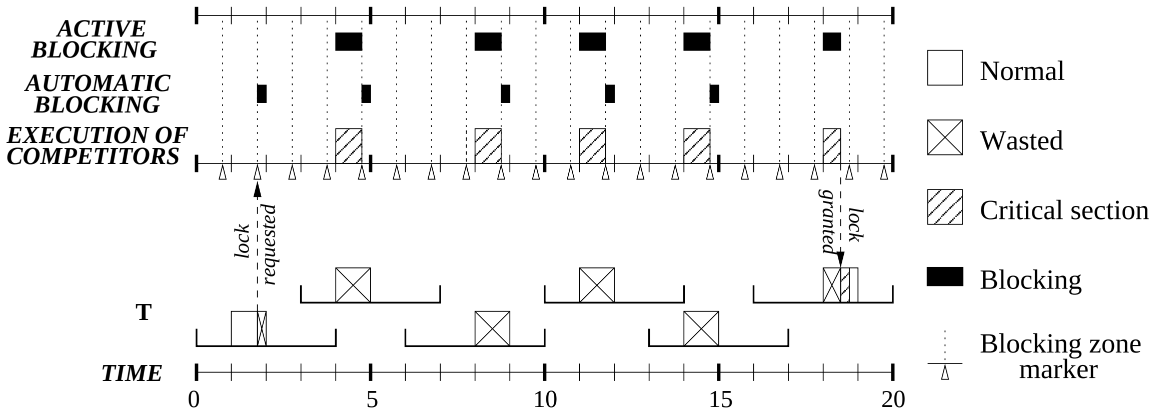


Figure 6.3: Worst-case blocking under the SP for a lock-requesting phase $T^{[i]}$ when $I(T, \ell) = 4.239Q$ and $T^{[i]}.B = 0.1Q$. T's request is initiated at the start of a blocking zone and competing requests execute only when T is both scheduled and not within a blocking zone. The blocking of each form experienced by T is shown with black bars at the top of the figure. The number of blocking zones crossed (while scheduled) is $1 + \lfloor \frac{4.239Q}{Q-0.1Q} \rfloor = 1 + \lfloor 4.71 \rfloor = 5$.

(R3) All active requests present at the start of a slot are serviced within that slot.

Notice that (R3) is a stronger requirement than both (R1) and (R2). Unlike (R1) and (R2), this restriction bounds the total time required by any combination of competing requests (up to M) that are serviced within the same slot. The measurements of Ramamurthy, cited earlier, suggest that (R3) can be expected to hold in many cases.

For (R3) to be satisfied with respect to a request for ℓ made by a locking phase $T^{[i]}$, the total processing time required by T's worst-case mix of $M - 1$ competing requests for ℓ (*i.e.*, $Q^1(T, \ell)$), must be strictly less than the length of time over which T's request is safe (*i.e.*, $Q - T^{[i]}.B$). The condition given below is sufficient to ensure that (R3) holds for a lock ℓ:

$$\left(\forall T : T.\hat{e}(\ell) > 0 : Q^1(T, \ell) \leq Q - \max \left\{ \ T^{[i]}.B \ \middle| \ T^{[i]}.R = \ell \ \right\}\right) \tag{6.2}$$

The $T.\hat{e}(\ell) > 0$ constraint in the above condition simply restricts attention to tasks with critical sections that require ℓ.

Unfortunately, some systems may not satisfy the condition given in (6.2). However, for such systems, it is still possible to apply a less-strict condition to *individual* critical sections in order to produce blocking estimates tighter than those provided by Theorem 6.1. Theorem 6.2, given next, considers

$$\frac{1}{m} Q^m(T, \ell) \leq Q - T^{[i]}.B, \tag{6.3}$$

which focuses on a specific phase $T^{[i]}$ of a task T that requires ℓ. This condition generalizes (6.2) by providing an integer parameter m (≥ 1) that determines the strictness of the condition. Informally, m denotes the maximum number of blocking zones that can be crossed (while scheduled) before a task is granted the requested lock. Increasing m has the effect of weakening the condition. Condition (6.2) corresponds to the strictest case in which all critical sections satisfy (6.3) for $m = 1$. Theorem 6.1 then reflects the limiting behavior as $m \to \infty$.

Theorem 6.2 *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$ that satisfies* (6.3) *for some $m \geq 1$, where ℓ is managed by the Skip Protocol and all blocking zones of ℓ satisfy* (6.1)*, phase $T^{[i]}$ is equivalent to*

$$\mathbf{C}\Big(\emptyset, T^{[i]}.e + Q^{m+1}(T, \ell) + m \cdot T^{[i]}.B\Big)$$

in a system that does not include supertasks.

Proof. Condition (6.3) implies that $Q^m(T, \ell) \leq m\left(Q - T^{[i]}.B\right)$ holds. Informally, this condition states that the amount of processing time required by the competitors of $T^{[i]}$ across m slots cannot consume all of the processor time available outside of T's blocking zones in those slots. Hence, T is guaranteed to receive the lock within the m^{th} slot in which it is

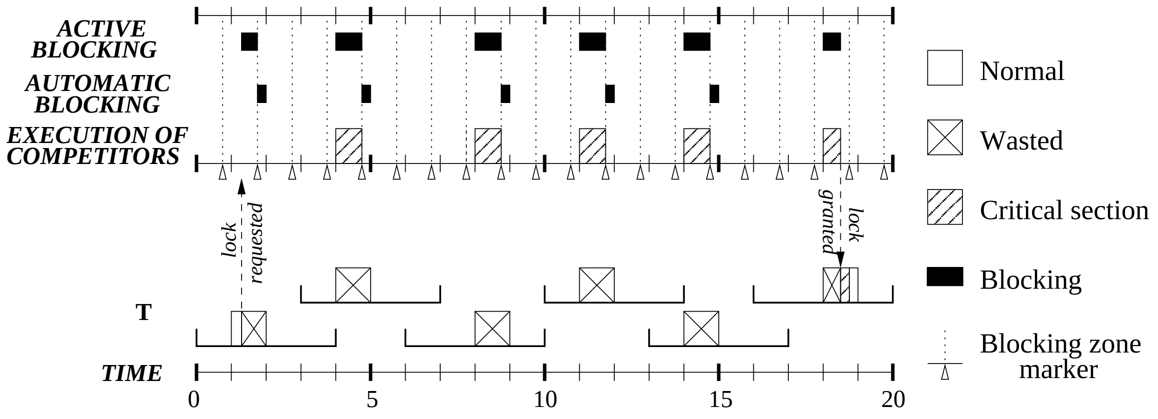


Figure 6.4: Worst-case blocking under the SP for a lock-requesting phase $T^{[i]}$ when $Q^5(T,\ell) = 4.25Q$ and $T^{[i]}.B = 0.1Q$. Since $\frac{1}{5}Q^5(T,\ell) = 0.85Q$, (6.3) holds for $m = 5$. The resulting worst-case scenario is depicted. The number of blocking zones crossed (while scheduled) is 5 (*i.e.*, m), while active blocking occurs within 6 (*i.e.*, $m+1$) quanta.

scheduled after issuing the lock request, if not sooner. This scenario is depicted in Figure 6.4. Including the quantum in which T issues the lock request, T is actively blocked across at most $m+1$ quanta. Hence, the total duration of active blocking cannot exceed $Q^{m+1}(T,\ell)$ time units. In addition, T may be automatically blocked in each of these quanta except for the last one, in which the lock is finally granted. Therefore, the request spans at most m blocking zones. Hence, the duration of phase $T^{[i]}$ is at most the sum of the durations of **(i)** $T^{[i]}$'s critical section, **(ii)** $Q^{m+1}(T,\ell)$ units of active blocking, and **(iii)** $m \cdot T^{[i]}.B$ units of automatic blocking. $\square$

6.4.5 Skip Protocol: Sample Implementation

To characterize the relative implementation overhead of the zone-based protocols, we now present a pseudocode implementation of the SP. To simplify the implementation, thereby exposing the underlying structure more clearly, we make the following (unrealistic) assumptions:[12]

- All processors share a common global clock.
- Tick interrupts occur with perfect timing (*i.e.*, no interrupt delay).
- All procedures execute instantaneously (*i.e.*, require zero time).

Despite the last assumption, the presented procedures *do* explicitly show where synchronization is needed in order to facilitate the removal of these assumptions.

The implementation is shown in Figure 6.5. We explain this listing in detail below.

Data structures. At each slot boundary, the kernel is assumed to store the time of the next slot boundary in *slot_end*. The **lock**, **request**, **processor**, and **task** records store information maintained by the kernel for each lock, pending lock request, processor, and task, respectively. (The code listing only shows fields used by the protocol; other fields will likely be needed in practice to manage other aspects of the system, such as I/O services.) We discuss the fields within each record below.

A **lock** record consists of at least the following four fields:

owner: identifies the task that currently holds the lock.

pending: count of unserviced requests for the lock.

processed: flag denoting whether a lock-granting decision has been applied yet.

next: current value of service counter, which enforces FIFO ordering of requests.

A **request** record consists of at least the following four fields:

lock: identifies which lock is currently requested or *nil* if no request is pending.

[12]As long as bounds on the clock error and interrupt delay on each processor can be determined, the presented algorithm can be easily adapted to real systems by simply modifying the implementation of `IsSafe`.

```
typedef lock:
    record
        owner: pointer to task initially nil;
        pending: 0...N initially 0;
        processed: boolean initially false;
        next: integer initially 0

private var
    T, U, V: pointer to task; L: pointer to lock;
    e: real; C: pointer to processor

procedure RequestLock(T, L, e)
1:  $T \rightarrow req.B := e$;
2:  $T \rightarrow req.lock := L$;
3:  AllowScheduling(false);
4:  LockKernel();
5:  if $L \rightarrow owner = nil \land$ IsSafe($T$) then
6:      $L \rightarrow owner := T$
    else
7:      if $L \rightarrow pending = 0$ then
8:          $L \rightarrow next := 0$ fi;
9:      $T \rightarrow req.tag := L \rightarrow next$;
10:     $L \rightarrow next := L \rightarrow next + 1$;
11:     $L \rightarrow pending := L \rightarrow pending + 1$;
12:     $T \rightarrow assigned \rightarrow csreq := T$;
13:     $T \rightarrow req.grant := false$ fi;
14: UnlockKernel();
15: AllowScheduling(true);
16: while $T \rightarrow req.grant = false$ do od

procedure ReleaseLock(T, L)
17: $T \rightarrow req.B := 0$;
18: $T \rightarrow req.lock := nil$;
19: LockKernel();
20: $U := nil$;
21: for $i := 1$ to $M$ do
22:     $V :=$ GetCPU($i$)$\rightarrow csreq$;
23:     if $V \neq nil \land$ IsSafe($V$)
            $\land V \rightarrow req.lock = L$ then
24:         if $U = nil$
                $\lor U \rightarrow req.tag > V \rightarrow req.tag$ then
25:             $U := V$ fi fi od;
26: $L \rightarrow owner := U$;
27: if $U \neq nil$ then
28:     $L \rightarrow pending := L \rightarrow pending - 1$;
29:     $U \rightarrow assigned \rightarrow csreq := nil$ fi;
30: UnlockKernel();
31: if $U \neq nil$ then
32:     $U \rightarrow req.grant := true$ fi

typedef request:
    record
        lock: pointer to lock initially nil;
        B: real initially 0;
        tag: integer initially 0;
        grant: boolean initially true

typedef task:
    record
        assigned: pointer to processor;
        req: request

typedef processor:
    record
        csreq: pointer to task initially nil;
        scheduled: pointer to task

shared var
    slot_end: integer

procedure IsSafe(T) returns boolean
33: return Time() $+ T \rightarrow req.B \leq slot\_end$

procedure DeactivateRequests()
34: for $i := 1$ to $M$ do
35:     GetCPU($i$)$\rightarrow csreq := nil$ od

procedure ActivateRequests()
36: for $i := 1$ to $M$ do
37:     $C :=$ GetCPU($i$);
38:     $T := C \rightarrow scheduled$;
39:     if $T \neq nil \land T \rightarrow req.lock \neq nil$
            $\land T \rightarrow req.grant = false$ then
40:         $C \rightarrow csreq := T$;
41:         $L := T \rightarrow req.lock$;
42:         $L \rightarrow processed := false$;
43:         if $L \rightarrow owner = nil \lor$
                $L \rightarrow owner \rightarrow req.tag > T \rightarrow req.tag$
                    then
44:             $L \rightarrow owner := T$ fi fi od;
45: for $i := 1$ to $M$ do
46:     $T :=$ GetCPU($i$)$\rightarrow csreq$;
47:     if $T \neq nil$ then
48:         $L := T \rightarrow req.lock$;
49:         if $L \rightarrow processed = false$ then
50:             $U := L \rightarrow owner$;
51:             $U \rightarrow req.grant := true$;
52:             $U \rightarrow assigned \rightarrow csreq := nil$;
53:             $L \rightarrow pending := L \rightarrow pending - 1$;
54:             $L \rightarrow processed := true$ fi fi od
```

Figure 6.5: Sample implementation of the Skip Protocol.

B: duration of the request's blocking zone.

tag: service-counter value assigned to the request.

grant: flag denoting whether the request has been granted the lock.

A **task** record consists of at least the following two fields:

req: the task's pending lock request, or *nil* if no request is pending.

assigned: the processor assigned to the task, or *nil* if the task is not executing.

A **processor** record consists of at least the following two fields:

csreq: the request issued by this processor's task, or *nil* if no request exists.

scheduled: the task assigned to this processor, or *nil* if the processor is idle.

The *assigned* fields of **task** records and the *scheduled* fields of **processor** records are assumed to be updated by the scheduler as scheduling decisions are made.

Detailed description. Code is not provided for the following procedures: `LockKernel`, `UnlockKernel`, `AllowScheduling`, `Time`, and `GetCPU`. A call to `LockKernel` obtains the special lock that guards the kernel data structures, while a call to `UnlockKernel` releases this lock. `AllowScheduling`(**boolean**) controls whether slot interrupts are delayed on the processor from which it is invoked, *i.e.*, invoking `AllowScheduling`(*false*) causes slot interrupts to be postponed until `AllowScheduling`(*true*) is called. (Recall that we discussed such interrupt postponement previously in Chapter 3.) A call to `Time` returns the current value of the shared global clock. Finally, `GetCPU`(i) returns a pointer to the processor record of the processor with index i, where $0 \leq i < M$.

`IsSafe`(T) determines whether task T's pending request is safe. (This procedure should not be invoked when no request is pending.) Due to the assumptions stated earlier, Figure 6.5

gives only a trivial implementation that compares the time remaining in the current slot to the blocking-zone length (line 33). Bounded clock error and interrupt delays can be accounted for by biasing either the comparison in line 33 or the B-values associated with the requests.

`RequestLock`(T, L, e) is invoked by task T to acquire lock L for at most e time units. In lines 1–2, the request parameters are stored in T's request record. (The assignment of B in line 1 is too simplistic for real systems since it does not account for practical issues, such as event jitter. We intentionally ignore such issues here since our goal is only to gain insight into the protocol's runtime overhead.) Following this, slot interrupts are postponed (line 3) and the kernel lock is obtained (line 4) to ensure that lines 5–13 execute atomically. If the lock is available and the request is safe (line 5), T is immediately granted the lock (line 6). Otherwise T takes a number from the service counter (lines 7–13) and then spins until granted the lock (line 16). Before taking its service number, T first checks whether any numbers are currently in use (line 7). If not, the counter is reset to 0 (line 8) to prevent rollover. The request is then assigned its service number (line 9) and both the service counter's value (line 10) and count of pending requests (line 11) are updated. The request is then stored in the local processor's record (line 12) and *grant* is cleared (line 13). Following this, the kernel lock is released (line 14) and slot interrupts are re-enabled (line 15). T then spins in line 16 until granted the lock. If the branch at line 5 is taken to line 6, then no spinning occurs at line 16. This is due to the fact that *grant* = *true* always holds when the procedure is invoked. Hence, it must also hold at line 16 if its value is not changed during the procedure. Notice that the call to `RequestLock` is a *synchronous* call, *i.e.*, it does not return until the lock is granted.

`ReleaseLock`(T, L) is invoked by task T to release lock L at the end of T's critical section. By design, this invocation is guaranteed to complete by the next slot interrupt. Hence, calls to `AllowScheduling` are unnecessary. Lines 17–18 clear the fields that record T's request information. The kernel lock is then obtained in line 19 to ensure the atomicity of lines 20–29. In lines 20–25, U is set to the request that should receive the lock next, or *nil* if no pending safe requests exist. The new owner of L is then recorded in line 27. If the lock was granted, then the count of pending requests (line 28) and processor's request marker (line 29) are updated. After releasing the kernel lock (line 30), the waiting task that was granted L, if any, is released from its spin (lines 31–32).

`DeactivateRequests` is automatically invoked by the kernel at slot boundaries, prior to making scheduling decisions for the new slot. This procedure simply consists of a loop that clears the request marker of each processor (lines 34-35).

`ActivateRequests` is automatically invoked by the kernel at slot boundaries after scheduling decisions have been made for the upcoming slot. This routine restores the request state of delayed requests and selects which of these requests are initially granted. In lines 37–38, the records for the processor and task scheduled on that processor are retrieved. If a scheduled task has a pending request (line 39), then the processor's marker is set (line 40), and the ownership of the lock is tentatively determined in lines 41–44. (Recall that all requests are safe at the start of a slot.) Once all scheduled tasks have been considered in the loop at line 36, these tentative ownership decisions are committed in the loop at line 45. Since M is expected to be smaller that $|\Gamma|$ in practice, the algorithm checks to see which locks are requested by scheduled tasks instead of checking each lock. Specifically, the loop checks the processor markers for pending requests (lines 46–48) rather than checking the *owner* field of every lock. However, since two scheduled tasks may request the same lock, care must be taken to ensure that each lock is processed only once. The *processed* flags are used for this purpose (lines 49 and 54). Committing an ownership decision consists of releasing the owner from its spin (line 51), clearing the request marker of the associated processor (line 52), and decrementing the count of pending requests for the lock (line 53).

Remarks. All of these routines, except `RequestLock` and `ReleaseLock` have an $O(M)$ time complexity. Since `RequestLock` and `ReleaseLock` depend on the kernel-locking procedures and `AllowScheduling`, their time complexities cannot be determined based on the provided details. As demonstrated by `ActivateRequests`, use of the SP does require a significant increase in the per-slot overhead in order to retain requests across slot boundaries. Indeed, the actions of the SP closely resemble those of a scheduler performing context switches.

6.4.6 Rollback Protocol: Description and Analysis

Under the Rollback Protocol (RP),[13] delayed requests are discarded (failed) at slot boundaries and must be re-issued by the requesting tasks when they resume execution. Hence, FIFO prioritization only applies to requests issued within the same slot. The RP sacrifices the guarantee that a request will be blocked by at most one request of each competing task in order to avoid the additional overhead of maintaining requests across multiple slots. A side effect is that preventing starvation (*i.e.*, guaranteeing (R1)) becomes more difficult. To guarantee starvation avoidance, we restrict consideration of the RP to cases in which (R3) holds. (Notice that (R3) mirrors the assumption made in Chapter 5 for lock-free operations.)

The next theorem characterizes the worst-case duration of a locking phase under the RP.

Theorem 6.3 *Given a phase* $T^{[i]} \in \mathcal{C}(\ell)$*, where* ℓ *satisfies* (6.2)*,* ℓ *is managed by the Rollback*

[13]The term *rollback* refers to the fact that a requesting task may need to re-issue a discarded request. This should not be confused with the "undo" processes commonly found in transaction-based systems.

Protocol, and all blocking zones of ℓ satisfy (6.1)*, phase $T^{[i]}$ is equivalent to*

$$\mathbf{C}\Big(\emptyset, T^{[i]}.e + 2 \cdot Q^1(T,\ell) + T^{[i]}.B\Big)$$

in a system that does not include supertasks.

Proof. The case considered here is almost identical to the $m = 1$ case of Theorem 6.2. However, since a requesting task T can have its request rejected due to crossing a slot boundary, its request can potentially be blocked by the same group of competitors in each slot. From (R3) and this observation, it follows that T's request can be delayed due to active (respectively, automatic) blocking for no more than $2 \cdot Q^1(T,\ell)$ (respectively, $T^{[i]}.B$) time units. □

6.4.7 Rollback Protocol: Sample Implementation

The sample implementation of the RP is shown in Figure 6.6. This implementation is based on the same assumptions and mostly the same data structures as the SP implementation. We now briefly describe the few exceptions. First, the maximum value of *pending* is now M. Since requests are no longer carried across slot boundaries at most M tasks (*i.e.*, the tasks that are scheduled) can have pending requests at any given time. (Under the SP, the value was bounded by the task count.) Second, each **request** record now has a *released* flag. In the SP implementation, the *grant* flags serve as both the triggers that release requesting tasks from their spins and as flags that denote when the lock has been granted. Only a single variable was needed for this purpose due to the fact that tasks were released from there spins only upon receiving the requested lock. Under the RP, a task may be released from its spin without receiving the lock, which implies that two flags are now needed. The *released* flag

```
typedef lock:
    record
        $owner$: pointer to task initially $nil$;
        $pending$: $0...M$ initially $0$;
        $next$: integer initially $0$

typedef request:
    record
        $lock$: pointer to lock initially $nil$;
        $B$: real initially $0$;
        $tag$: integer initially $0$;
        $released$: boolean initially $true$;
        $grant$: boolean initially $true$

procedure RequestLock($T$, $L$, $e$)
1:  $T \rightarrow req.B := e$;
2:  $T \rightarrow req.lock := L$;
    do
3:      AllowScheduling($false$);
4:      LockKernel();
5:      if $L \rightarrow owner = nil \wedge$ IsSafe($T$) then
6:          $T \rightarrow req.grant := true$;
7:          $L \rightarrow owner := T$
        else
8:          if $L \rightarrow pending = 0$ then
9:              $L \rightarrow next := 0$ fi;
10:         $T \rightarrow req.tag := L \rightarrow next$;
11:         $L \rightarrow next := L \rightarrow next + 1$;
12:         $L \rightarrow pending := L \rightarrow pending + 1$;
13:         $T \rightarrow assigned \rightarrow csreq := T$;
14:         $T \rightarrow req.released := false$;
15:         $T \rightarrow req.grant := false$ fi;
16:     UnlockKernel();
17:     AllowScheduling($true$);
18:     while $T \rightarrow req.released = false$ do od
19: while $T \rightarrow req.grant = false$

procedure ReleaseLock($T$, $L$)
20: $T \rightarrow req.B := 0$;
21: $T \rightarrow req.lock := nil$;
22: LockKernel();
23: $U := nil$;
24: for $i := 1$ to $M$ do
25:     $V :=$ GetCPU($i$)$\rightarrow csreq$;
26:     if $V \neq nil \wedge$ IsSafe($V$)
            $\wedge\ V \rightarrow req.lock = L$ then
27:         if $U = nil \vee$
                $U \rightarrow req.tag > V \rightarrow req.tag$
                then
28:             $U := V$ fi fi od;
29: $L \rightarrow owner := U$;
30: if $U \neq nil$ then
31:     $L \rightarrow pending := L \rightarrow pending - 1$;
32:     $U \rightarrow assigned \rightarrow csreq := nil$;
33:     $U \rightarrow req.grant := true$ fi;
34: UnlockKernel();
35: if $U \neq nil$ then
36:     $U \rightarrow req.released := true$ fi

procedure DeactivateRequests()
37: for $i := 1$ to $M$ do
38:     $C :=$ GetCPU($i$);
39:     $T := C \rightarrow csreq$;
40:     $C \rightarrow csreq := nil$;
41:     if $T \neq nil$ then
42:         $T \rightarrow req.lock \rightarrow pending := 0$;
43:         $T \rightarrow released := true$ fi od
```

Figure 6.6: Sample implementation of the Rollback Protocol.

acts as the spin-release variable in this implementation.

Detailed Description. Since request contexts do not need to be saved and restored, this implementation has no `ActivateRequests` procedure. The implementations of `IsSafe`, `Time`, `AllowScheduling`, `LockKernel`, `UnlockKernel`, and `GetCPU` are common to both the SP and

RP implementations, as are the purpose, timing, and assumptions made of the remaining three procedures. We describe each of these three procedures below.

As before, an invocation of `RequestLock` begins by storing the request parameters in T's record (lines 1–2). Following this, the retry loop for the request is entered. The request is made as before in lines 3–18 (corresponding to lines 3–16 in Figure 6.5), with a few minor exceptions. Since $grant = true$ may not hold at line 3, the flag's value is now explicitly set to *true* if T immediately takes ownership of L (in the branch spanning lines 6-7). Similarly, two flags (*grant* and *released*) are now initialized (lines 14–15) when setting up the request, instead of one. Finally, a task that is released from its spin may have been released due to a failure. Hence, following its release from line 18, T checks the value of *grant* (line 19) to determine whether the lock was granted. If not, the request is retried by executing lines 3–18 again. Notice that the call to `RequestLock` is still synchronous.

`ReleaseLock` is virtually identical to that used by the SP. Indeed, the only difference is the need to set the *grant* flag (line 33) so that, when released by line 36, the lock-holding task is able to determine whether it has been granted the lock.

`DeactivateRequests` has two purposes under the RP. First, as in the SP, the processor request markers are cleared (line 40). Second, if a request is found to be pending on a processor at the slot boundary (lines 39 and 41–43), then it must be failed. This is accomplished by resetting the count of pending requests for the requested lock to zero (line 42) and releasing the requesting task from its spin (line 43) so that a retry will occur when the task resumes. Since $pending = 0$ for locks for which no request is pending, it follows from line 42 that $pending = 0$ for all locks upon completion of this call. This implies both that the pending-request counter is correctly re-initialized at slot boundaries, and that the first request for each lock in a slot is guaranteed to reset the service counter to zero at lines 8–9.

Remarks. The time complexity of these routines match the corresponding routines in the SP implementation. However, `DeactivateRequests` is almost certain to introduce significantly less per-slot overhead than the combination of `ActivateRequests` and `DeactivateRequests` in the SP implementation. This decreased per-slot overhead somewhat offsets the extra locking overhead caused by using the RP instead of the SP. (Recall that the worst-case blocking scenario is usually worse under the RP than under the SP because requests are discarded at slot boundaries.) Hence, neither protocol provides an obvious advantage over the other. (Indeed, this is the reason why we present both protocols.)

6.4.8 The Impact of Supertasking

Both the SP and RP exploit characteristics inherent under quantum-based scheduling. Hence, this approach is compatible with quantum-based supertasking, but not with fully preemptive supertasking. The adaptation of Theorems 6.1, 6.2, and 6.3 for cases in which all competing tasks are members of quantum-based supertasks is straightforward: $Q^m(T, \ell)$ and $I(T, \ell)$ are replaced by $Q^m(\mathcal{S}, \ell)$ and $I(\mathcal{S}, \ell)$, respectively, where $T \in \mathcal{S}$.

6.5 Supporting Long Critical Sections

In this section, we present a simple server-based protocol to support critical sections of arbitrary length. The design of this protocol focuses on avoiding the problems associated with inheritance-based approaches. The resulting protocol is simple to use and analyze, but is expected to yield only mediocre performance in practice. The primary purpose of this protocol is to provide a baseline performance measurement that can be used to evaluate the performance of the SP, RP, and other protocols. The specific goals of the design are listed below.

1. Blocking durations should be independent of the weights of requesting tasks.
2. Requesting tasks should not be required to change weights.
3. Modifying task parameters should impact performance in a predictable way.
4. The protocol should not introduce substantial per-slot overhead.

6.5.1 Static-weight Server Protocol

When using lock servers, a server task V executes all critical sections guarded by ℓ in place of the requesting tasks. Such servers are actually quite common in practice. In addition to implementing critical section and kernel calls, such special processes are often used to implement basic communication services, such as remote-procedure calls (RPCs).

Issues. The primary issue when using server tasks is the scheduling of the server. Many approaches have been proposed to exploit the structure of the system in which the servers will execute. For instance, in fixed-priority systems, servers can be scheduled as normal tasks and inherit the priority of the requesting tasks while servicing requests. This policy, called *priority tracking*, addresses the potential for priority inversion caused by the server's processing of requests on behalf of low-priority tasks. For example, kernel threads in the LynxOS use this approach [70]. Similarly, in reservation-based systems (*e.g.*, see [15, 20]), the reservation[14] of the requesting task can be passed to the server. This ensures that the processor time consumed by the server is properly charged against the requesting tasks.

[14]A *reservation* is a form of rate-enforcement mechanism. Specifically, tasks are assigned processor time *budgets* that are consumed as the tasks execute. When the budgets are exhausted, the tasks become ineligible to execute. Additional rules define when and how budgets are replenished.

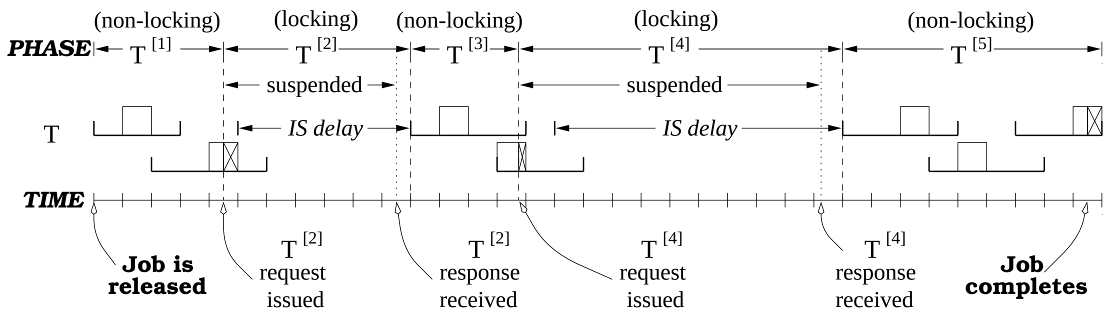


Figure 6.7: Illustration of the behavior of a five-phase task T with weight 7/19 when all locking phases use the SWSP. Phases $T^{[1]}$, $T^{[3]}$, and $T^{[5]}$ do not require locks, and hence are executed locally by T while phases $T^{[2]}$ and $T^{[4]}$ are executed remotely by lock servers. Phase transitions are shown across the top of the figure and time is shown across the bottom. Arrows show when each request begins and ends. Unshaded boxes show where T executes locally while boxes containing X's denote unutilized processor time.

To satisfy Restriction 1 (on page 271), we consider only the use of server tasks with statically defined weights. When using servers, the duration of a locking phase is determined by the worst-case responsiveness of the associated server task. This responsiveness, in turn, depends on the scheduling of the server and the amount of processor time required to complete the operations. Hence, the use of static server weights satisfies Restriction 1. We refer to the server protocol considered here as the Static-weight Server Protocol (SWSP).

Detailed description. Let $V(\ell)$ denote the server task that implements lock ℓ. Each server is assumed to maintain a FIFO-ordered request queue, $\mathsf{WAIT}\,(\ell)$. As long as $\mathsf{WAIT}\,(\ell)$ is non-empty, $V(\ell)$ releases subtasks as early as is permitted under the GIS task model. On the other hand, if $V(\ell)$ finds that $\mathsf{WAIT}\,(\ell)$ is empty, then it delays the release of its next subtask until $\mathsf{WAIT}\,(\ell)$ becomes non-empty again. Stated more precisely, we assume that the SWSP servers use an early-release-and-stall server policy (as described in Chapter 3). We further assume that a requesting task T issues its request via a synchronous call with behavior similar to an RPC, *i.e.*, T is suspended until the response is received.

Figure 6.7 illustrates the behavior of one job of a five-phase task T when all locking phases use the SWSP. As shown, server delays can be treated just as any other form of suspension with one exception: the local processing required to initiate the request and process the response must be factored into the execution-time estimates of the phases that precede and succeed, respectively, the locking phase. (When using the SP and the RP, this overhead can be factored into the locking phase's requirements.)

Analysis. Before continuing, we define additional shorthand expressions unique to the SWSP analysis. First, let $\delta\left(A, w\right)$ denote the shortest interval of time over which a task with weight w that does not experience intra-sporadic delays is guaranteed to receive at least A quanta, where A is a positive integer. By Lemma 3.3,

$$\delta\left(A, w\right) = \left\lceil \frac{A + \beta - 1}{w} \right\rceil + \epsilon.$$

Informally, $\delta\left(A, w\right)$ is one slot less than the worst-case span of any $A+1$ consecutive windows, as illustrated in Figure 6.8.

The following lemma and corollary equates an SWSP locking phase to a non-locking phase for the purpose of selecting a task weight and determining schedulability of the system. (Recall that $\mathbf{I}(\theta)$ denotes a suspension phase of maximum duration θ.)

Theorem 6.4 *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$, where ℓ is managed by the SWSP, phase $T^{[i]}$ is equivalent to*

$$\mathbf{I}\left(\delta\left(\left\lceil \frac{T^{[i]}.e + I(T, \ell)}{Q} \right\rceil, V(\ell).w\right)\right)$$

in a system that does not include supertasks.

Proof. Since $I(T, \ell)$ is a trivial upper bound on the time required by competing requests,

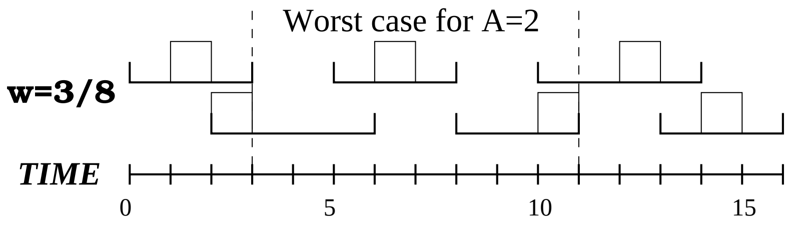


Figure 6.8: The marked interval shows the interval that defines $\delta(A, w)$ when $w = \frac{3}{8}$, $A = 2$, $\beta = 2$, and $\epsilon = 0$. In this case, $\delta(A, w) = \lceil (2 + 2 - 1)\frac{8}{3} \rceil + 0 = 8$. As shown, $\delta(A, w)$ is one slot shorter than the longest span of any $A + 1$ consecutive windows.

the server cannot consume more than $\left\lceil \frac{T^{[i]}.e + I(T,\ell)}{Q} \right\rceil$ quanta without completing T's critical section. Since the server remains active while T's request is pending, it follows that the server completes T's critical section after a delay of at most $\delta\left(\left\lceil \frac{T^{[i]}.e + I(T,\ell)}{Q} \right\rceil, V(\ell).w\right)$ slots. $\Box$

Assigning server weights. We now describe a simple algorithm for assigning weights to servers. Our algorithm focuses on dividing a fixed fraction of the processor bandwidth reserved for servers, denoted ∂, among those servers in proportion to their requirements. Let

$$\mathcal{U}(T, \ell) = \frac{\sum_{T^{[i]} \in \mathcal{C}(\ell)} T^{[i]}.e}{T.p}.$$

Informally, $\mathcal{U}(T, \ell)$ is T's utilization of lock ℓ. Also, let

$$\mathcal{U}(\ell) = \sum_{T \in \tau} \mathcal{U}(T, \ell),$$

which is the total utilization of lock ℓ by all tasks in τ. (We also refer to this quantity as the *lock utilization* of ℓ.) ∂ can be proportionally distributed among the servers by using lock utilizations like relative weights,[15] as suggested by the formula below.

$$V(\ell).w = \frac{\mathcal{U}(\ell)}{\sum\limits_{\ell' \in \Gamma} \mathcal{U}(\ell')} \cdot \partial$$

Impact of supertasking. The SWSP is compatible with quantum-based supertasks. However, because critical sections execute remotely, the use of supertasks does not impact the worst-case blocking overhead. (We assume that lock servers are not assigned to supertasks.)

6.6 Experimental Results

In this section, we present the results of four experimental studies that cover the range of locking alternatives discussed in this chapter. As in previous chapters, these studies focus on measuring the performance of the different alternatives when applied to randomly generated task sets taken from a structured sample space. In particular, we restrict attention to task sets in which each task requires at least one lock during its job and each job has between two and twenty-four phases. Critical sections occur in consecutive phases with probability $\frac{1}{4}$. In addition, each study imposes an upper limit on the duration of each critical section, which is denoted C. Specific details are provided for each study below.

Supertask assignment. These studies consider the use of quantum-based supertasking with the SP and the RP. Tasks are assigned to supertasks using a modified version of the heuristic presented in Section 5.4. That heuristic selects lock-free objects in non-increasing order by weighted contention. For each object ℓ, tasks that access ℓ are then assigned to supertasks in non-increasing order by $A(T, \ell)$. When using locks, $\mathcal{U}(\ell)$ replaces the weighted

[15]In the presented formula, we ignore the unit upper limit on weights. An algorithm for dividing bandwidth while respecting such a restriction can be found in [17].

contention as the criteria for selecting ℓ,[16] and $T.\hat{e}(\ell)$ replaces $A(T, \ell)$ as the criteria for selecting tasks. As in Chapter 5, we consider each of the FF and NF assignment rules.

6.6.1 Study 1: Performance under (R3)

In this section, we present the details and results of the first study. This study focused on measuring the relative performance of each applicable locking approach when (R3) holds.

Sample space. For this study, the sample space was defined as follows:

- M is in the range $2, \ldots, 16$;
- $|\tau|$ is in the range $5 \cdot \log_2 M, \ldots, 30 \cdot \log_2 M$;
- $\tau.u$ is in the range $0.2 \cdot M, \ldots, 0.8 \cdot M$;
- $|\Gamma|$ is in the range $\log_2 M, \ldots, 10 \cdot \log_2 M$;
- $T.p$ is in the range $50, \ldots, 2000$;
- C equals 0.025, *i.e.*, 2.5% of a quantum.

These ranges were chosen rather arbitrarily based on values that seemed reasonably likely to occur in practice. C was selected so that (R3) was likely to hold for the generated task sets.

Sampling. Samples were collected as follows:

- M was set to each value in the set $\{ 2^x \mid 1 \leq x \leq 4 \}$;
- $|\tau|$ was set to each value in the set $\{ x \cdot \log_2 M \mid 5 \leq x \leq 30 \}$;

[16] Though both the lock utilization and weighted contention are reasonable predictors of contention, these quantities are not analogous. Lock utilization reflects the amount of processor time consumed by the execution of critical sections in the limit. Weighted contention is the frequency of access to an object in the limit multiplied by the object's retry cost.

- $\tau.u$ was set to each value in the set $\{\ 0.025x \cdot M \mid 8 \leq x \leq 32\ \}$;
- $|\Gamma|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 1 \leq x \leq 10\ \}$.

For each combination of the above parameter assignments, ten valid task sets were generated and evaluated. In this context, a task set must satisfy several constraints to be considered valid. First, under the zone-based protocols, all locks must satisfy (R3) and the task set must be schedulable on M processors. (Since the analysis presented earlier for the SP and the RP is based on M, we must verify that the task set is actually schedulable on M processors for the analysis to be correct.) Second, the task set must be feasible under the SWSP when using an unlimited number of processors. Feasibility can be checked by computing task weights for the special case in which each lock server is assigned a unit weight. Such a case is guaranteed to produce the lowest possible task weights under the SWSP due to the fact that each server exhibits the best possible responsiveness. Hence, if a task weight exceeds unity under these conditions, then the task set is not schedulable under the SWSP, regardless of the processor count. Recall that sampling validity is *not* related directly to feasibility or schedulability. It is only a measure of the fraction of generated samples that could be quantitatively compared. Since the analysis of the SWSP is independent of M, it is not necessary to discard samples that are not schedulable on M processors under the SWSP.

Measurement. The following measurements were taken during the experimental runs.

Ideal: the sum of the ideal weights[17] of all tasks (a baseline measurement); this measurement reflects mapping overhead alone.

SWSP: the total weight of all tasks when all locks use the SWSP and supertasks are not

[17] A task's ideal weight is computed by treating critical sections as if they require no shared resources.

used; this measurement reflects mapping and locking overhead.

SP: the total weight of all tasks when all locks use the SP and supertasks are not used; this measurement reflects mapping and locking overhead.

SP/Ideal QB (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic when using the SP for all locks (a baseline measurement); this measurement reflects mapping and locking overhead.

SP/Ideal QB (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic when using the SP for all locks (a baseline measurement); this measurement reflects mapping and locking overhead.

SP/QB-EPDF (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic under the QB-EPDF scenario when using the SP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

SP/QB-EPDF (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic under the QB-EPDF scenario when using the SP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

SP/QB-EDF (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic under the QB-EDF scenario when using the SP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

SP/QB-EDF (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic under the QB-EDF scenario when using the SP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

RP: the total weight of all tasks when all locks use the RP and supertasks are not used; this measurement reflects mapping and locking overhead.

RP/Ideal QB (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic when using the RP for all locks (a baseline measurement); this measurement reflects mapping and locking overhead.

RP/Ideal QB (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic when using the RP for all locks (a baseline measurement); this measurement reflects mapping and locking overhead.

RP/QB-EPDF (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic under the QB-EPDF scenario when using the RP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

RP/QB-EPDF (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic under the QB-EPDF scenario when using the RP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

RP/QB-EDF (FF): the sum of the ideal weights of all supertasks created by the FF form of the assignment heuristic under the QB-EDF scenario when using the RP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

RP/QB-EDF (NF): the sum of the ideal weights of all supertasks created by the NF form of the assignment heuristic under the QB-EDF scenario when using the RP for all locks; this measurement reflects mapping, reweighting, and locking overhead.

To ensure termination when reweighting, $\mathcal{S}.w_\phi + 10^{-4}$ was passed as $w_{\min}$ when invoking

`Reweight` (see Section 4.5). When discussing results, we ignore trends in the reweighting overhead; a discussion of these trends can be found in Chapter 4.

Overall performance. The tables shown in Figures 6.9–6.12 summarize the overall results of this study. Separate tables are given for each value of M considered in the study. These tables report the mean inflation experienced under each approach along with related statistics using the same format as the tables presented in previous chapters. Since all halflengths are negligible relative to the associated means, we ignore them in the discussion that follows. In addition, the standard deviation measurements show no trends other than those discussed in earlier chapters. Hence, we focus primarily on the Mean and Best columns of the tables in the discussion that follows. Notice that the inflation under the SWSP often exceeds M. Recall that task sets are not required to be schedulable on M processors under the SWSP since this is not necessary to enable a quantitative comparison. (The criteria for sampling validity was discussed in detail earlier on page 277.) The SWSP analysis effectively calculates the minimum number of processors needed by the task set (which may exceed M).

Consider the $M = 2$ results, which are shown in Figure 6.9. As expected, the zone-based approach performs much better than the server approach. Indeed, the mean inflation produced by the SWSP is two orders of magnitude greater than all other means. In this case, the average performance of the SP and the RP are effectively the same, which suggests that the RP is likely preferable since it introduces less per-slot overhead.

The supertasking trends closely resemble those observed earlier in Chapter 5. Specifically, the benefit of using supertasks is limited for low processor counts. As a result, the reweighting overhead is prohibitively large under QB-EPDF. Even under QB-EDF, the overhead was observed to be unacceptably high for a significant fraction (10.582%) of the tested task sets.

$M = 2$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.017054	0.010149	0.00010254	*N/A*
SWSP	3.400218	2.275139	0.02298694	0.000%
SP	0.031534	0.018193	0.00018382	10.582%
SP/Ideal QB (FF)	0.023252	0.014283	0.00014431	*N/A*
SP/Ideal QB (NF)	0.023492	0.014422	0.00014572	*N/A*
SP/QB-EPDF (FF)	0.052283	0.060166	0.00060789	*N/A*
SP/QB-EPDF (NF)	0.047581	0.055227	0.00055799	*N/A*
SP/QB-EDF (FF)	0.023562	0.014248	0.00014395	8.672%
SP/QB-EDF (NF)	0.023755	0.014422	0.00014572	2.437%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	78.309%
RP	0.031838	0.018221	0.00018410	*N/A*
RP/Ideal QB (FF)	0.023842	0.014629	0.00014780	*N/A*
RP/Ideal QB (NF)	0.023987	0.014731	0.00014883	*N/A*
RP/QB-EPDF (FF)	0.052756	0.060324	0.00060949	*N/A*
RP/QB-EPDF (NF)	0.047996	0.055362	0.00055936	*N/A*
RP/QB-EDF (FF)	0.024151	0.014629	0.00014780	*N/A*
RP/QB-EDF (NF)	0.024250	0.014697	0.00014849	*N/A*

Figure 6.9: Summary of the $M = 2$ results of the first locking experiment in which all locks respect (R3).

Finally, neither the FF nor NF assignment rules always produces the best results, though the FF rule produced less inflation in the majority of the tested sets.

The remaining tables in Figures 6.10–6.12 show the same basic trends. A notable exception is that the locking overhead progressively increases as M increases. As a result, the benefit of using supertasks increases with M. Despite this, QB-EPDF supertasking continues to provide only negligible benefit, if any. (As M increases, more supertasks are typically needed, and hence reweighting overhead increases also.) QB-EDF supertasking, on the other hand, consistently reduces inflation throughout. Finally, notice that no tested sets favored the use of the SWSP instead of the SP. (See Appendix B for a more detailed summary of which approach was favored by the tested samples.)

In the remainder of this section, we present graphs to show how inflation scales with

$M = 4$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.033922	0.017635	0.00017818	*N/A*
SWSP	7.357997	4.515327	0.04562074	0.000%
SP	0.101851	0.056321	0.00056904	0.971%
SP/Ideal QB (FF)	0.062335	0.036892	0.00037274	*N/A*
SP/Ideal QB (NF)	0.063476	0.037577	0.00037966	*N/A*
SP/QB-EPDF (FF)	0.098554	0.079448	0.00080271	*N/A*
SP/QB-EPDF (NF)	0.095652	0.079568	0.00080391	*N/A*
SP/QB-EDF (FF)	0.062775	0.036878	0.00037260	35.465%
SP/QB-EDF (NF)	0.063865	0.037563	0.00037952	7.785%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	55.780%
RP	0.107083	0.056675	0.00057261	*N/A*
RP/Ideal QB (FF)	0.064867	0.038013	0.00038407	*N/A*
RP/Ideal QB (NF)	0.065953	0.038665	0.00039066	*N/A*
RP/QB-EPDF (FF)	0.100987	0.080075	0.00080904	*N/A*
RP/QB-EPDF (NF)	0.098063	0.080218	0.00081049	*N/A*
RP/QB-EDF (FF)	0.065307	0.038000	0.00038393	*N/A*
RP/QB-EDF (NF)	0.066342	0.038652	0.00039052	*N/A*

Figure 6.10: Summary of the $M = 4$ results of the first locking experiment in which all locks respect (R3).

the task set parameters. In all cases, the “Ideal QB (FF)” and “SP/QB-EDF (FF)” curves (respectively, the “Ideal QB (NF)” and “SP/QB-EDF (NF)” curves) were virtually co-linear. To simplify the graphs, the “Ideal QB (FF)” and “Ideal QB (NF)” curves have been omitted. In addition, all curves representing the use of the SP with supertasks are almost co-linear to the corresponding RP curve. For example, the “SP/QB-EPDF (NF)” and “RP/QB-EPDF (NF)” curves are co-linear or close to being co-linear in all graphs.

Impact of the task count. Since inflation under the SWSP is two orders of magnitude larger than that under the other approaches, each set of measurements is shown using two graphs. Figures 6.13, 6.14, 6.15, and 6.16 show how inflation varies with the task count on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP’s inflation. Figures 6.17, 6.18, 6.19, and 6.20 show inflation at the scale of the SP and the RP. Plots of

$M = 8$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.050873	0.024980	0.00025239	*N/A*
SWSP	11.543613	6.711607	0.06781092	0.000%
SP	0.234618	0.144966	0.00146466	1.589%
SP/Ideal QB (FF)	0.134715	0.083349	0.00084212	*N/A*
SP/Ideal QB (NF)	0.138098	0.085674	0.00086561	*N/A*
SP/QB-EPDF (FF)	0.210349	0.170830	0.00172599	*N/A*
SP/QB-EPDF (NF)	0.228375	0.226486	0.00228831	*N/A*
SP/QB-EDF (FF)	0.135430	0.083295	0.00084157	64.783%
SP/QB-EDF (NF)	0.138831	0.085604	0.00086490	11.715%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	21.912%
RP	0.271631	0.150256	0.00151812	*N/A*
RP/Ideal QB (FF)	0.144974	0.087898	0.00088808	*N/A*
RP/Ideal QB (NF)	0.148815	0.090366	0.00091301	*N/A*
RP/QB-EPDF (FF)	0.220535	0.172612	0.00174399	*N/A*
RP/QB-EPDF (NF)	0.239380	0.228189	0.00230551	*N/A*
RP/QB-EDF (FF)	0.145690	0.087852	0.00088762	*N/A*
RP/QB-EDF (NF)	0.149550	0.090311	0.00091246	*N/A*

Figure 6.11: Summary of the $M = 8$ results of the first locking experiment in which all locks respect (R3).

the sampling validity can be seen in Figures 6.21 and 6.22.

Consider the graphs of sampling validity first. For low processor counts, the sampling validity linearly decreases as the task count increases. According to Figures 6.17, 6.18, 6.19, and 6.20, the RP is the likely cause of this gradual decline (since it produces the highest inflation). For higher processor counts, a dip also occurs for low task counts. As the graphs of inflation show, the high reweighting overhead suffered by QB-EPDF is the apparent cause of this drop. Though Figure 6.19 does not appear to be impacted by the sampling validity, the "SP/QB-EPDF (NF)" and "RP/QB-EPDF (NF)" measurements shown in Figure 6.20 do appear to be lower than expected around task counts of 20. In such cases, we say that the measurements are *skewed* by the sampling validity. In this case, the measurements appearing

$M = 16$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.067556	0.032527	0.00032864	*N/A*
SWSP	16.151387	9.012471	0.09105778	0.000%
SP	0.418817	0.317556	0.00320844	5.663%
SP/Ideal QB (FF)	0.256111	0.174459	0.00176265	*N/A*
SP/Ideal QB (NF)	0.263267	0.181058	0.00182933	*N/A*
SP/QB-EPDF (FF)	0.509596	0.467962	0.00472806	*N/A*
SP/QB-EPDF (NF)	0.646442	0.727375	0.00734906	*N/A*
SP/QB-EDF (FF)	0.257800	0.173954	0.00175755	75.072%
SP/QB-EDF (NF)	0.265356	0.180178	0.00182043	15.072%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	4.192%
RP	0.570231	0.365531	0.00369315	*N/A*
RP/Ideal QB (FF)	0.293139	0.191669	0.00193653	*N/A*
RP/Ideal QB (NF)	0.303522	0.199722	0.00201790	*N/A*
RP/QB-EPDF (FF)	0.548215	0.472258	0.00477148	*N/A*
RP/QB-EPDF (NF)	0.689126	0.729944	0.00737501	*N/A*
RP/QB-EDF (FF)	0.294834	0.191196	0.00193176	*N/A*
RP/QB-EDF (NF)	0.305629	0.198887	0.00200946	*N/A*

Figure 6.12: Summary of the $M = 16$ results of the first locking experiment in which all locks respect (R3).

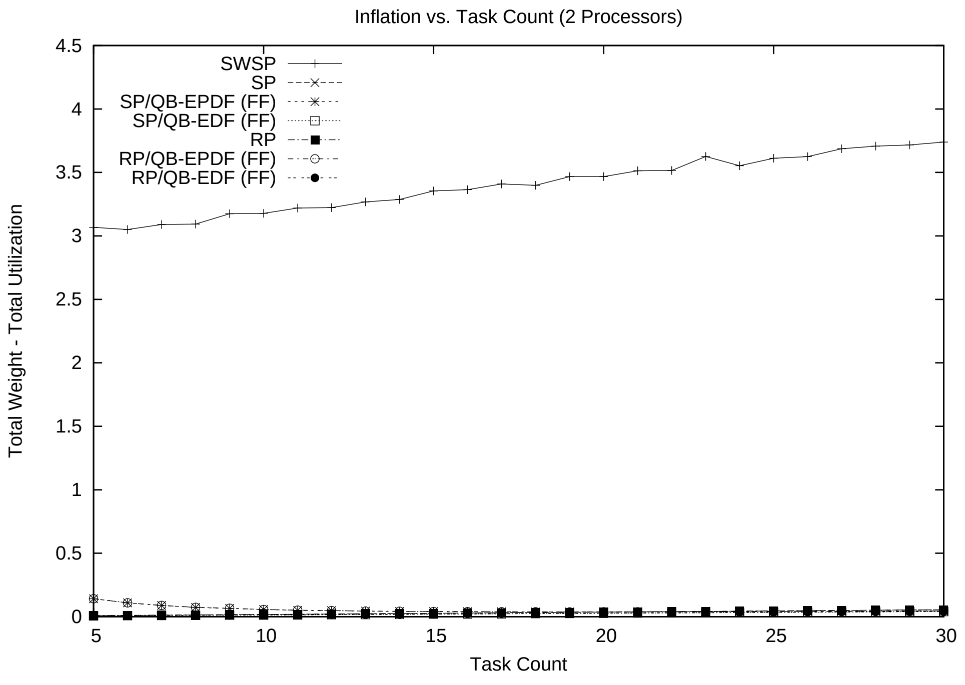
to be skewed toward zero, which means that they are lower than expected.[18]

Observation 6.1 *Locking overhead increases approximately linearly with the task count under all approaches.*

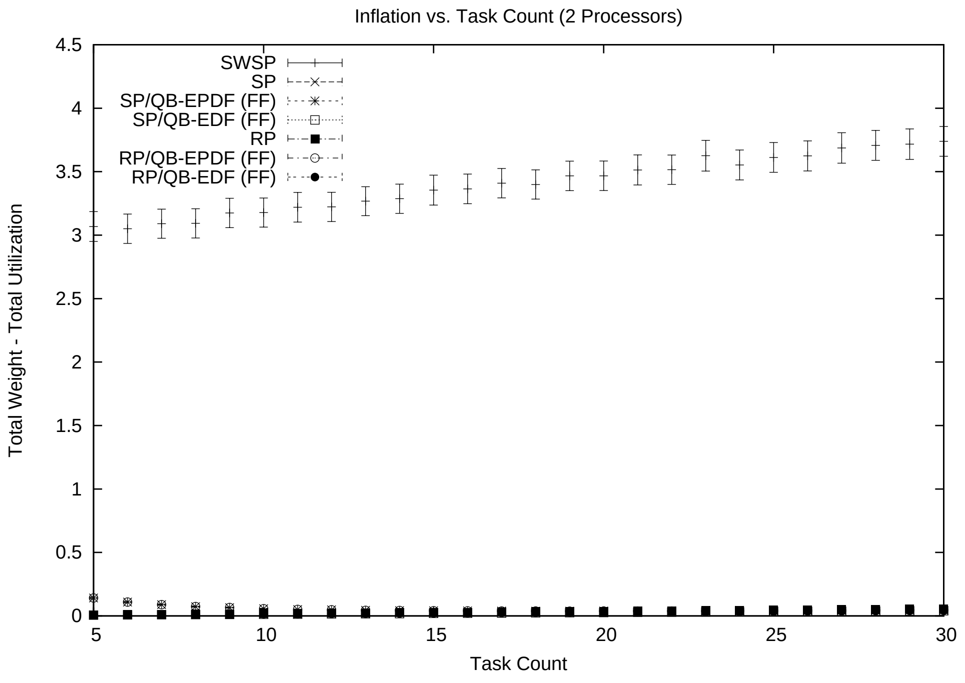
Explanation: Consider the performance of the SWSP in Figures 6.13–6.16 first. Under the SWSP, blocking estimates assume that one request of each competitor executes before the request in question. Increasing the task count increases the number of competitors. Hence, the observed behavior is not surprising.

Now consider the performance of the zone-based protocols in Figures 6.17–6.20. (The high inflation at low task counts under the QB-EPDF-based measurements are the result of

[18]It is impossible to prove that unexpected behavior is the result of sample filtering. We are careful to consider sampling validity as an explanation of unexpected behavior only when the observed behavior (*i.e.*, the shape of the curve) appears to be correlated with the sampling validity curve.

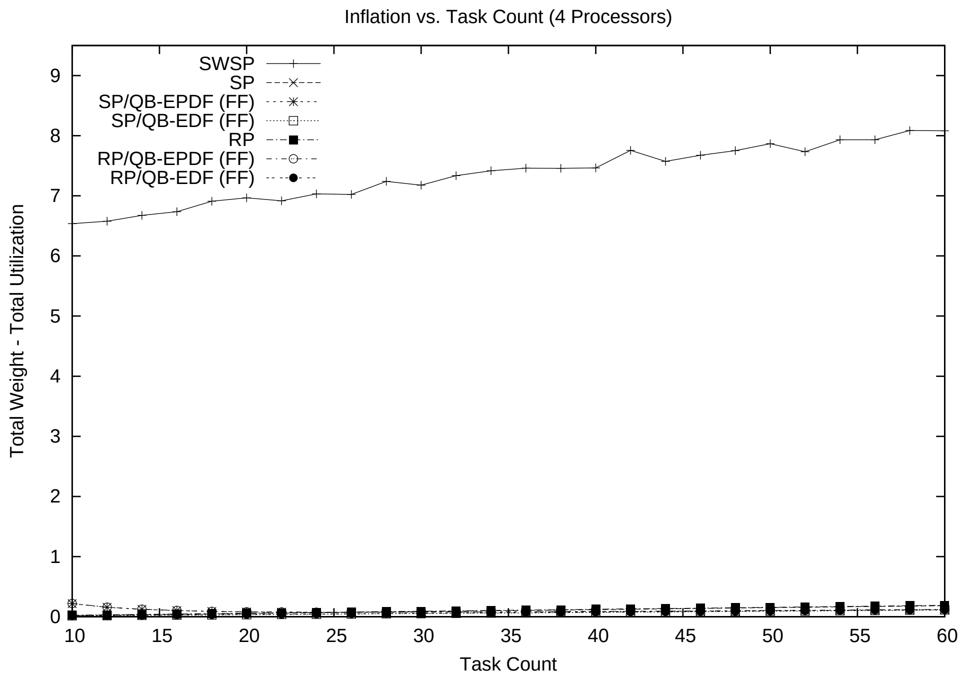


(a) Sample Means Only

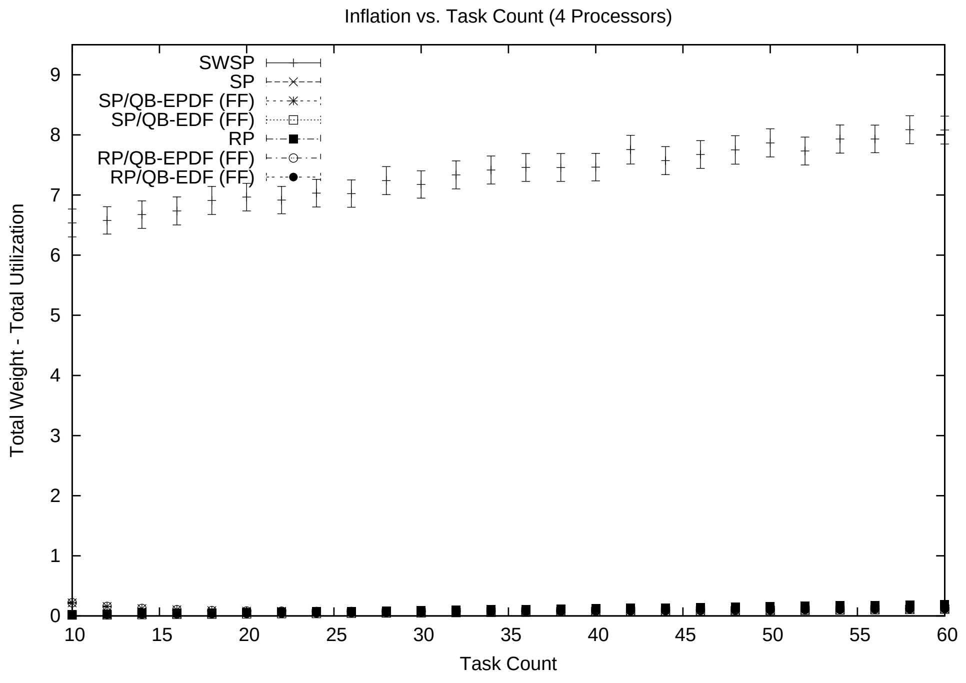


(b) 99% Confidence Interval

Figure 6.13: Plots show how inflation varies as the task count is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

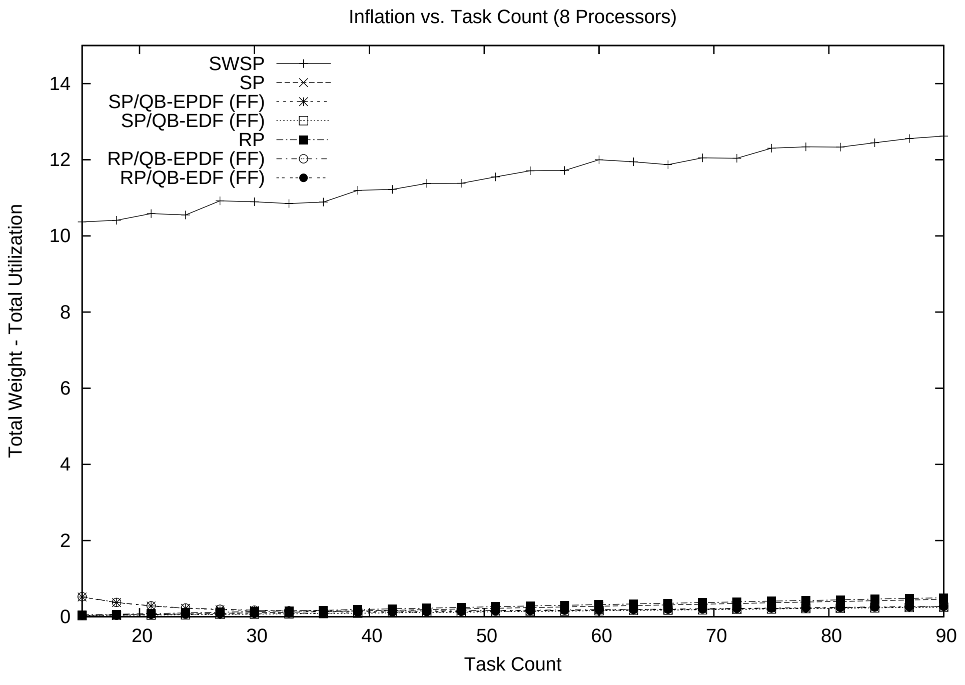


(a) Sample Means Only

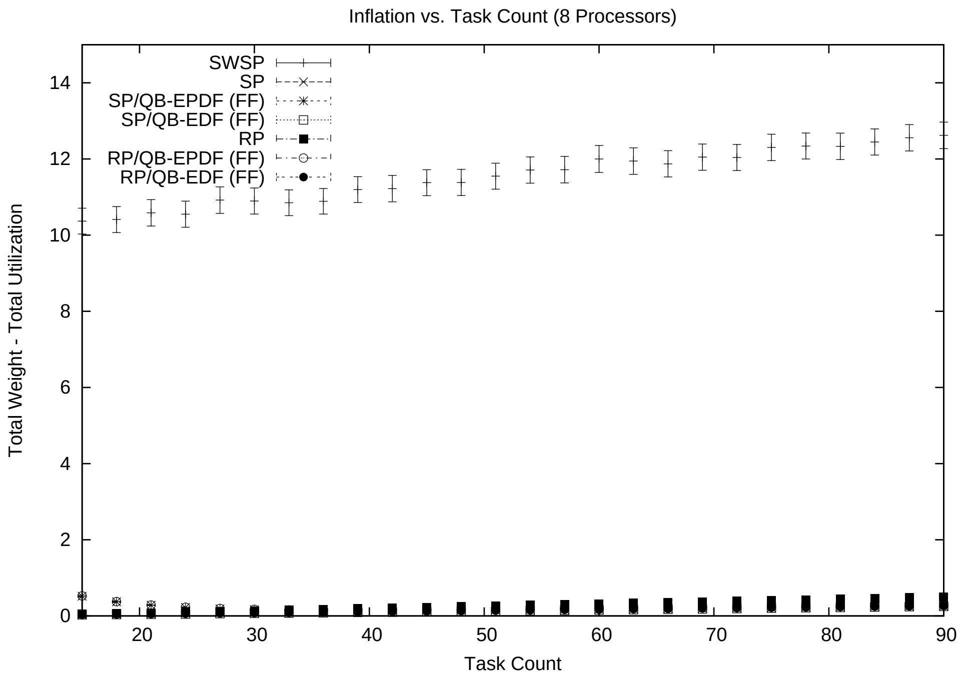


(b) 99% Confidence Interval

Figure 6.14: Plots show how inflation varies as the task count is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

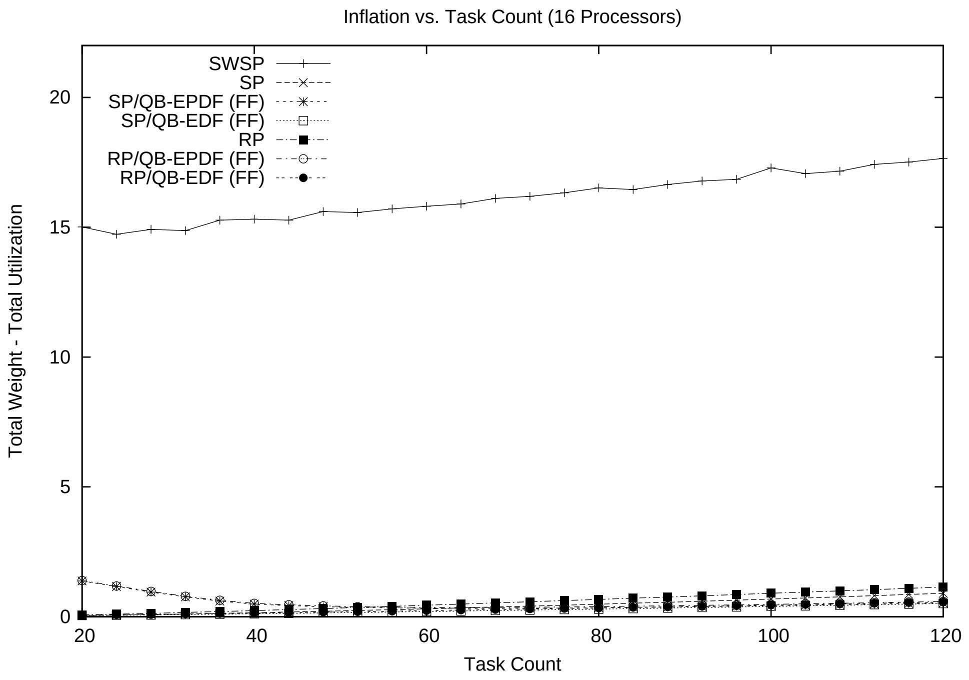


(a) Sample Means Only

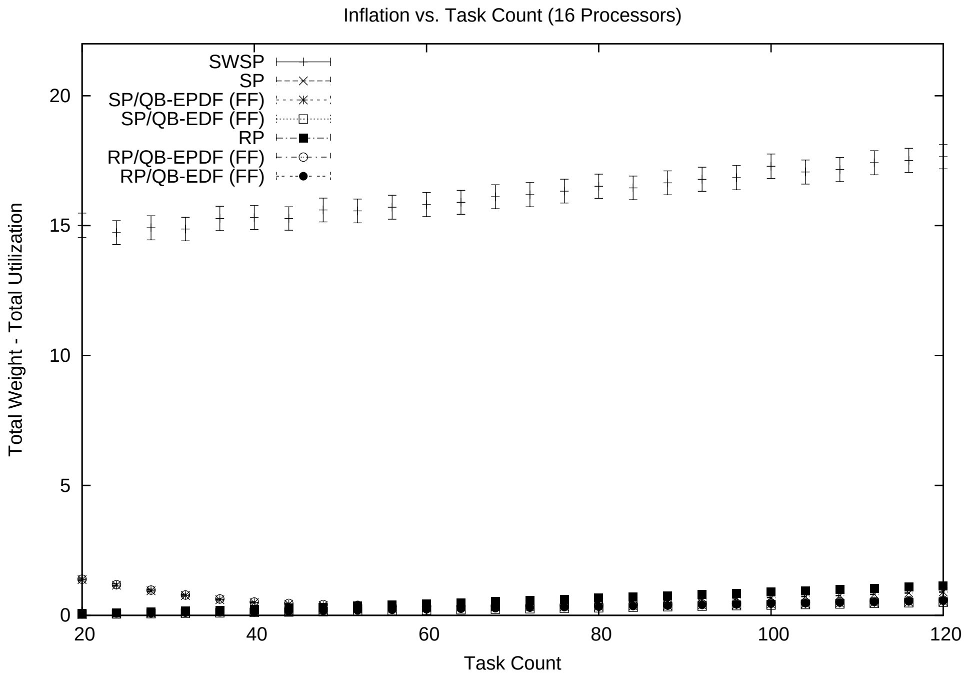


(b) 99% Confidence Interval

Figure 6.15: Plots show how inflation varies as the task count is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

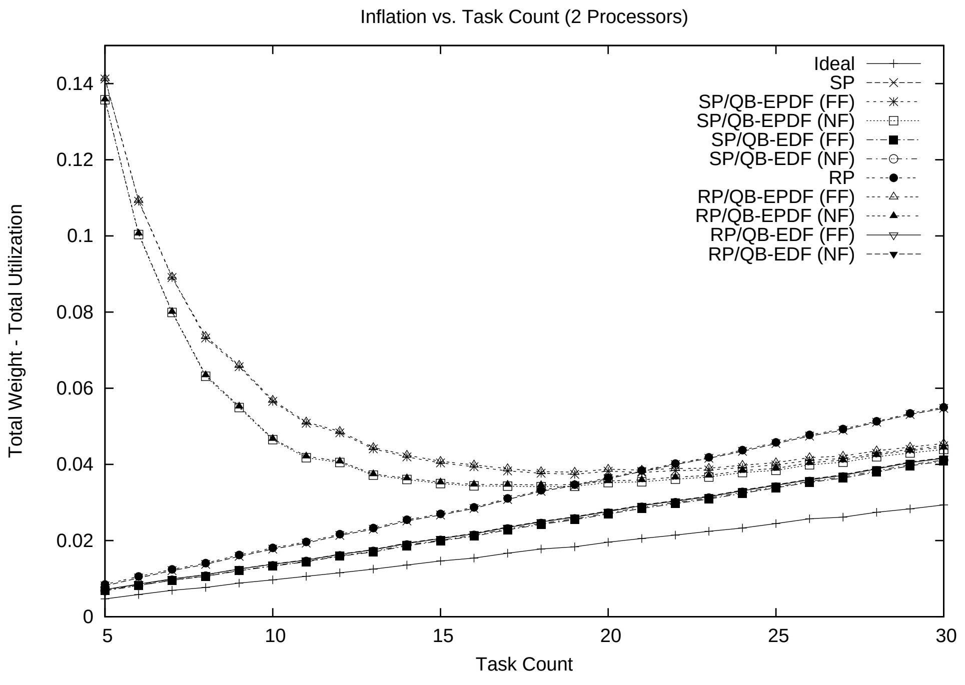


(a) Sample Means Only

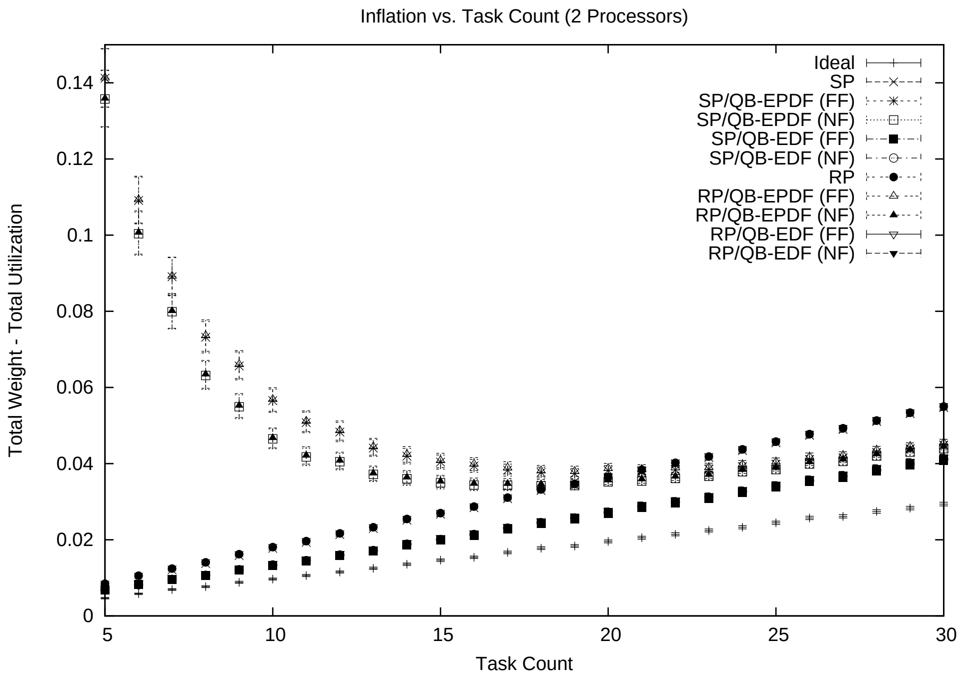


(b) 99% Confidence Interval

Figure 6.16: Plots show how inflation varies as the task count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

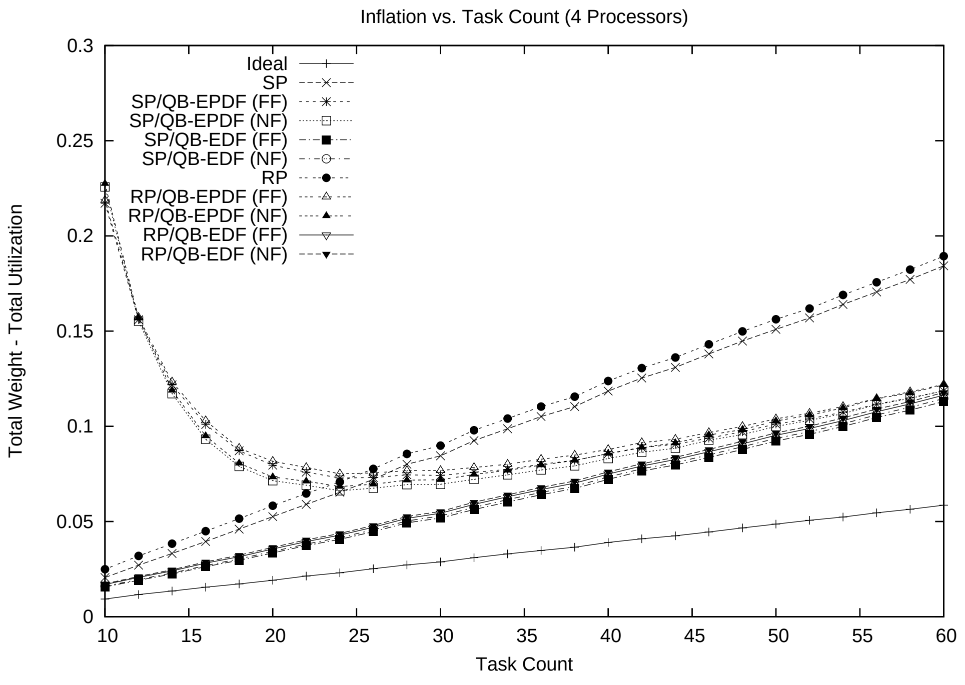


(a) Sample Means Only

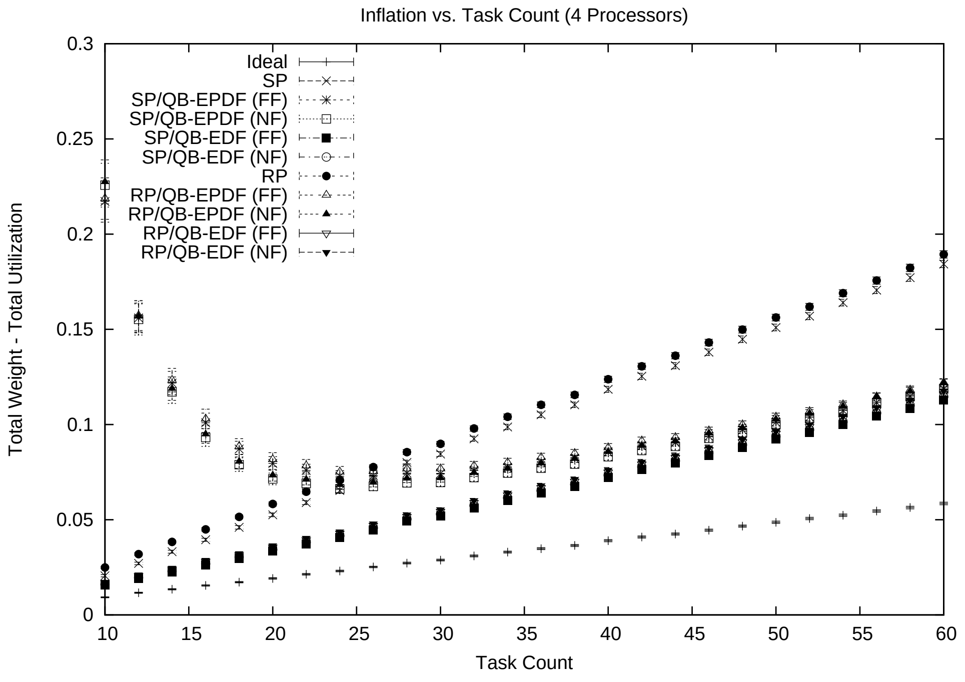


(b) 99% Confidence Interval

Figure 6.17: Plots show how inflation varies as the task count is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

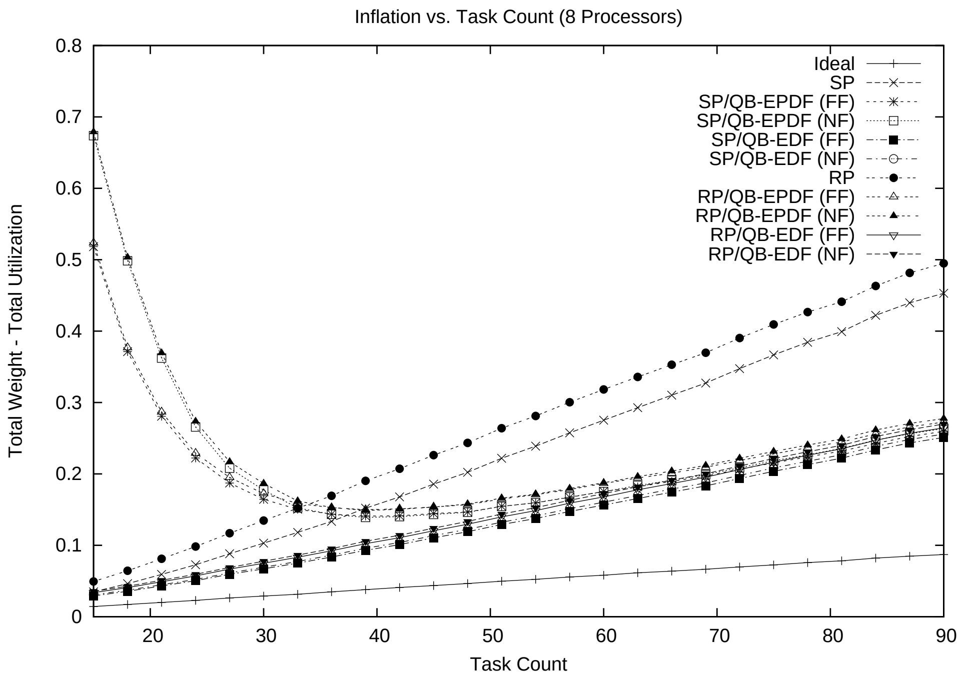


(a) Sample Means Only

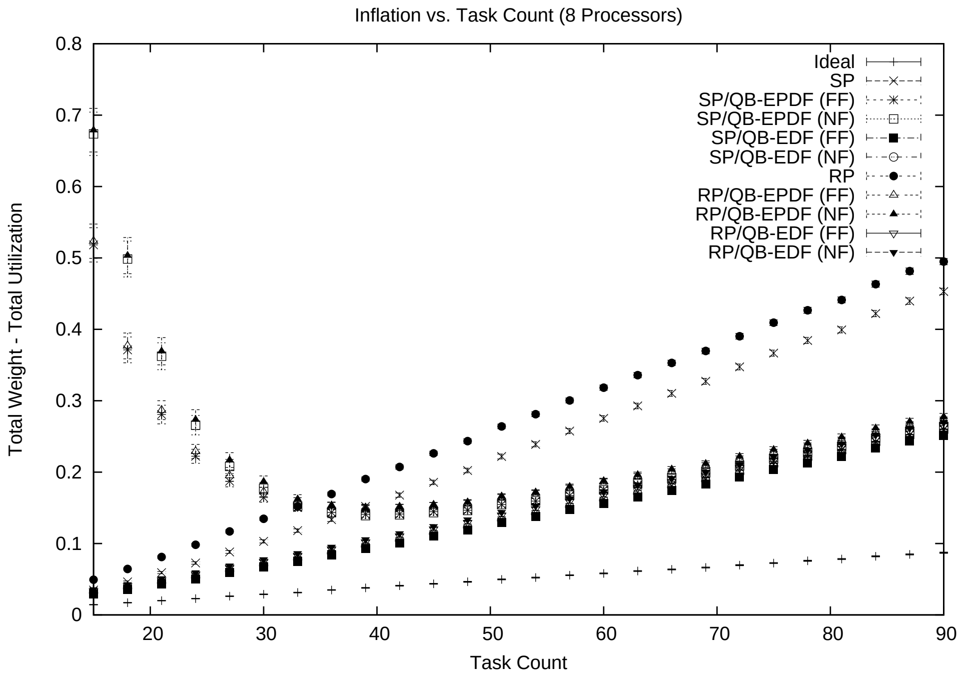


(b) 99% Confidence Interval

Figure 6.18: Plots show how inflation varies as the task count is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

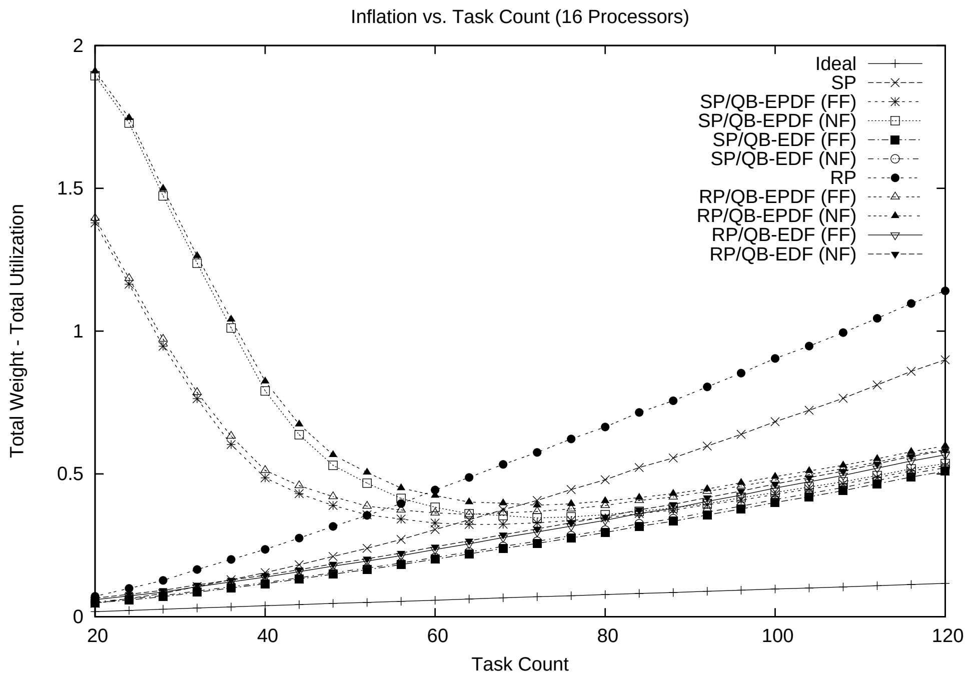


(a) Sample Means Only

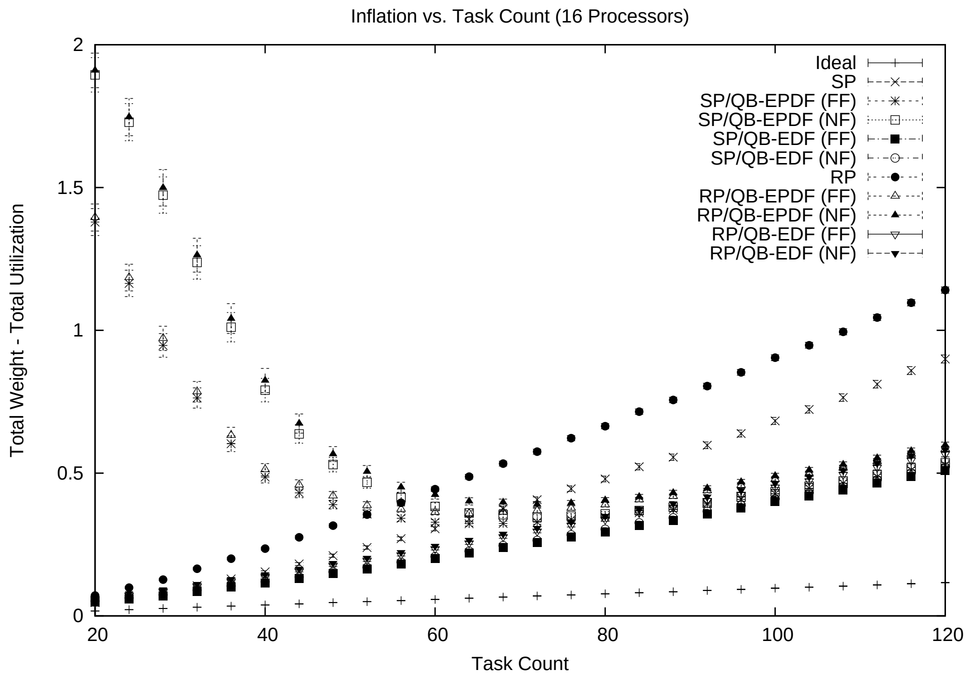


(b) 99% Confidence Interval

Figure 6.19: Plots show how inflation varies as the task count is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

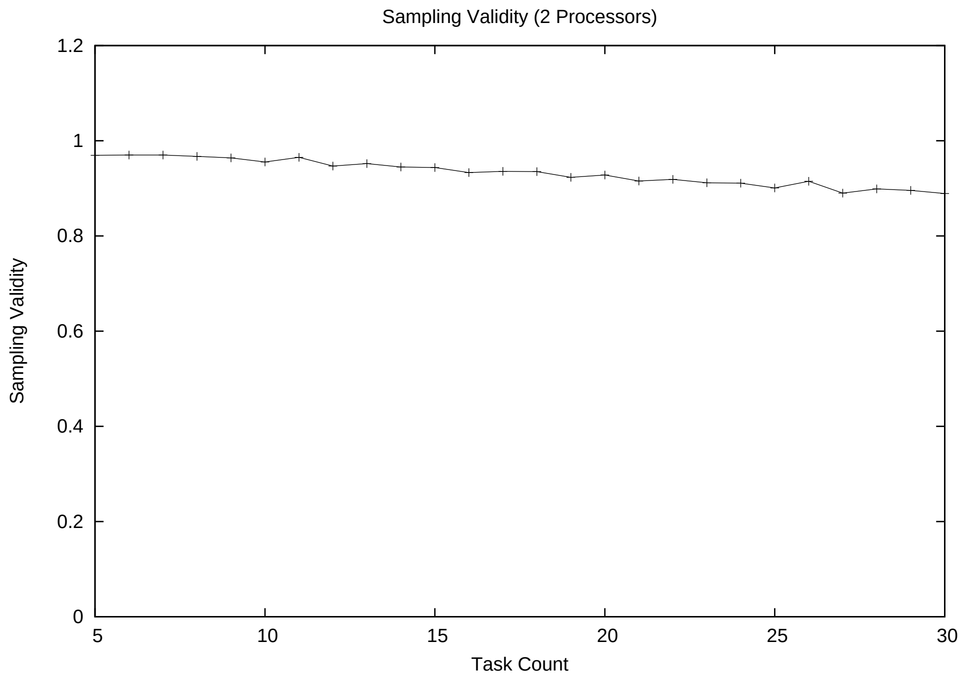


(a) Sample Means Only

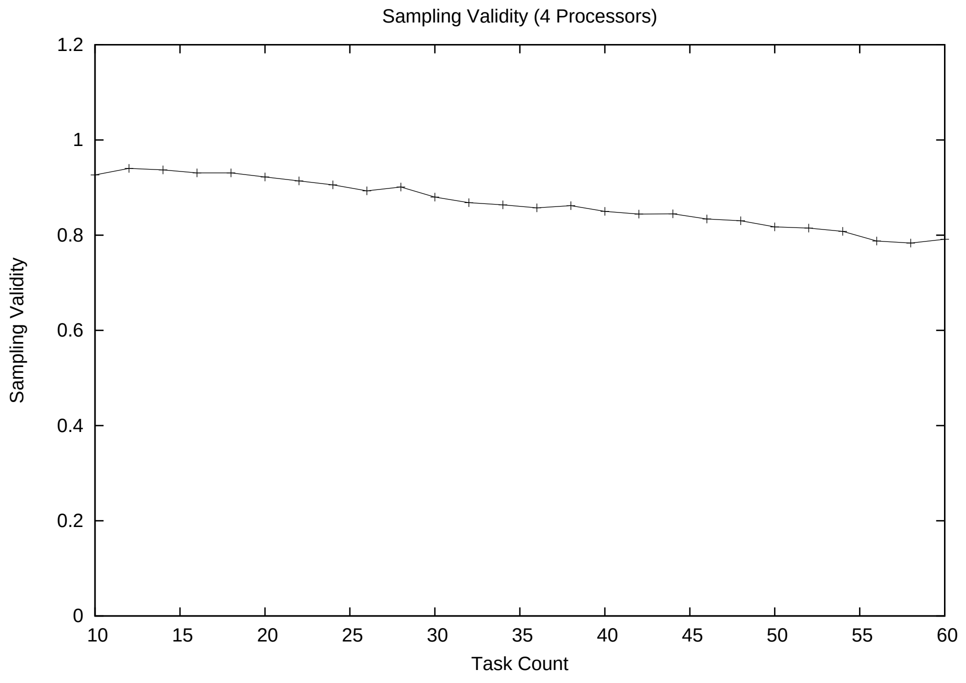


(b) 99% Confidence Interval

Figure 6.20: Plots show how inflation varies as the task count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

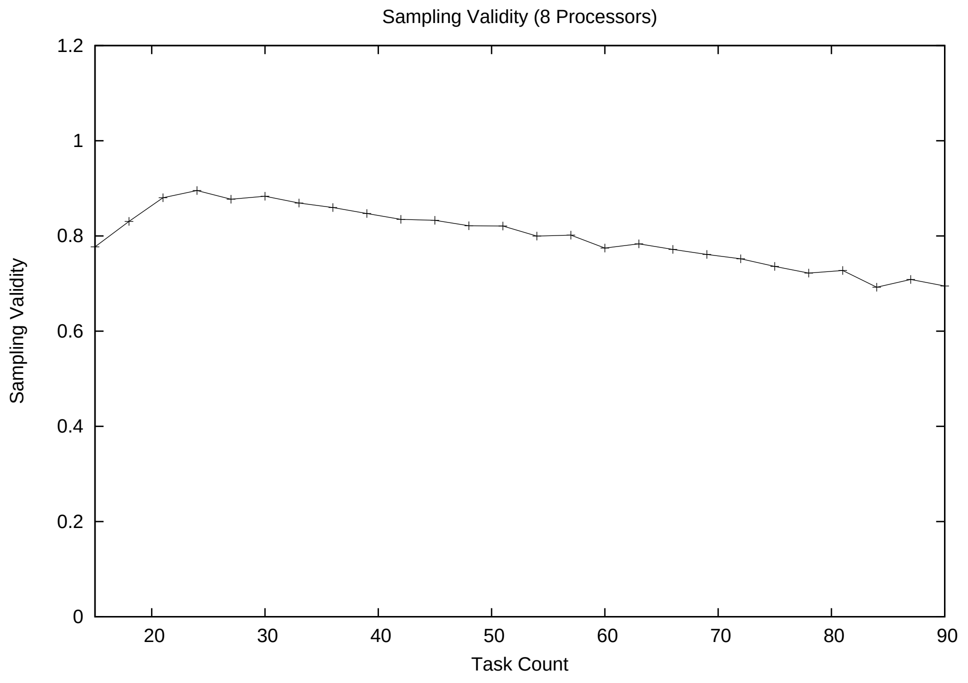


(a) $M = 2$

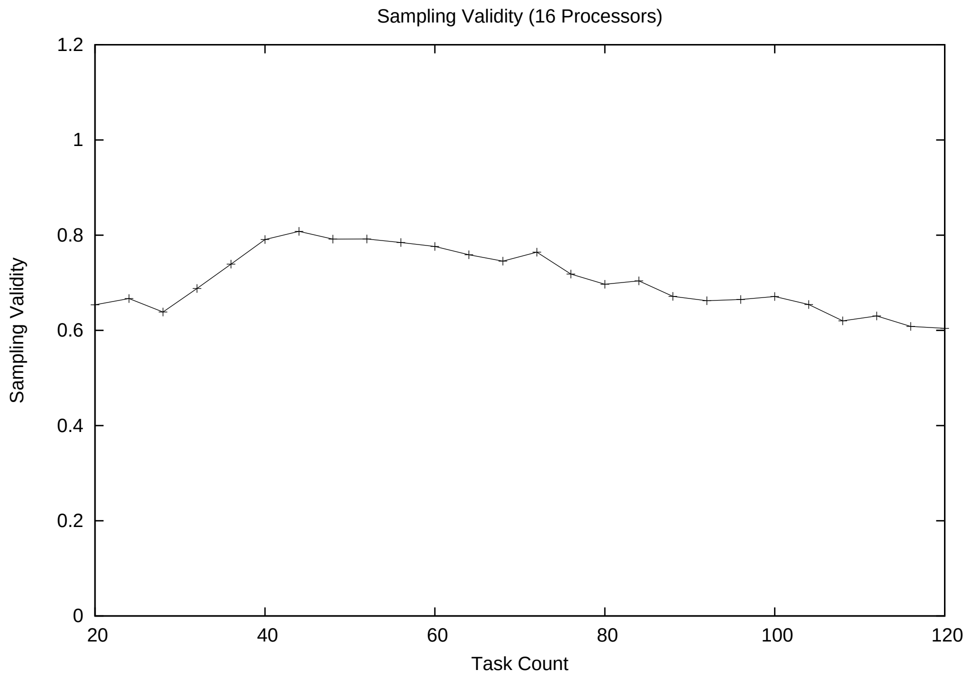


(b) $M = 4$

Figure 6.21: Plots show the sampling validity for the graphs showing inflation plotted against the task count when (R3) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



(b) $M = 16$

Figure 6.22: Plots show the sampling validity for the graphs showing inflation plotted against the task count when (R3) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

reweighting overhead.) Under these protocols, blocking estimates are based upon either the $M-1$ or $2(M-1)$ worst competitors, depending on the protocol. As the task count increases, more critical sections are introduced. As a result, the blocking produced by the worst-case subset of competitors tends to increase also. The reason for this is easily illustrated by an example. Suppose that n values are randomly selected from the integer range $1 \cdots 50$ and the largest value of the n is selected. When $n = 1$, the selected value is not likely to exceed 45. However, when $n = 100$, this is likely to be the case. In general, the selected value is likely to be higher for larger values of n. By similar reasoning, locking overhead tends to increase with the task count, at least when task sets are randomly generated.

Observation 6.2 *The RP scales worse with M than the SP.*

Explanation: This trend is predicted by the analysis presented earlier and can be seen in Figures 6.17–6.20. Specifically, recall that the SP enforces FIFO prioritization of requests across slots, while the RP does not. Because of this, a request can be blocked twice by the same worst-case collection of $M-1$ competing requests under the RP, but only once by the worst-case collection of $2(M-1)$ competing requests under the SP. The observed property follows from the fact that these counts are both functions of M.

Observation 6.3 *QB-EPDF supertasking provides a benefit on average on two, four, eight, and sixteen processors when the task count exceeds 20, 25, 40, and 70, respectively.*

Explanation: Figures 6.17–6.20 show that the performance of QB-EPDF supertasking varies significantly. Apparently, the mediocre performance suggested by the table summaries can largely be attributed to the high reweighting overhead suffered at low task counts.

Impact of the lock count. Figures 6.23, 6.24, 6.25, and 6.26 show how inflation varies with the lock count on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.27, 6.28, 6.29, and 6.30 show the remaining approaches. Plots of the sampling validity are shown in Figures 6.31 and 6.32.

Observation 6.4 *Locking overhead increases with increasing lock count under the SWSP.*

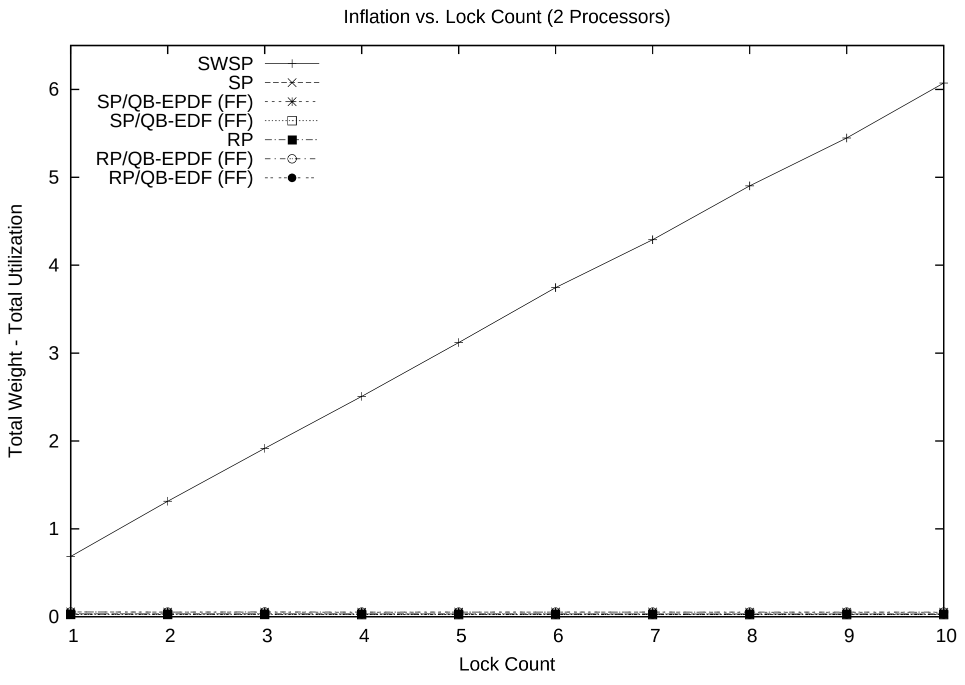
Explanation: Based on Figures 6.23–6.26, the most significant overhead under the SWSP is the server weights. Since one server was created for each lock in this study, the inflation suffered under the SWSP increases approximately linearly with the lock count.

Observation 6.5 *Locking overhead decreases with increasing lock count under the zone-based protocols.*

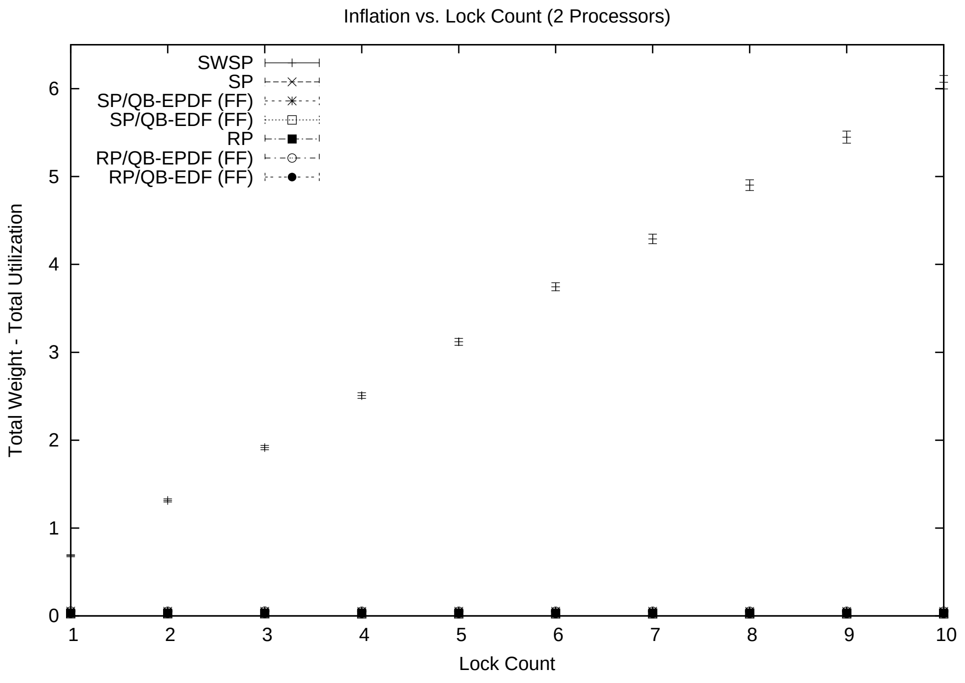
Explanation: This trend, shown in Figures 6.27–6.30, reflects the cost of locking granularity. Consider a task set that uses multiple locks. Suppose that this set of locks is replaced by a single universal lock that guards all critical sections. By imposing such coarse-grained locking, contention is made worse because parallel execution of critical sections is more constrained. Hence, in these graphs, the lock count reflects the granularity of locking, and *not* the number of critical sections or degree of sharing. The high inflation at low lock counts is likely the cause of the drop in sampling validity observed in the $M = 8$ and $M = 16$ cases.

Observation 6.6 *On eight or more processors, the use of supertasking appears to improve scalability with respect to the granularity of locking.*

Explanation: This trend can be seen by comparing the SP and RP curves to the supertasking curves in Figures 6.29 and 6.30. Since the supertasking approaches focus on reducing contention, this trend was expected.

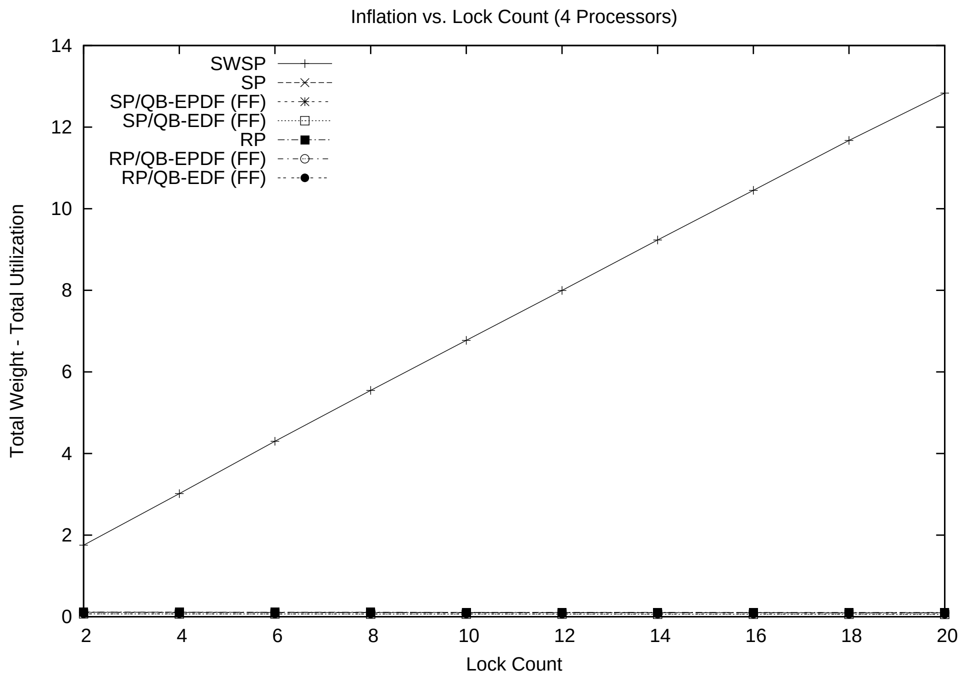


(a) Sample Means Only

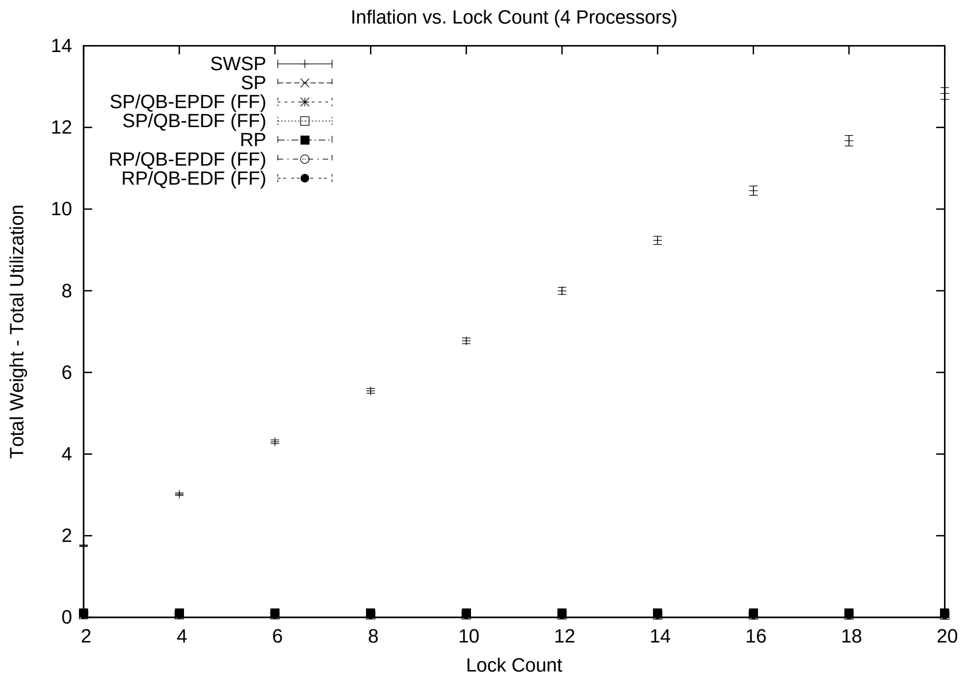


(b) 99% Confidence Interval

Figure 6.23: Plots show how inflation varies as the lock count is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

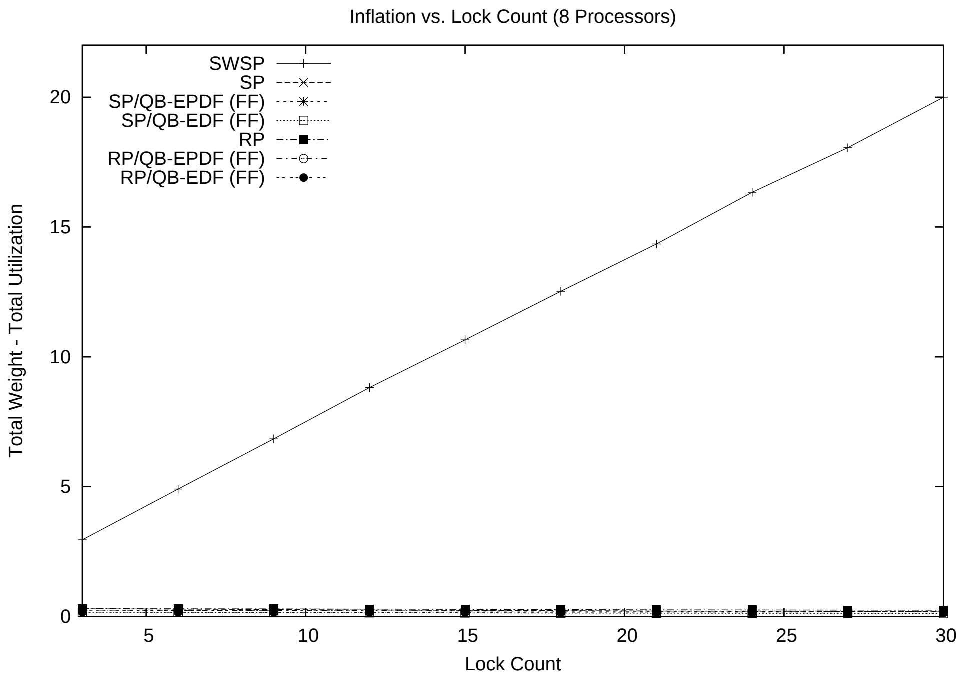


(a) Sample Means Only

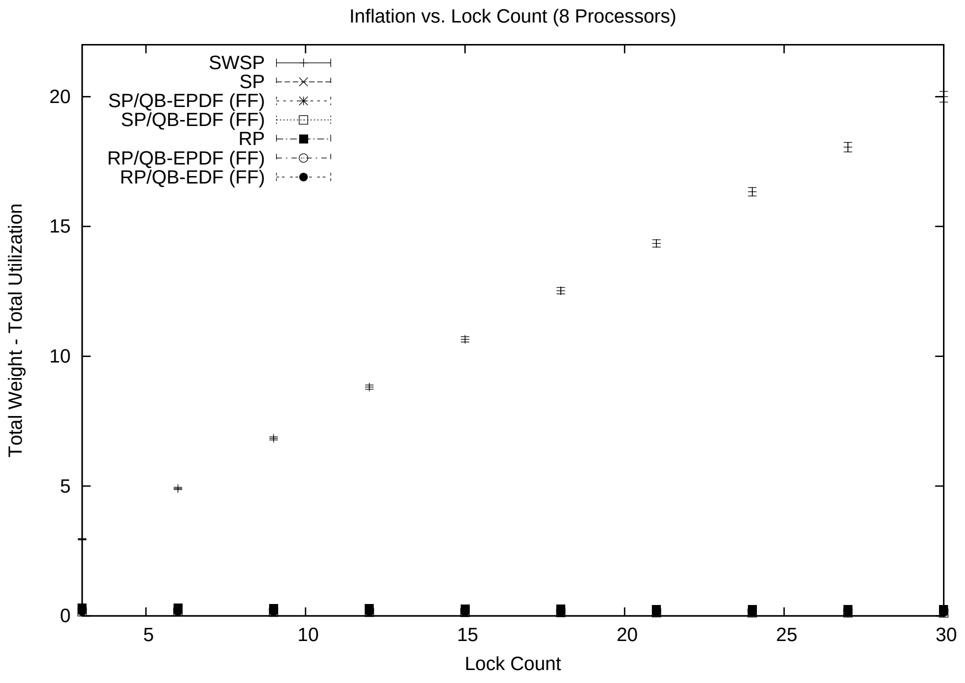


(b) 99% Confidence Interval

Figure 6.24: Plots show how inflation varies as the lock count is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

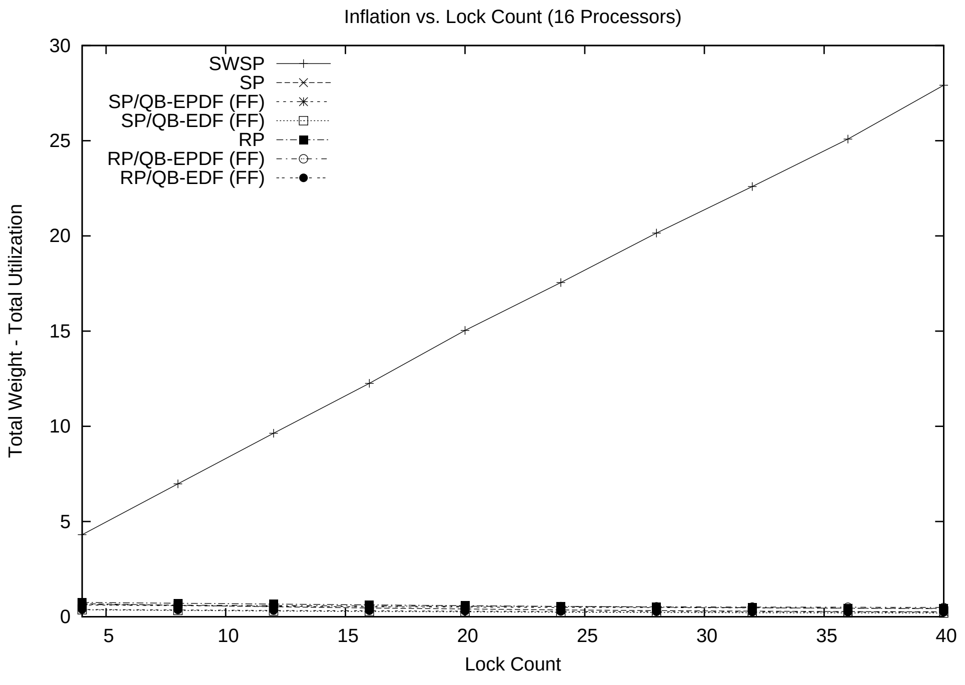


(a) Sample Means Only

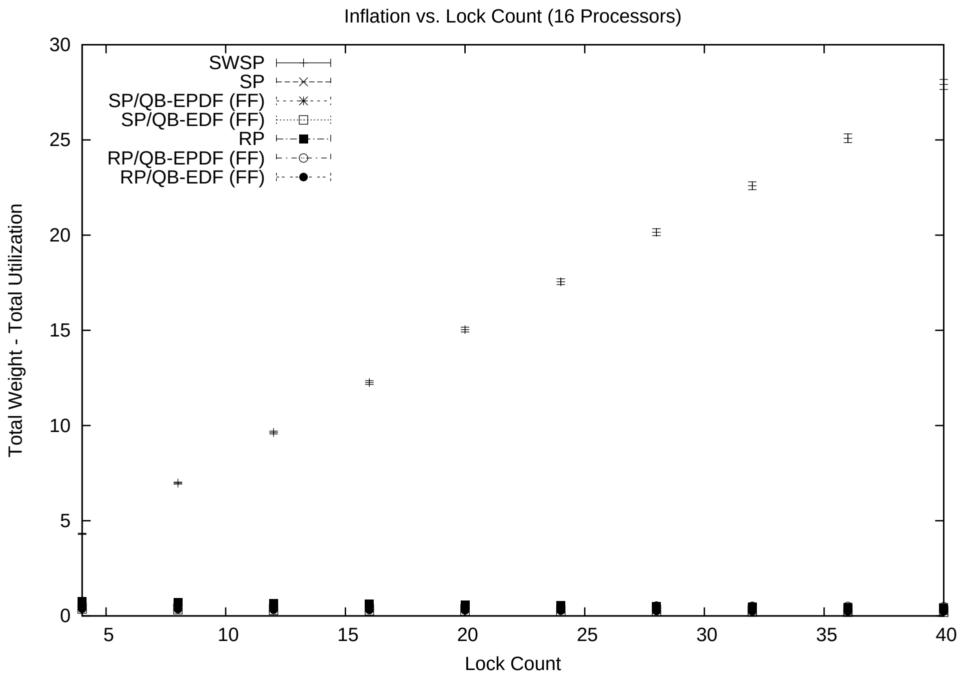


(b) 99% Confidence Interval

Figure 6.25: Plots show how inflation varies as the lock count is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

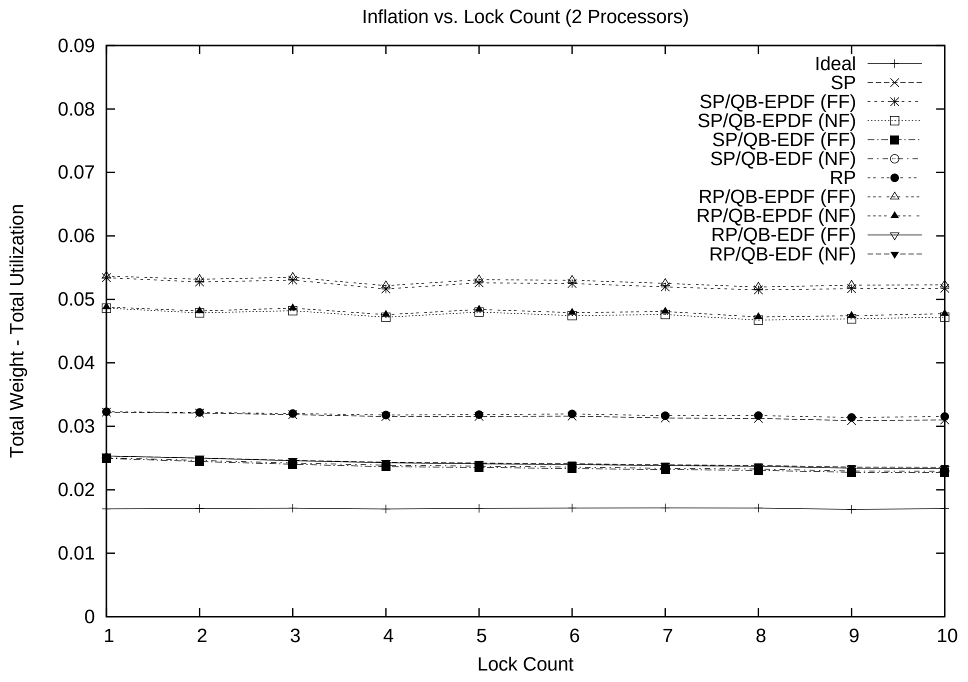


(a) Sample Means Only

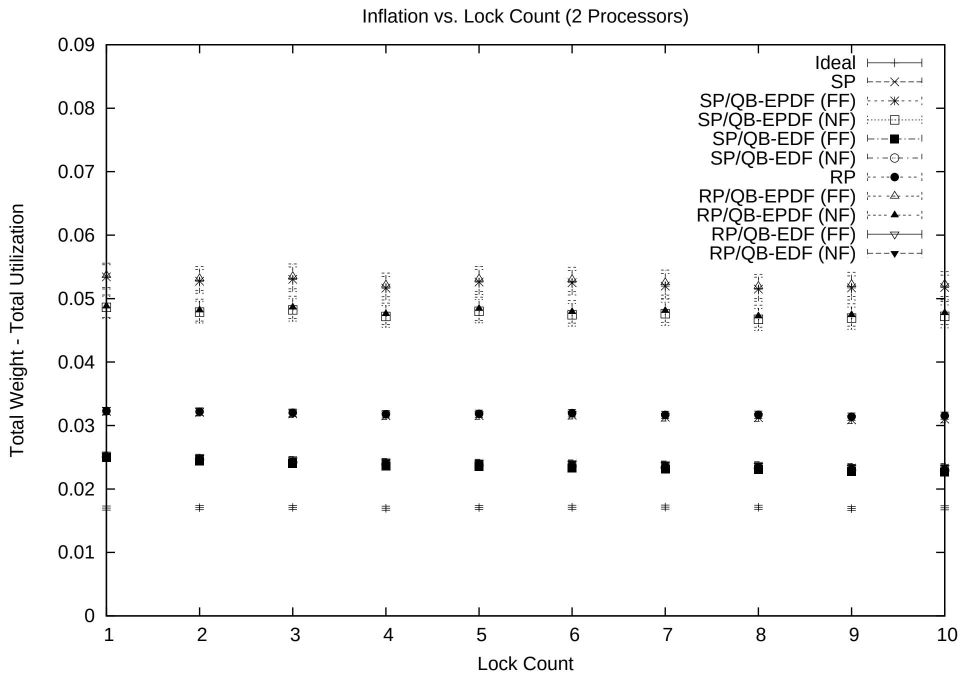


(b) 99% Confidence Interval

Figure 6.26: Plots show how inflation varies as the lock count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

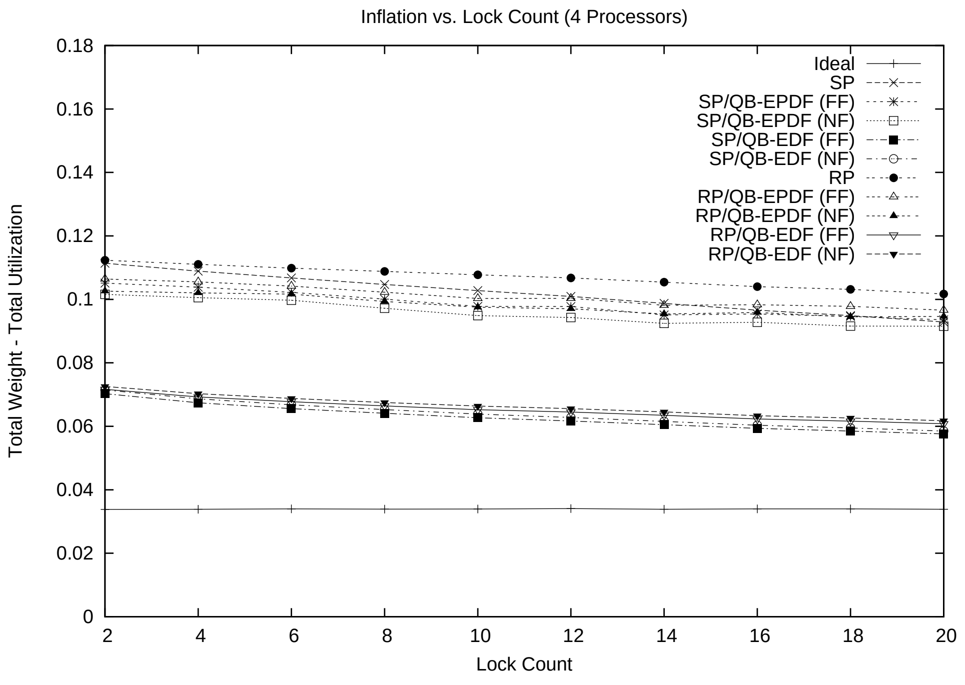


(a) Sample Means Only

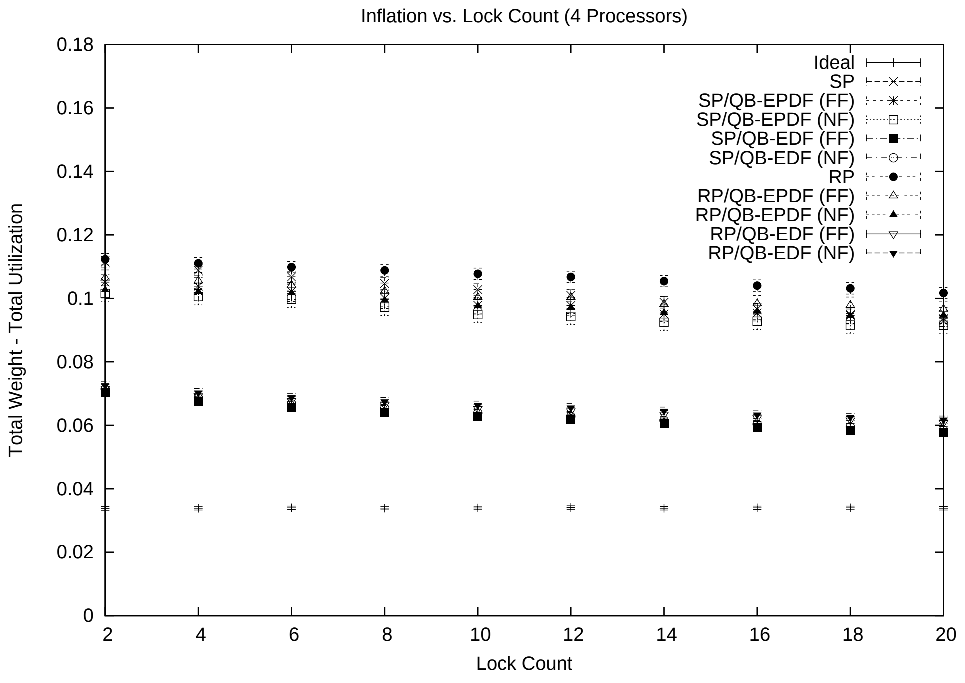


(b) 99% Confidence Interval

Figure 6.27: Plots show how inflation varies as the lock count is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

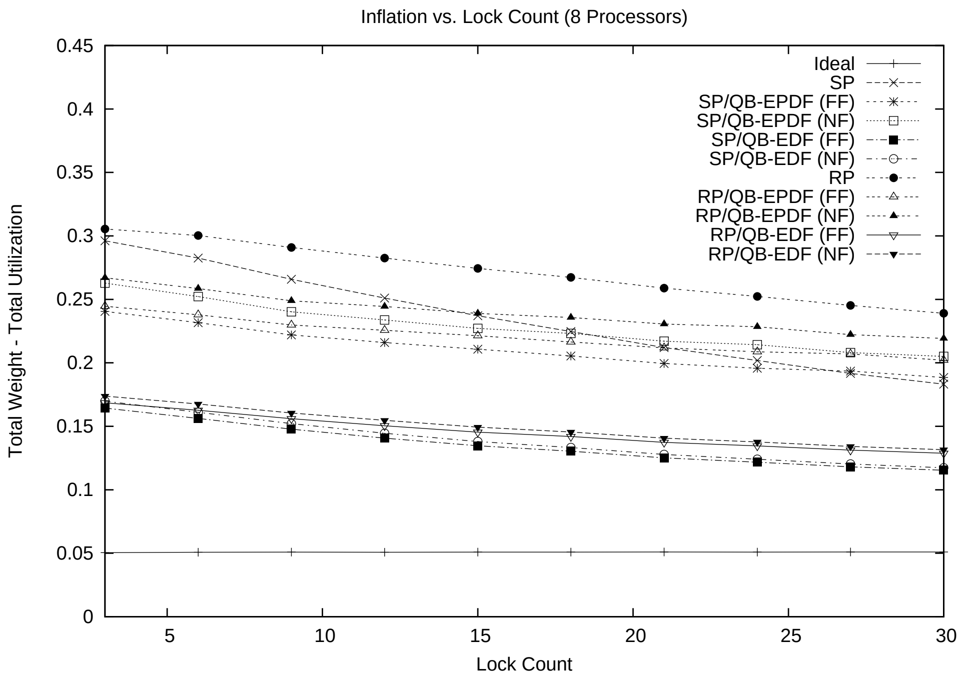


(a) Sample Means Only

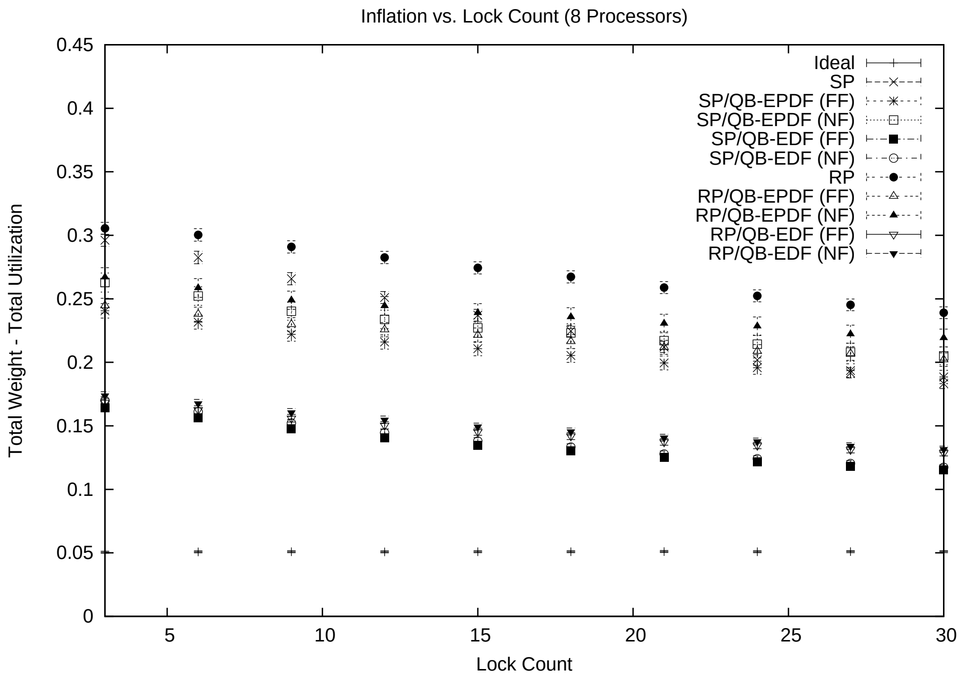


(b) 99% Confidence Interval

Figure 6.28: Plots show how inflation varies as the lock count is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

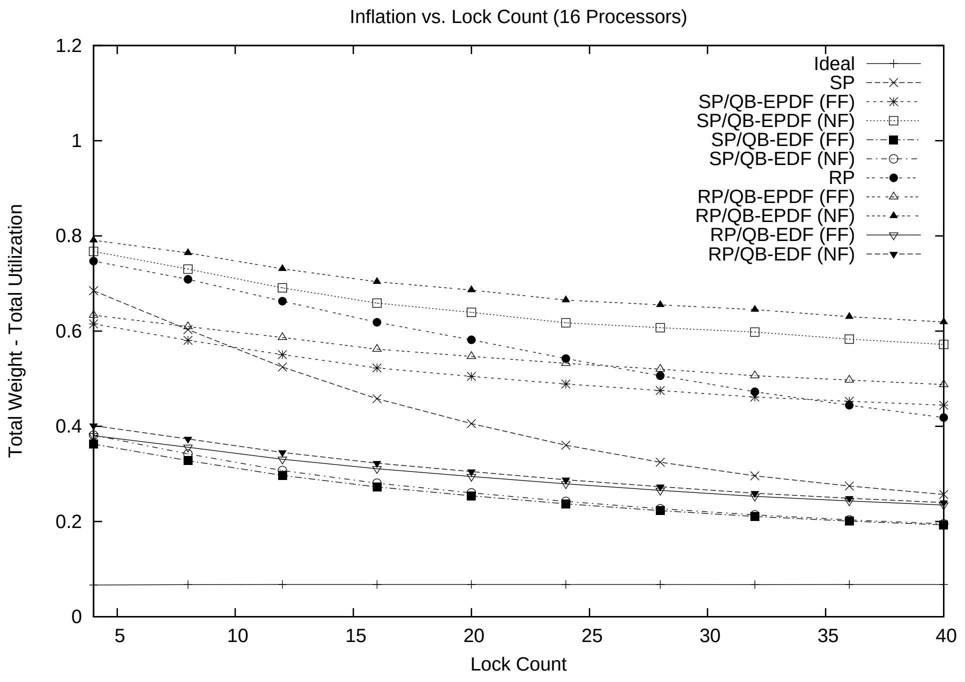


(a) Sample Means Only

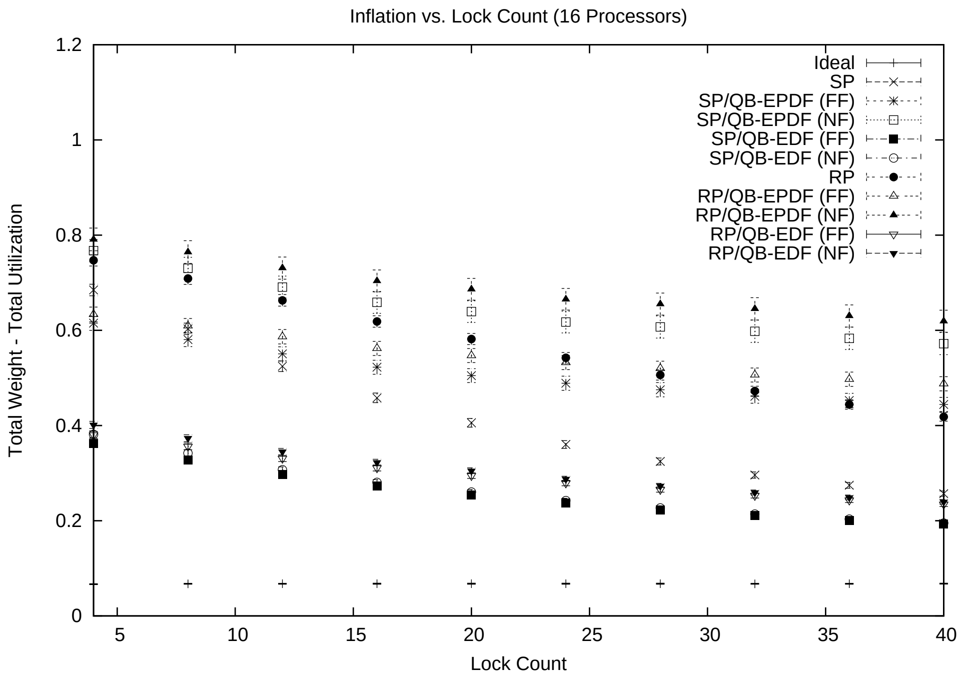


(b) 99% Confidence Interval

Figure 6.29: Plots show how inflation varies as the lock count is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

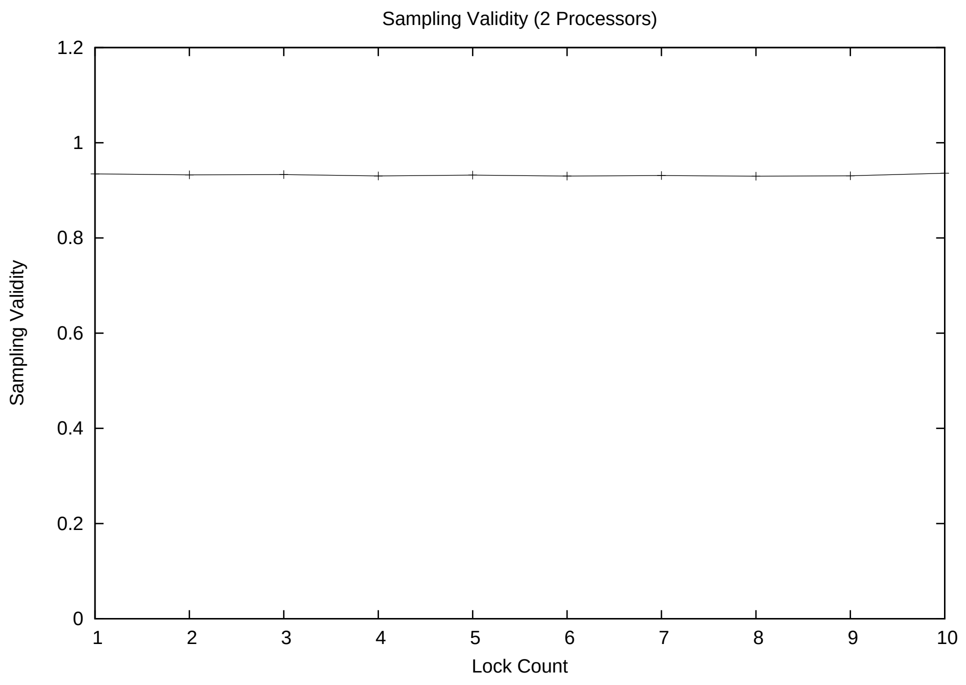


(a) Sample Means Only

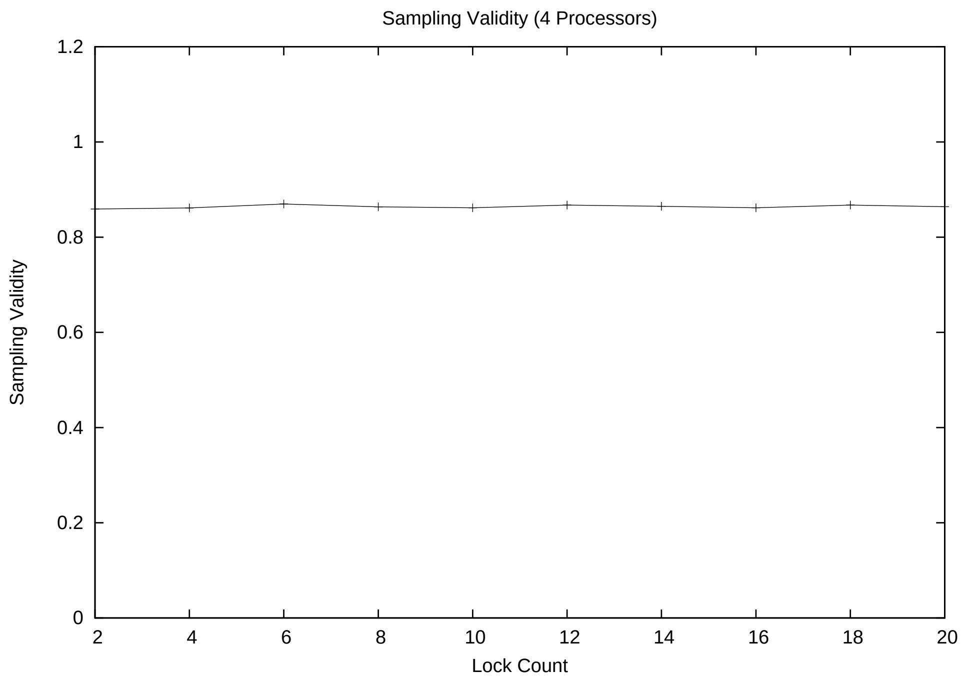


(b) 99% Confidence Interval

Figure 6.30: Plots show how inflation varies as the lock count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

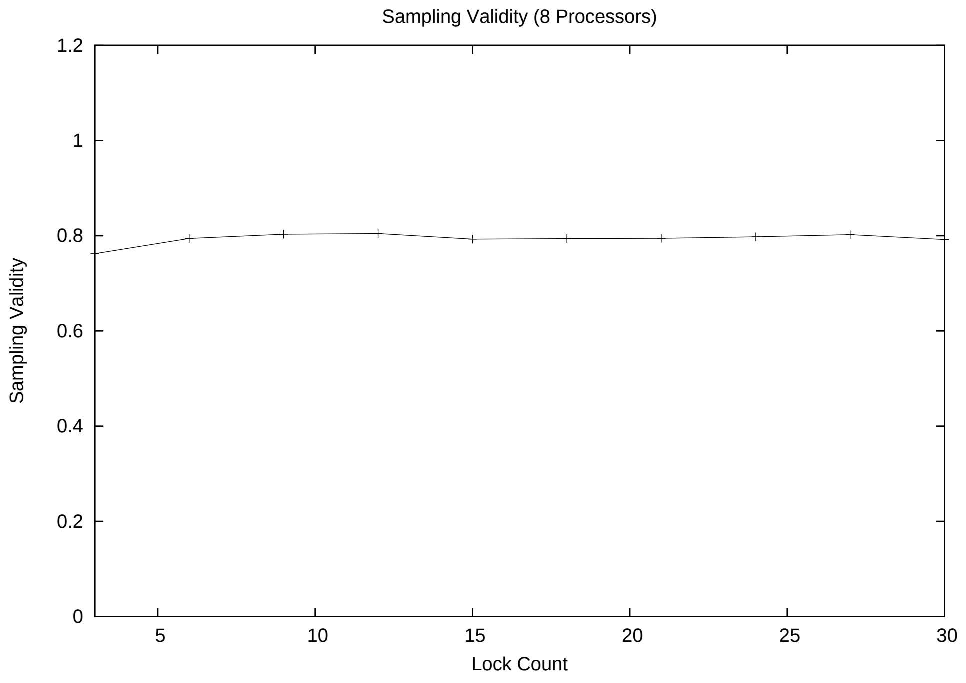


(a) $M = 2$

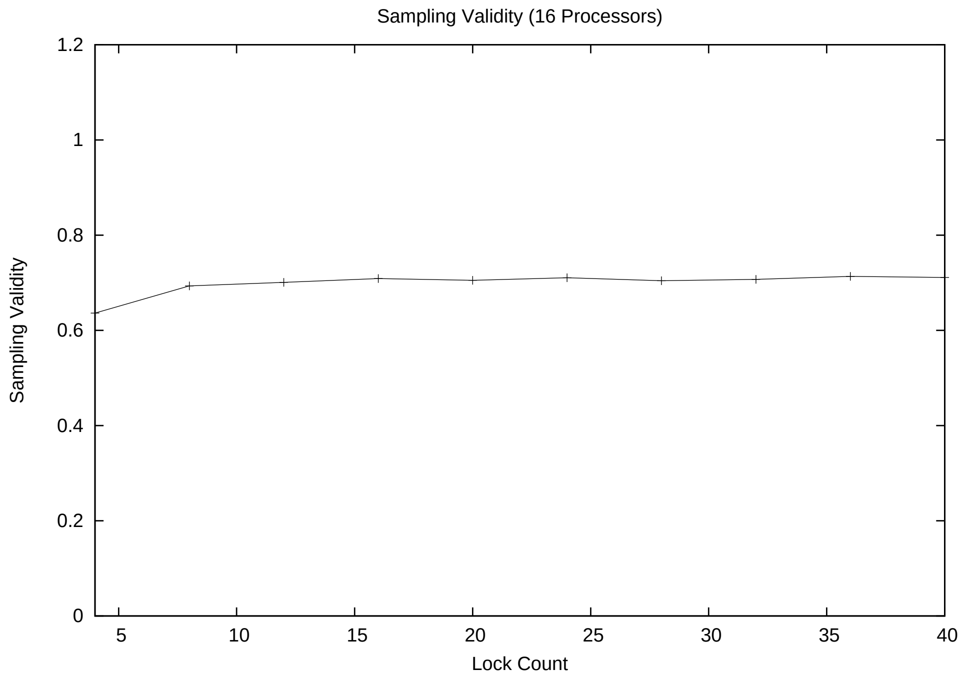


(b) $M = 4$

Figure 6.31: Plots show the sampling validity for the graphs showing inflation plotted against the lock count when (R3) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



(b) $M = 16$

Figure 6.32: Plots show the sampling validity for the graphs showing inflation plotted against the lock count when (R3) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

Impact of utilization. Figures 6.33, 6.34, 6.35, and 6.36 show how inflation varies with the total utilization on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.37, 6.38, 6.39, and 6.40 show the remaining approaches. Plots of the sampling validity are shown in Figures 6.41 and 6.42.

With the exception of the SP, RP, and SWSP trends, all trends shown in these graphs have been discussed previously in Chapters 4 and 5. Hence, we restrict attention to these three curves when considering trends below.

Observation 6.7 *Inflation increases linearly, but slowly, under the SWSP with increasing utilization.*

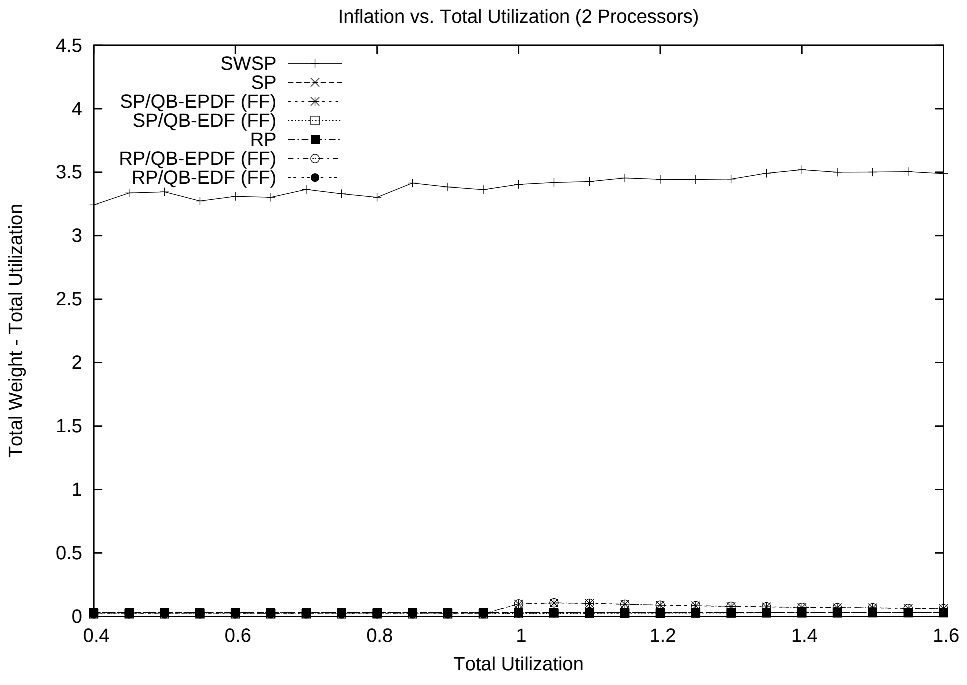
Explanation: This trend can be seen by comparing the far left and far right sample means in Figures 6.33–6.36. The cause of the trend is not immediately apparent. This is likely an artifact of the rule used to assign weights to lock servers. Further study is needed.

Observation 6.8 *Locking overhead is independent of utilization under the SP and the RP.*

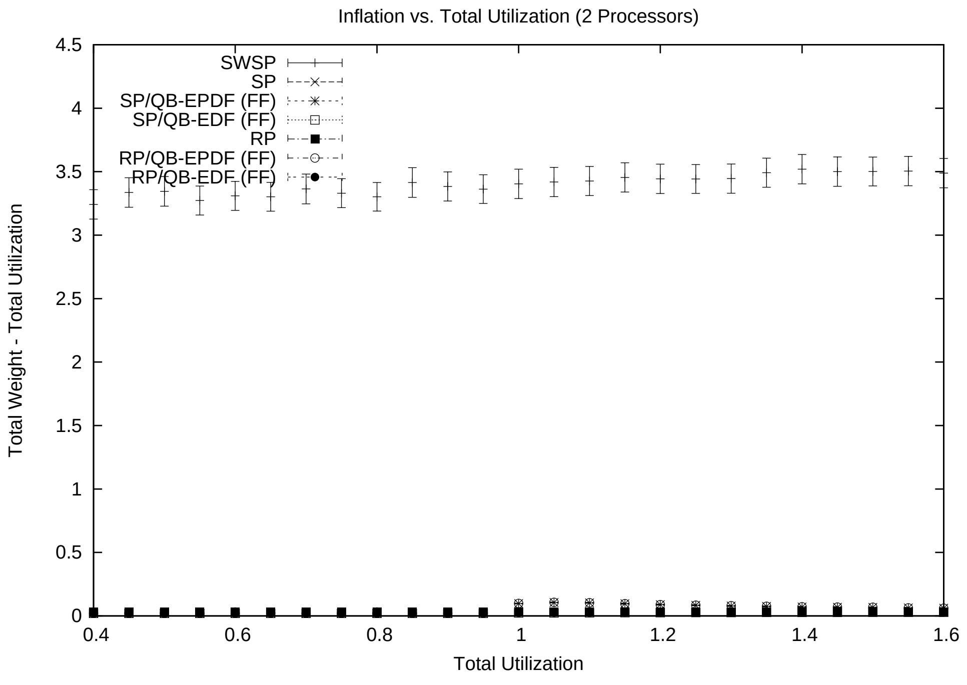
Explanation: This observation is based on Figures 6.37–6.40. (The odd behavior of the QB-EPDF-based measurements reflects changes in the reweighting overhead and was discussed earlier in Chapter 4.) Nothing in the analysis presented earlier suggested that locking overhead should vary with utilization. Hence, this behavior was expected.

Observation 6.9 *QB-EPDF supertasking tends to benefit the SP and the RP only when the utilization is below $\frac{M}{2}$.*

Explanation: This observation is also based on Figures 6.37–6.40. Higher utilizations tend to produce tasks with larger weights. As a result the number of tasks that can be placed in the

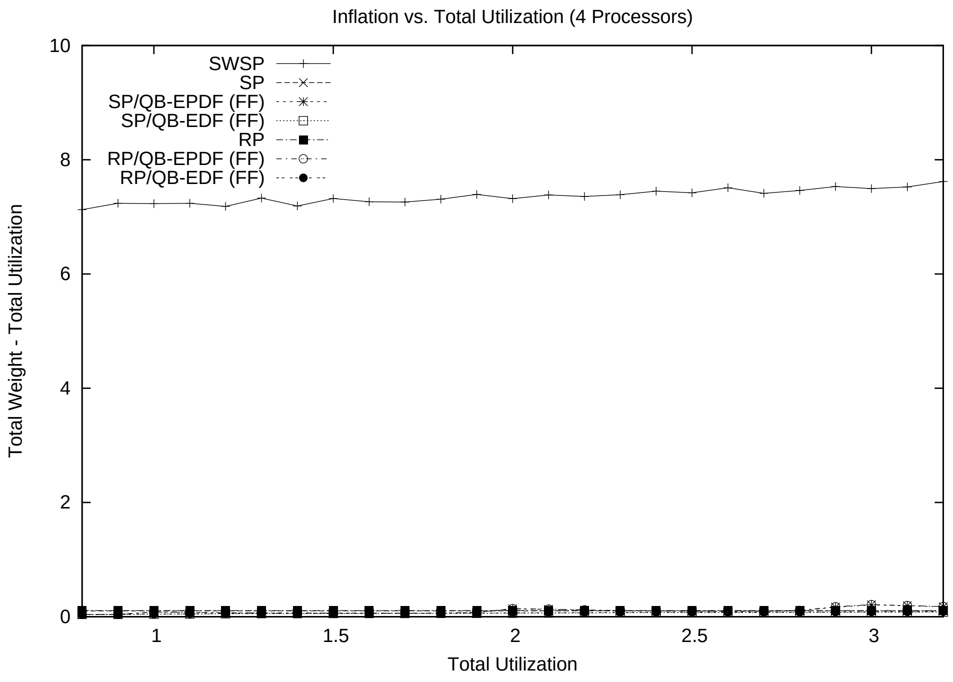


(a) Sample Means Only

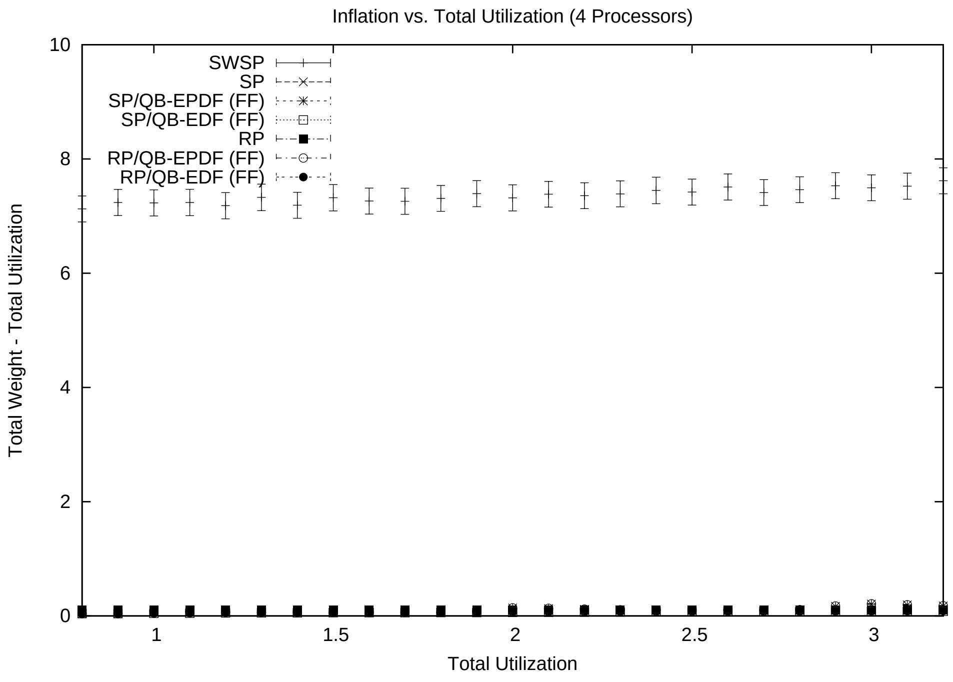


(b) 99% Confidence Interval

Figure 6.33: Plots show how inflation varies as utilization is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

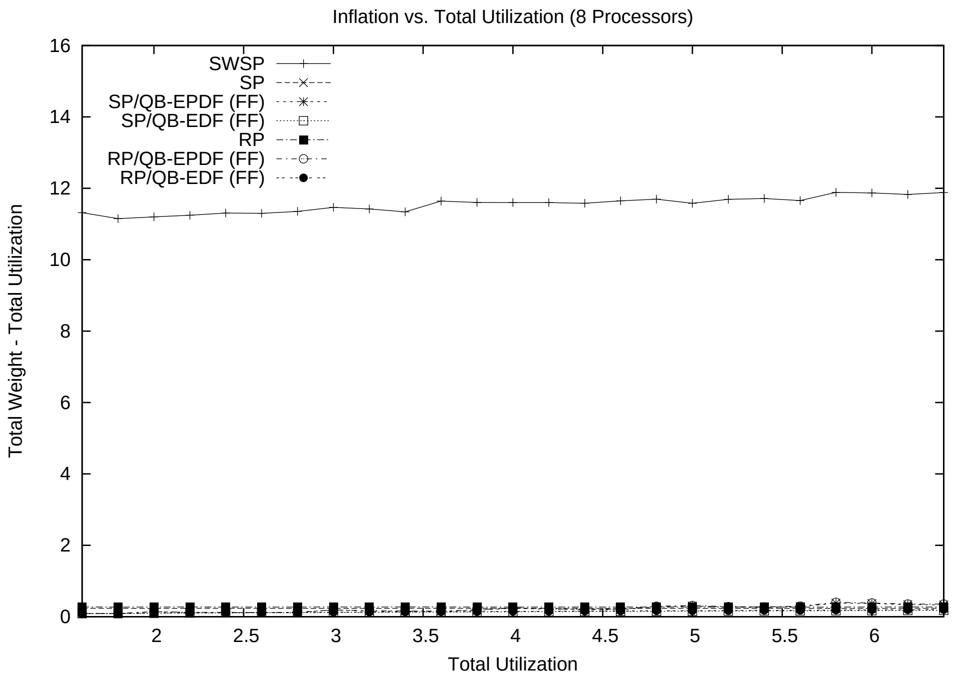


(a) Sample Means Only

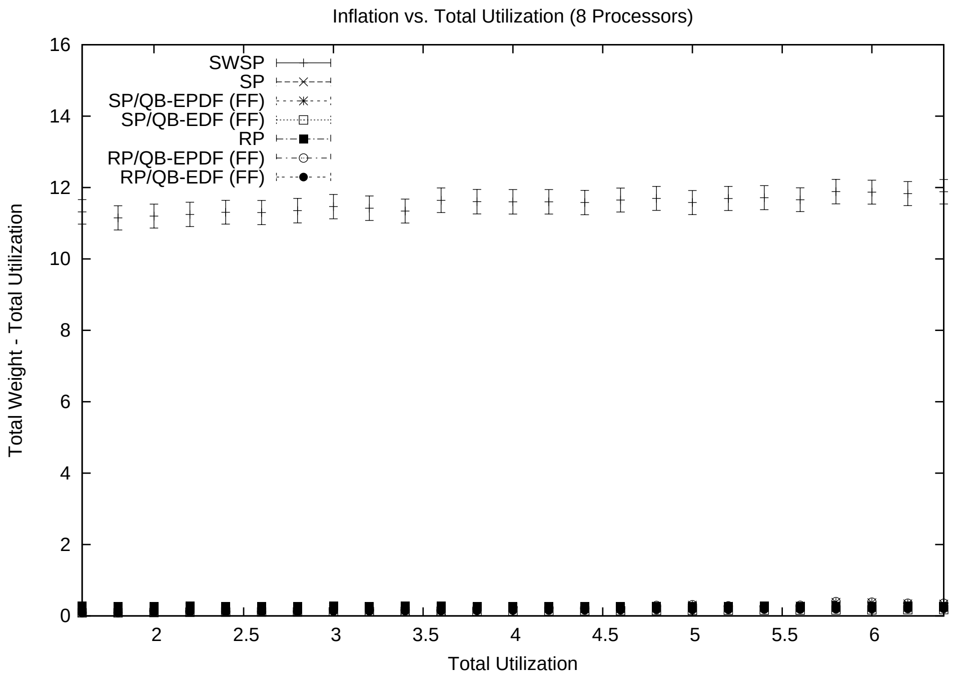


(b) 99% Confidence Interval

Figure 6.34: Plots show how inflation varies as utilization is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

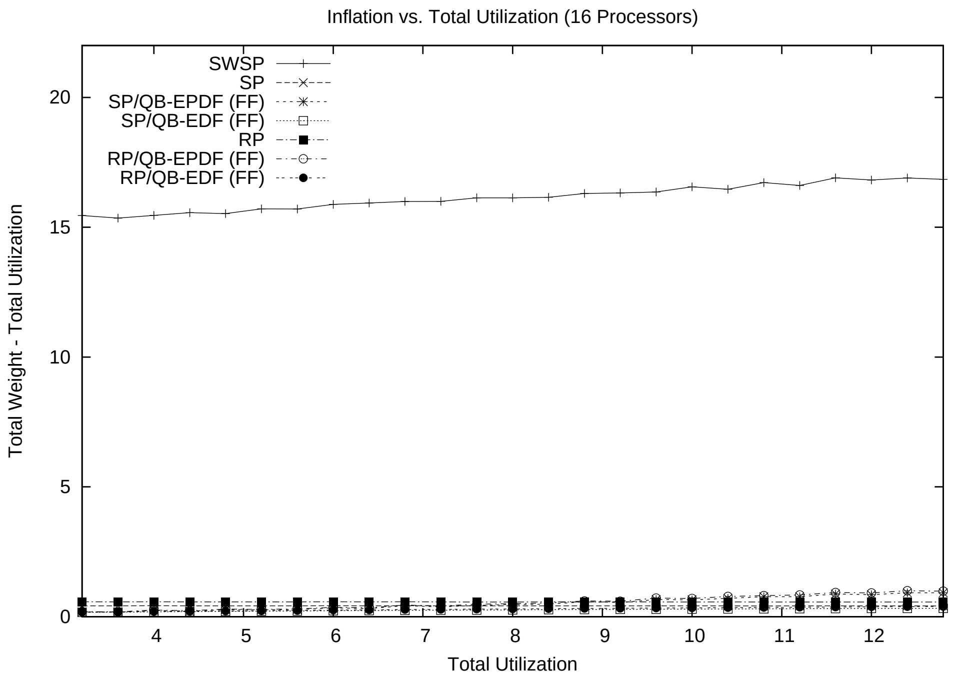


(a) Sample Means Only

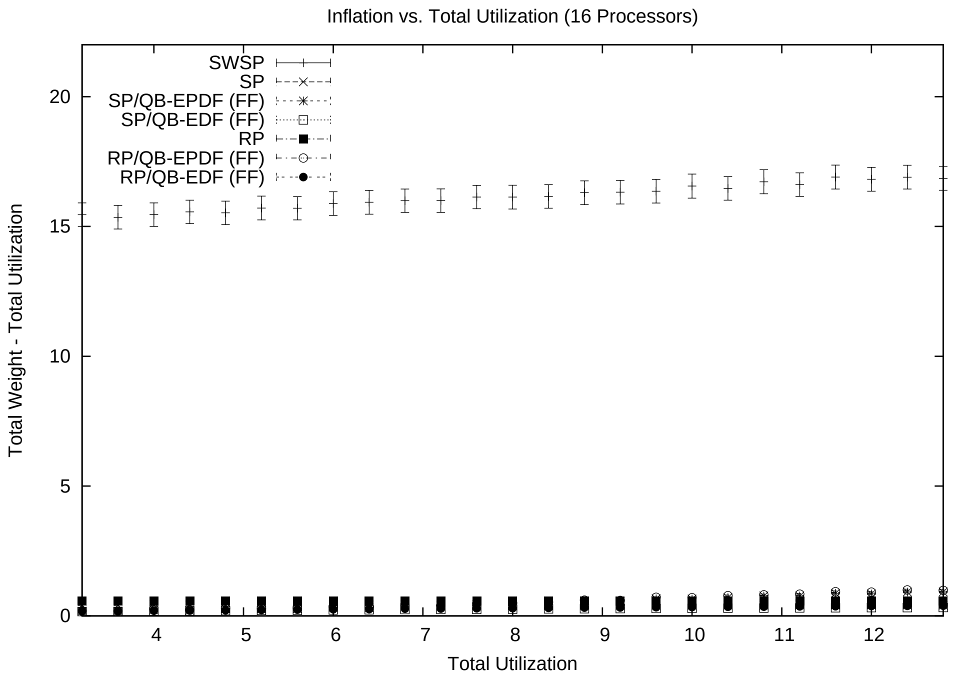


(b) 99% Confidence Interval

Figure 6.35: Plots show how inflation varies as utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

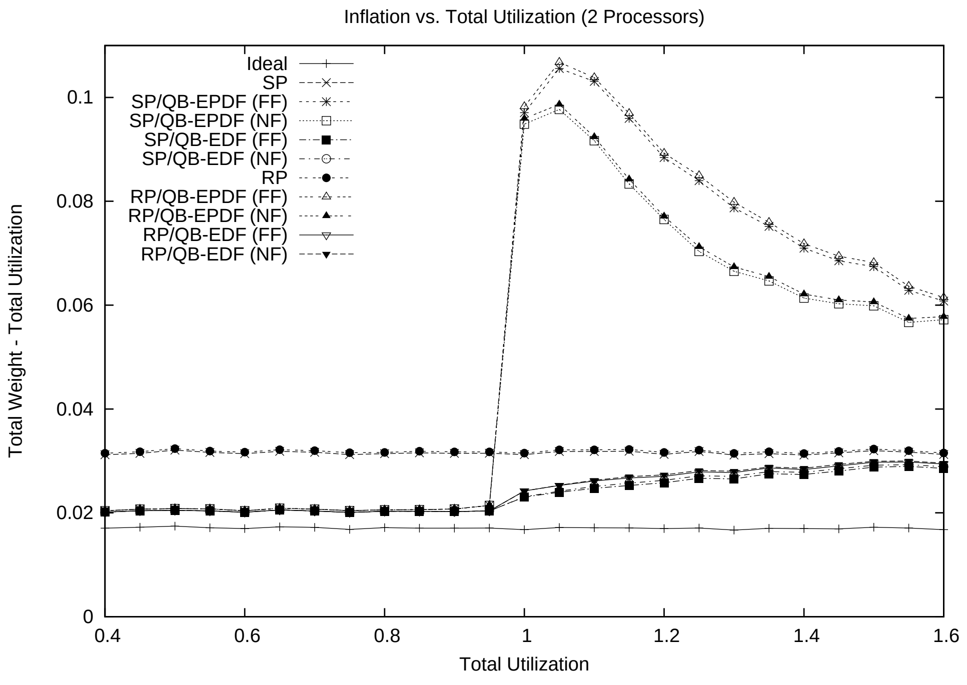


(a) Sample Means Only

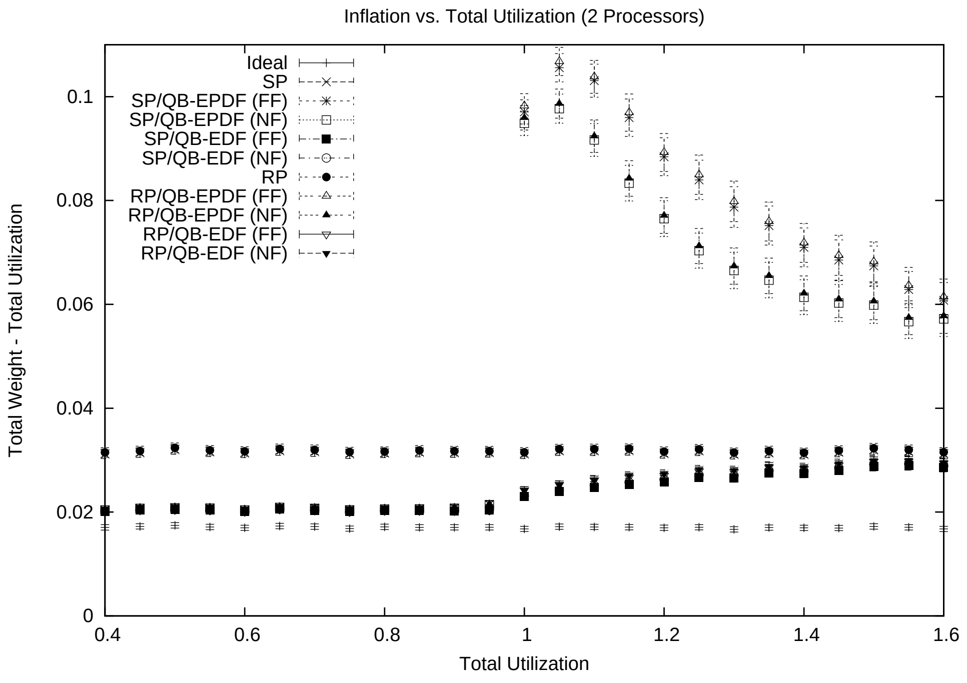


(b) 99% Confidence Interval

Figure 6.36: Plots show how inflation varies as utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

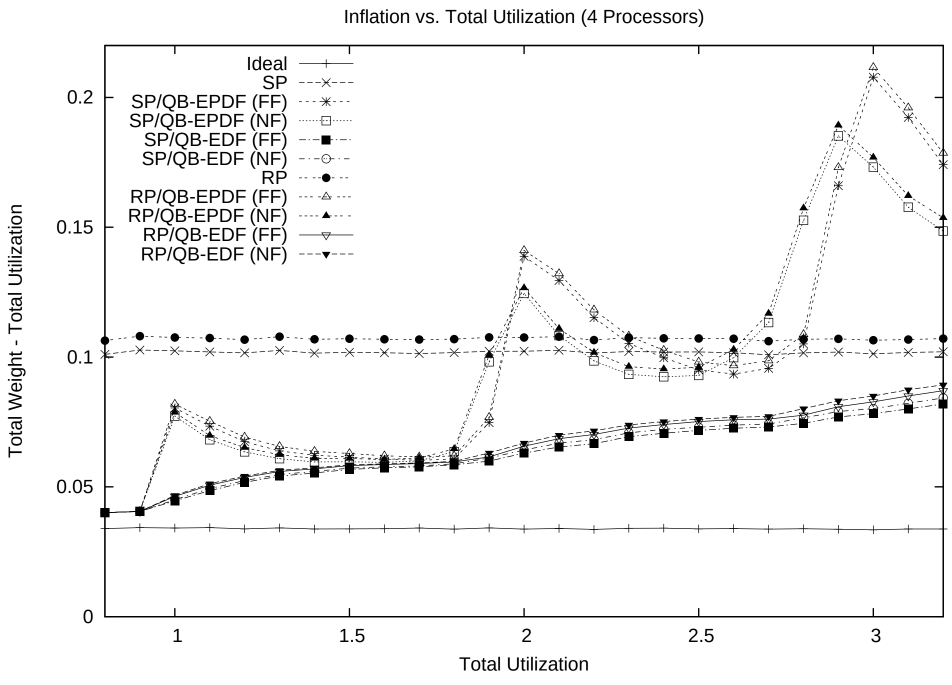


(a) Sample Means Only

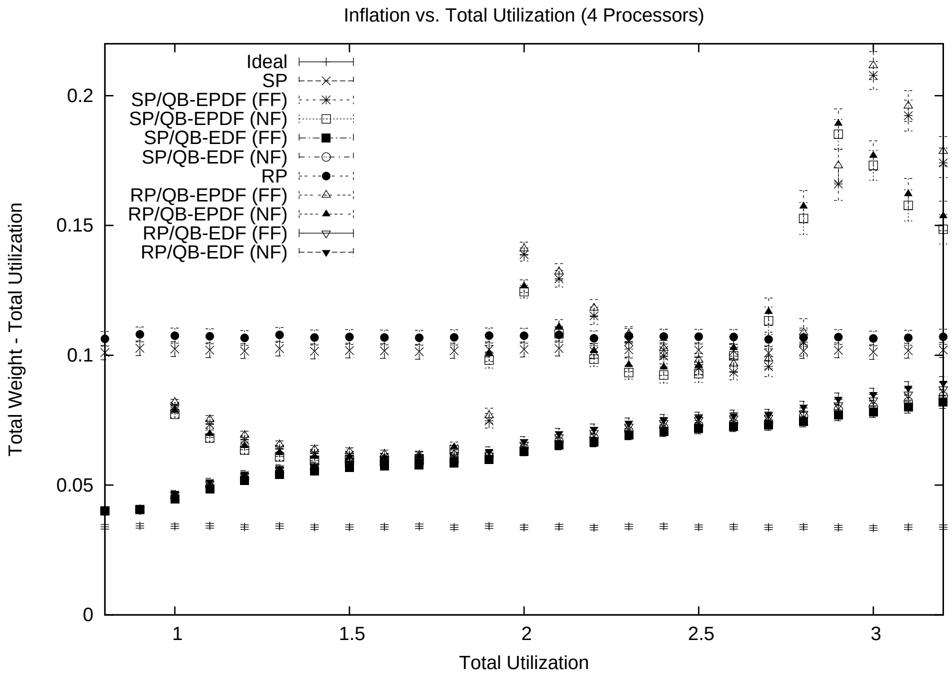


(b) 99% Confidence Interval

Figure 6.37: Plots show how inflation varies as utilization is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

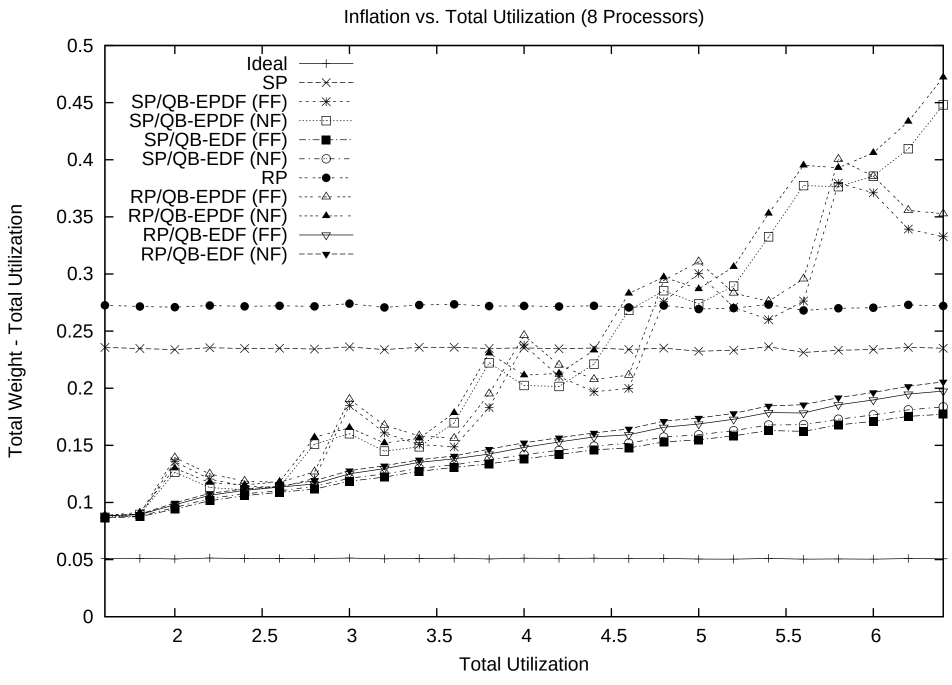


(a) Sample Means Only

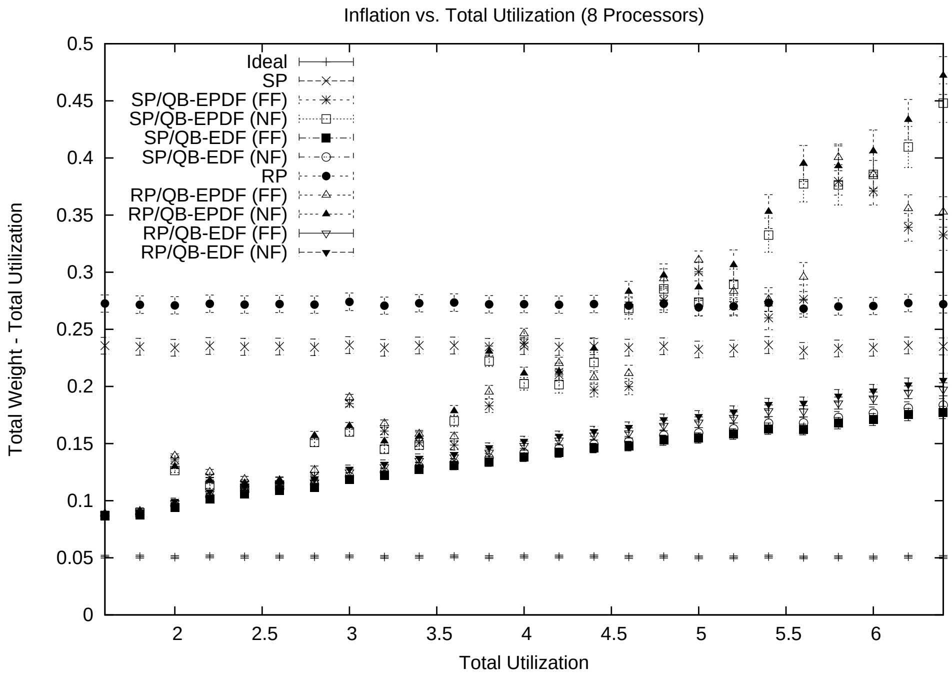


(b) 99% Confidence Interval

Figure 6.38: Plots show how inflation varies as utilization is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

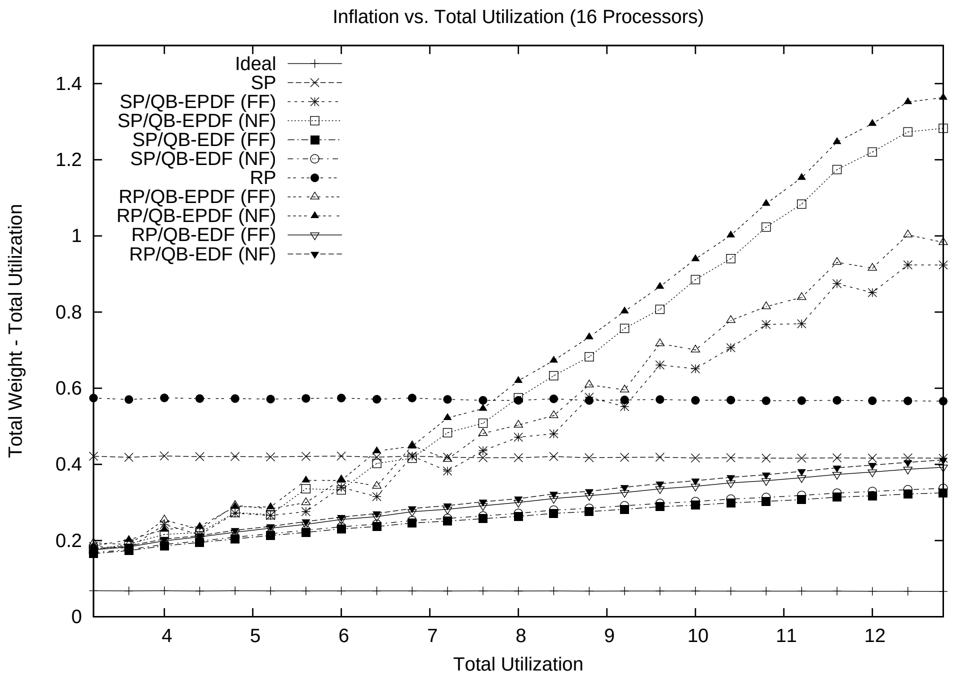


(a) Sample Means Only

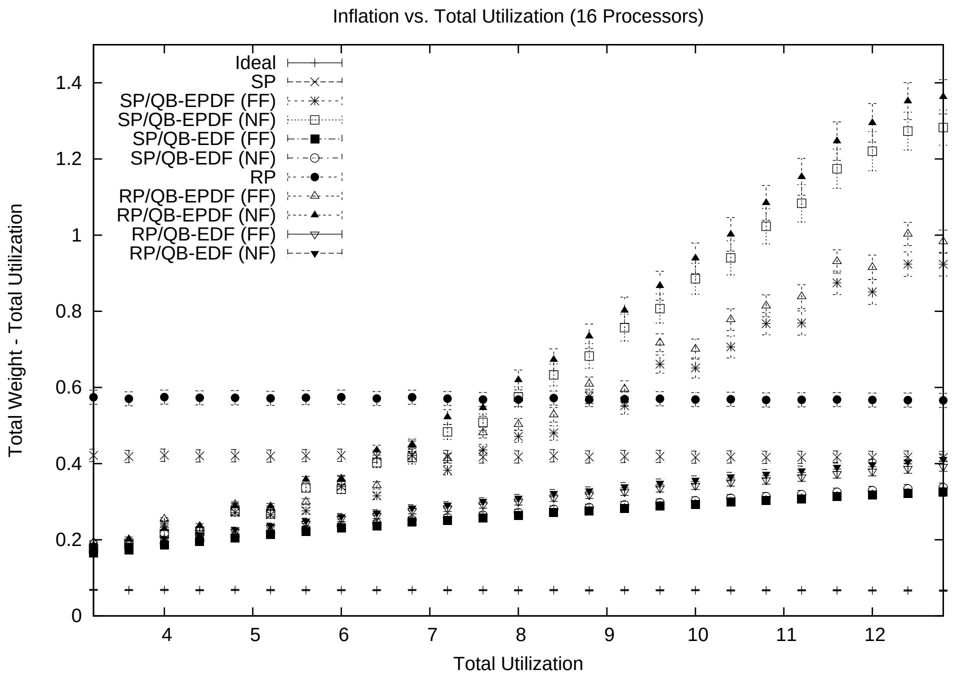


(b) 99% Confidence Interval

Figure 6.39: Plots show how inflation varies as utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

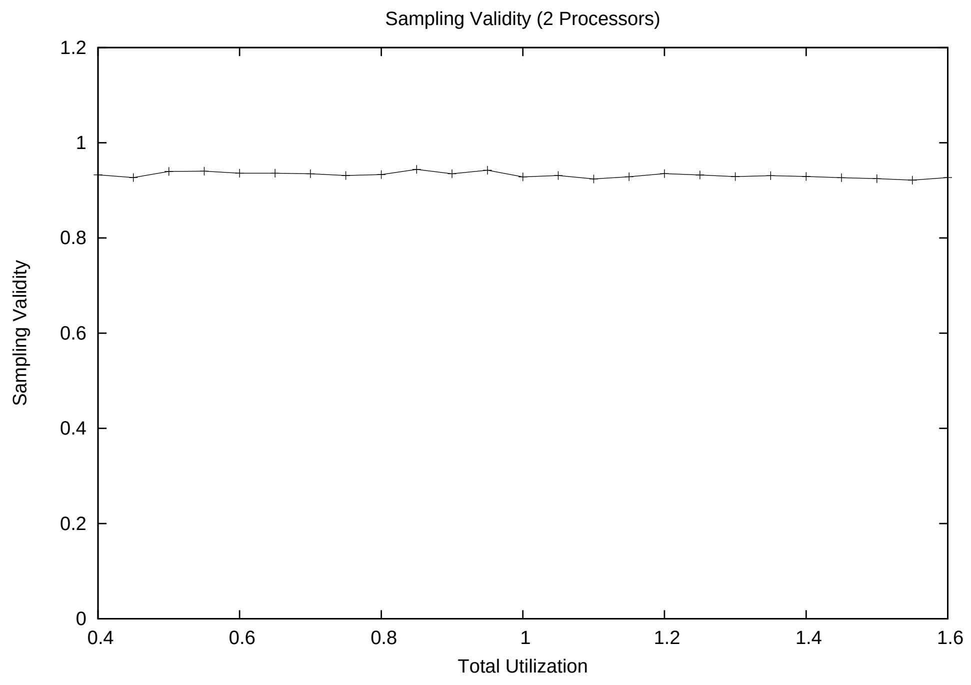


(a) Sample Means Only

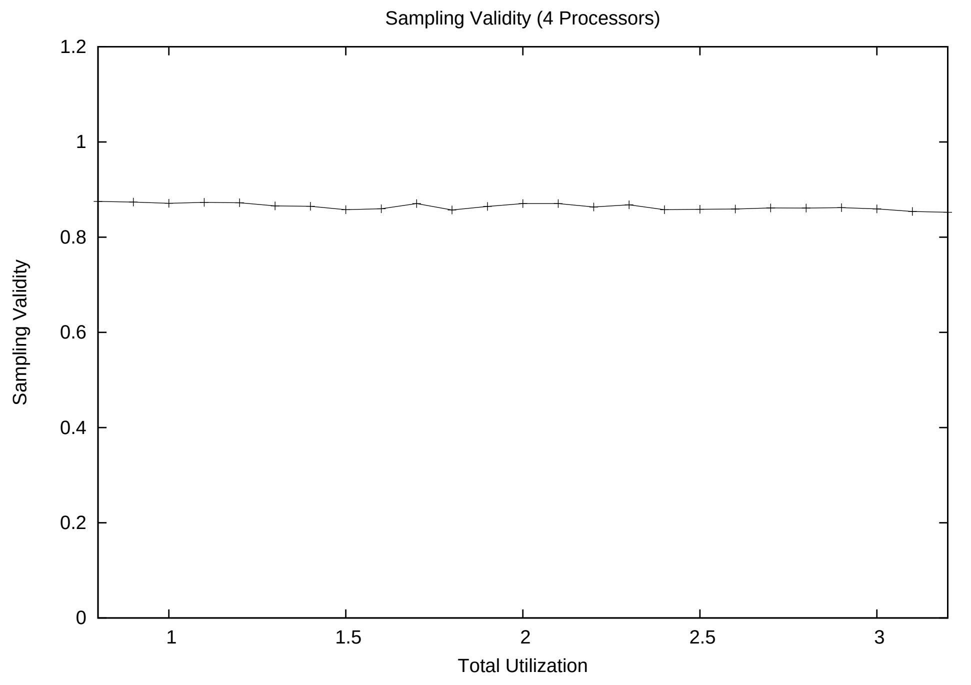


(b) 99% Confidence Interval

Figure 6.40: Plots show how inflation varies as utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

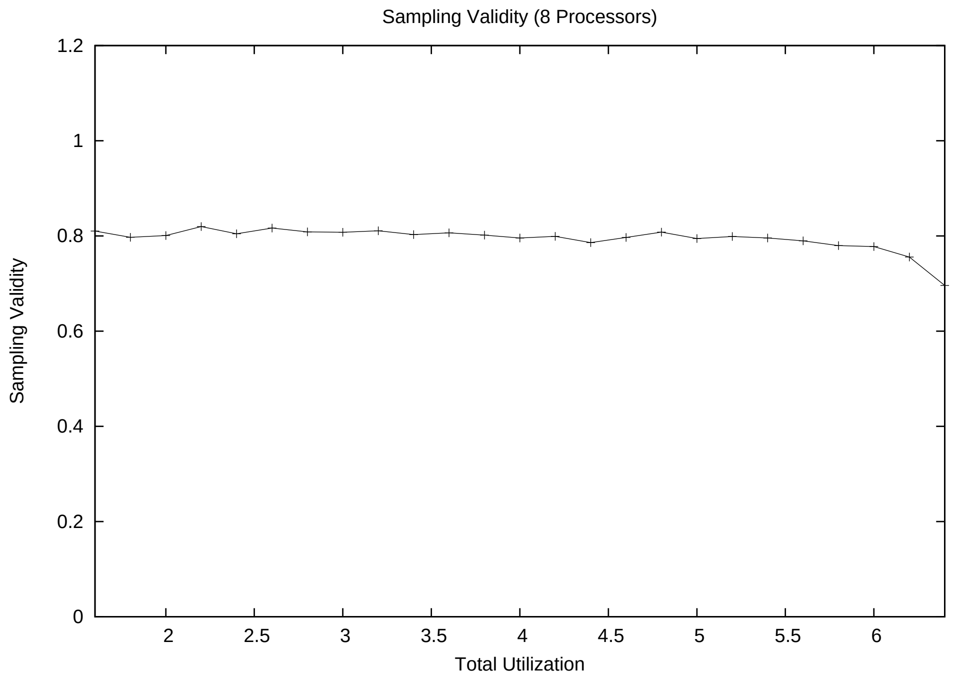


(a) $M = 2$

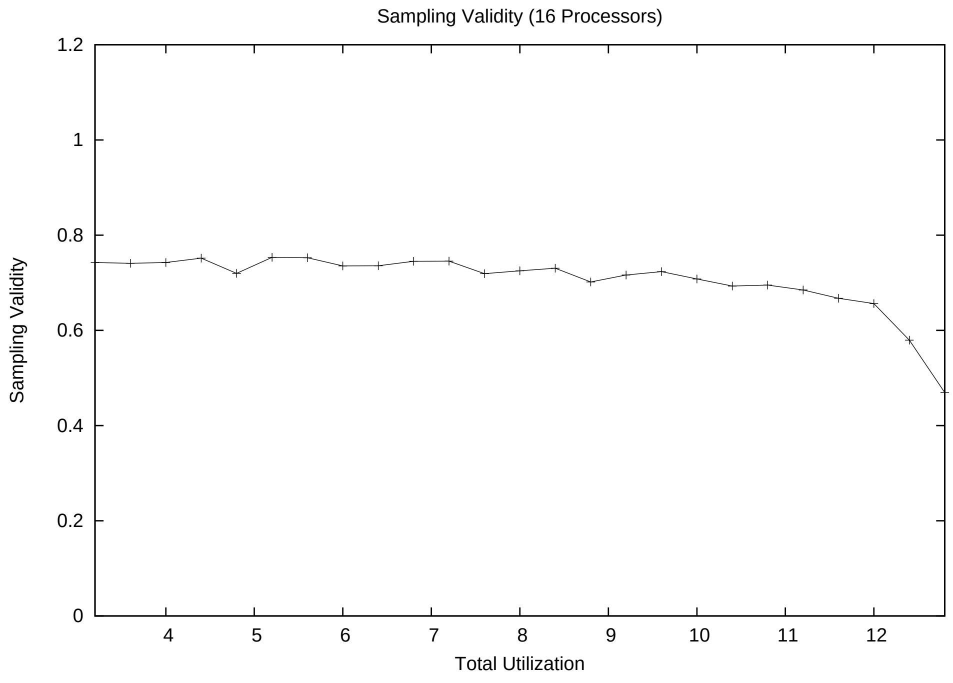


(b) $M = 4$

Figure 6.41: Plots show the sampling validity for the graphs showing inflation plotted against utilization when (R3) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



(b) $M = 16$

Figure 6.42: Plots show the sampling validity for the graphs showing inflation plotted against utilization when (R3) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

same supertask decreases with increasing utilization. As observed earlier, low component task counts produce high reweighting overhead. The observed trend is the result of this property.

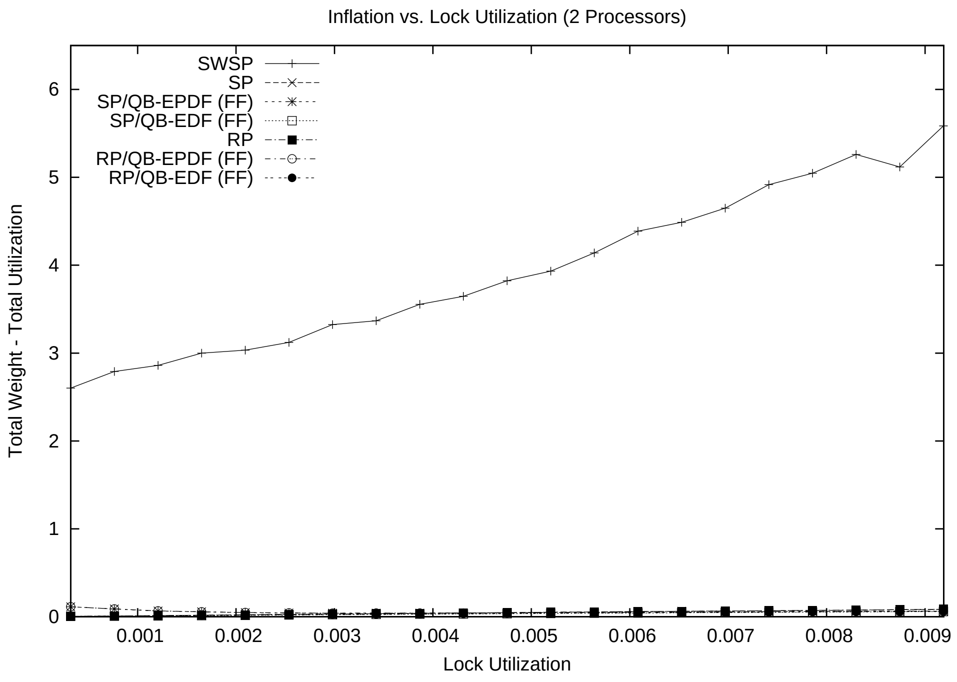
Impact of lock utilization. Figures 6.43, 6.44, 6.45, and 6.46 show how inflation varies with the lock utilization on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.47, 6.48, 6.49, and 6.50 show the remaining approaches. Plots of the sampling validity are shown in Figures 6.51 and 6.52.

Unlike the previous graphs, those based on lock utilization do *not* represent uniform sampling. Recall that the sample space was systematically sampled from with respect to parameters like the task count and lock count. However, no attempt was made to control the lock utilization. As a result, the number of samples associated with each sample mean varies from point to point.

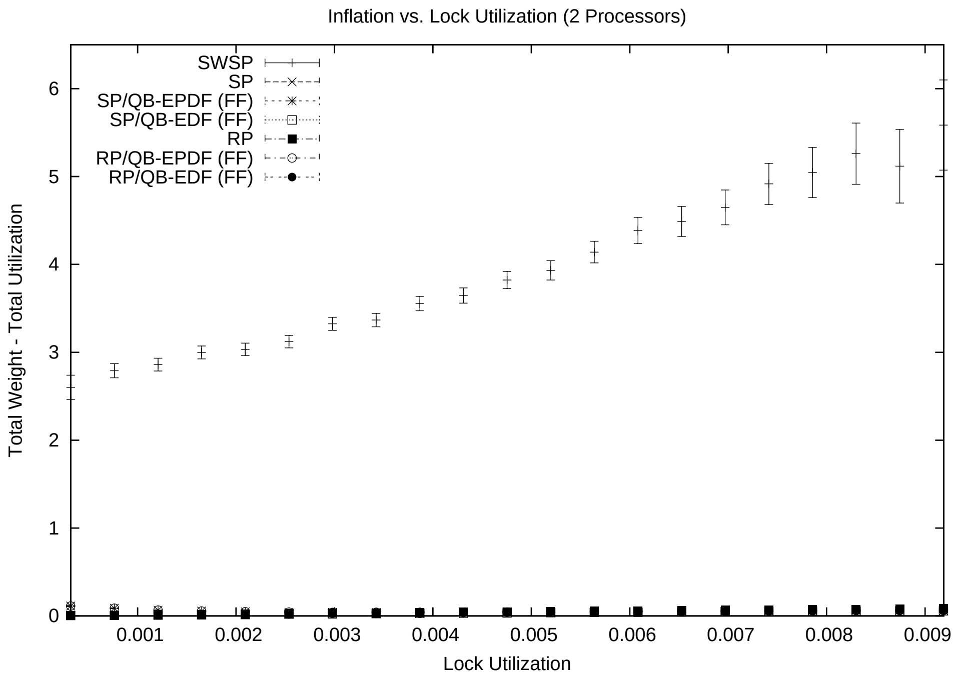
In Chapter 5, we noted a strong resemblance between the task count graphs and the weighted contention graphs. It appears that a similar relationship exists between the task count graphs and the lock utilization graphs shown here. Again, we speculate that these quantities are correlated due to random generation. A correlation likely exists in real task sets also, though the degree of the correlation may differ. (This expectation is based on the assumption that adding tasks to a task set is likely to introduce more resource sharing.) Since the explanation of trends shown in these graphs were given previously in the context of the task count graphs (see page 282), we forgo further discussion of these graphs.

6.6.2 Study 2: Performance under (R1)

In this section, we present the details and results of the second study. This study focused on measuring the relative performance of the locking approaches when (R1) holds. Since the RP

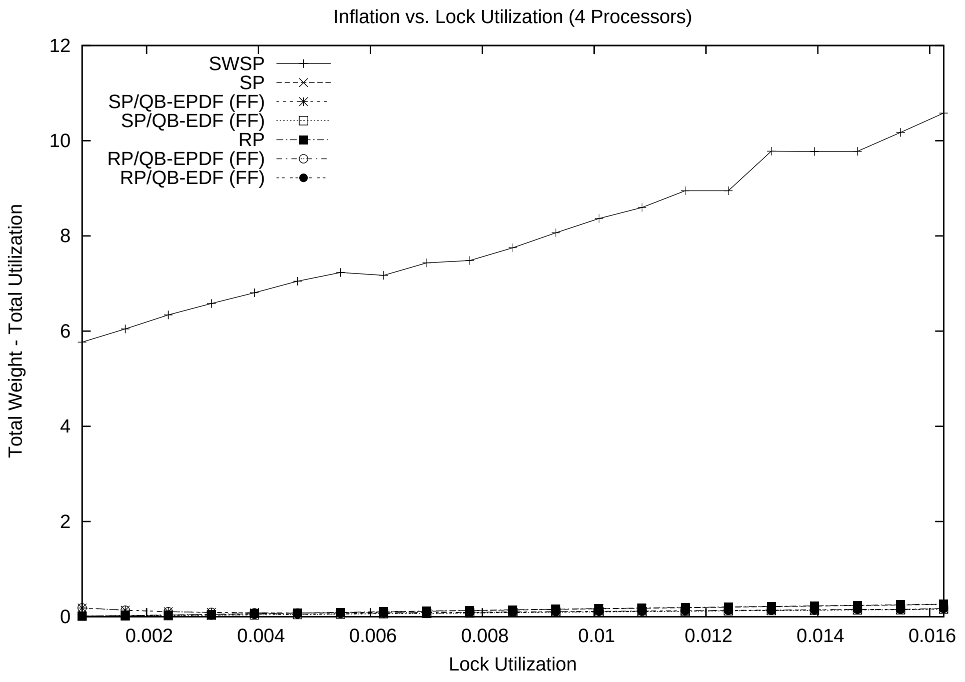


(a) Sample Means Only

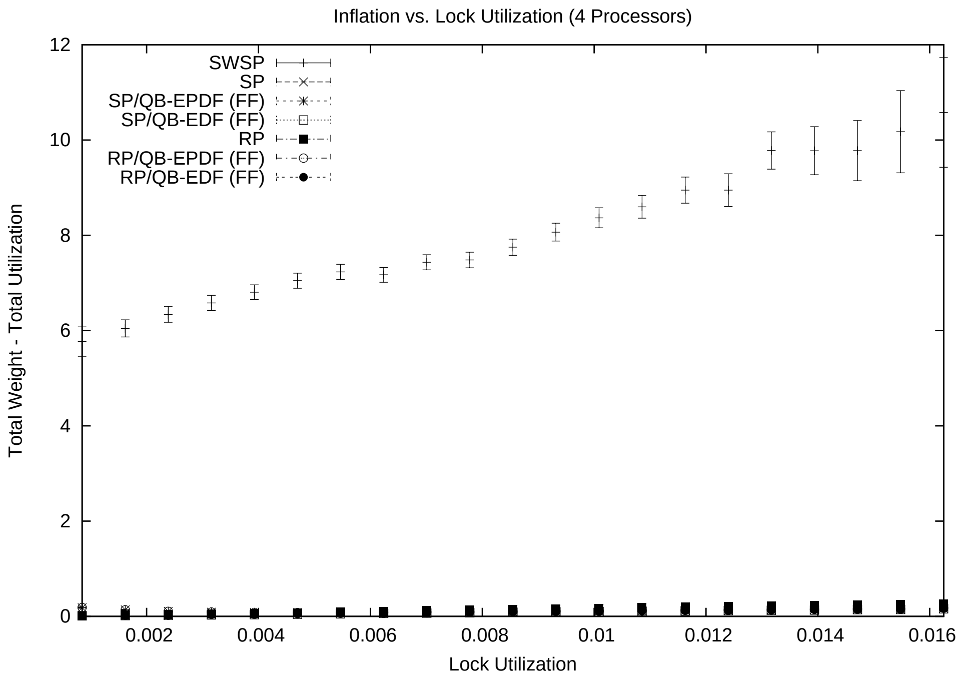


(b) 99% Confidence Interval

Figure 6.43: Plots show how inflation varies as lock utilization is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

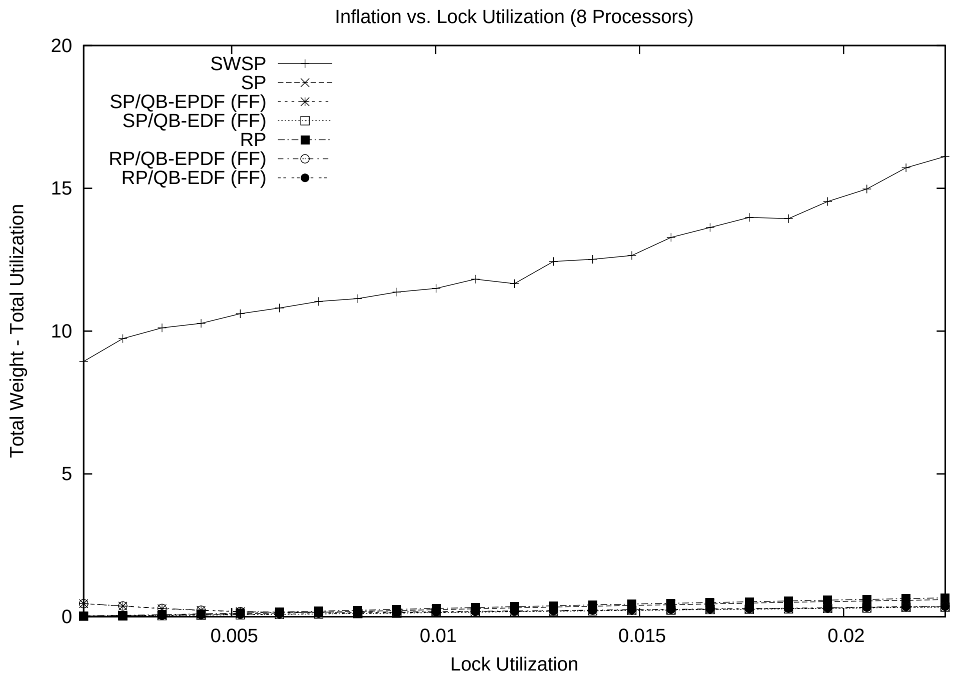


(a) Sample Means Only

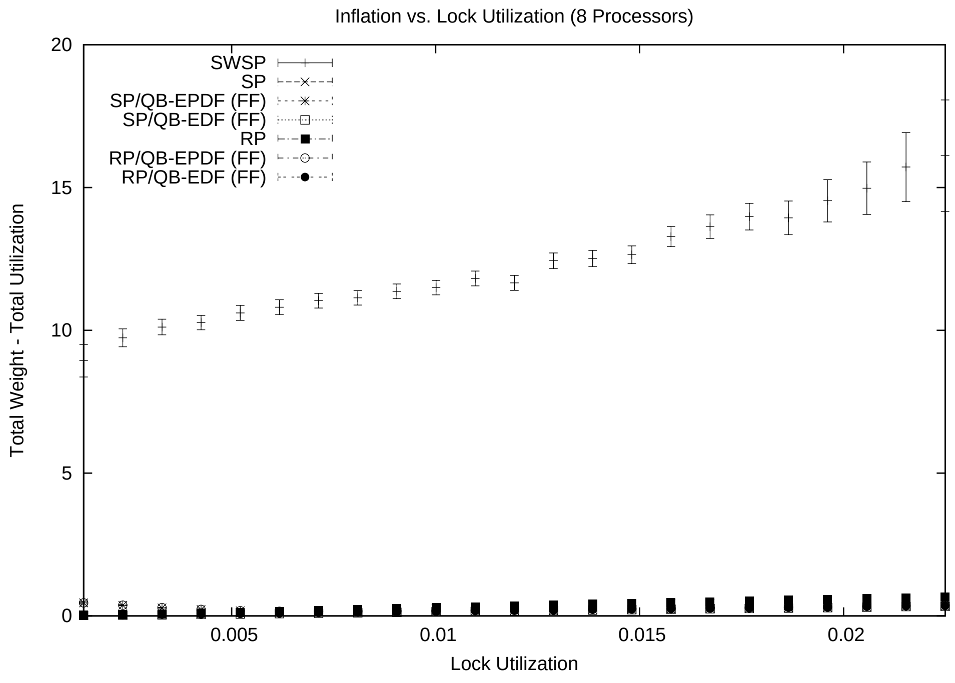


(b) 99% Confidence Interval

Figure 6.44: Plots show how inflation varies as lock utilization is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

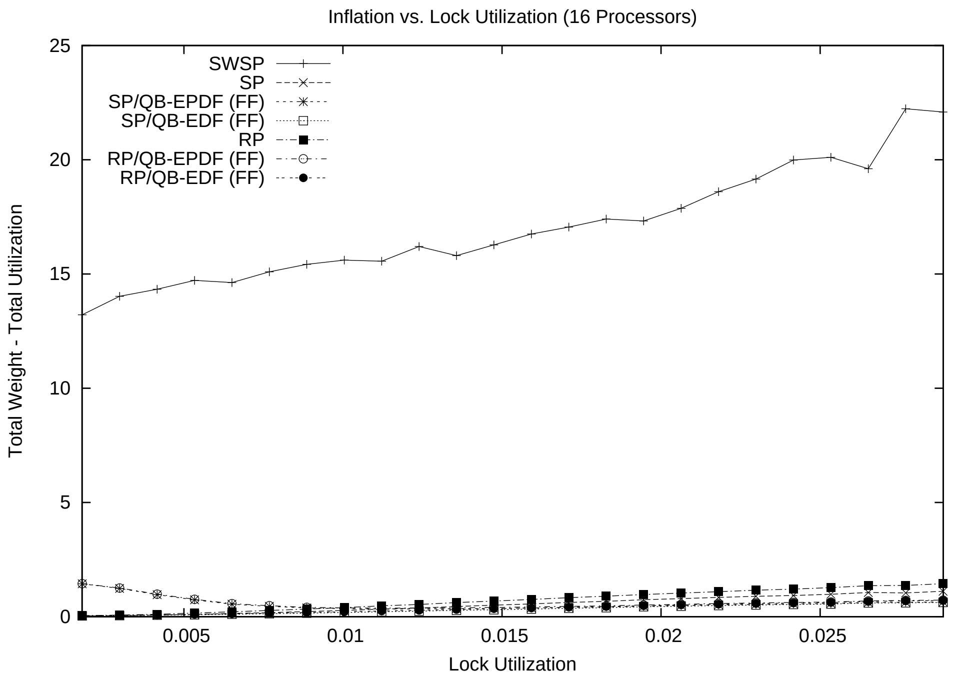


(a) Sample Means Only

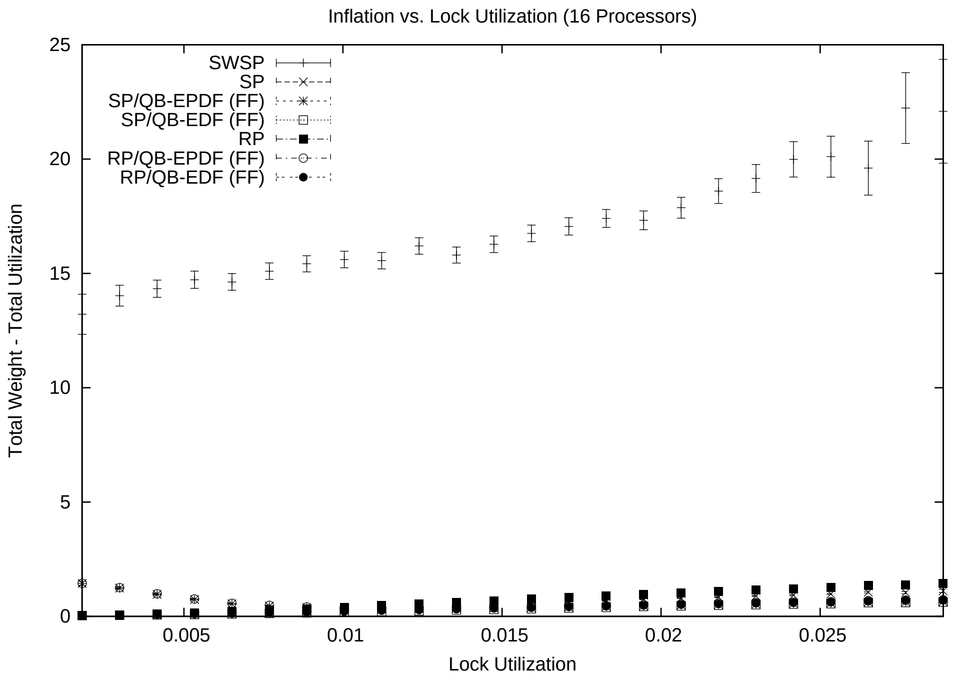


(b) 99% Confidence Interval

Figure 6.45: Plots show how inflation varies as lock utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

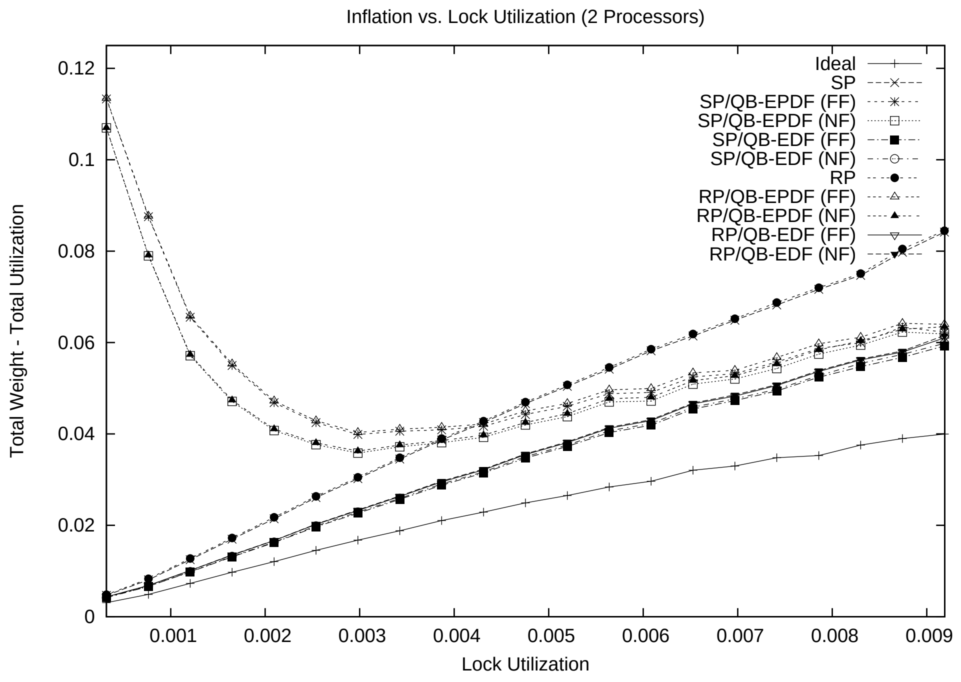


(a) Sample Means Only

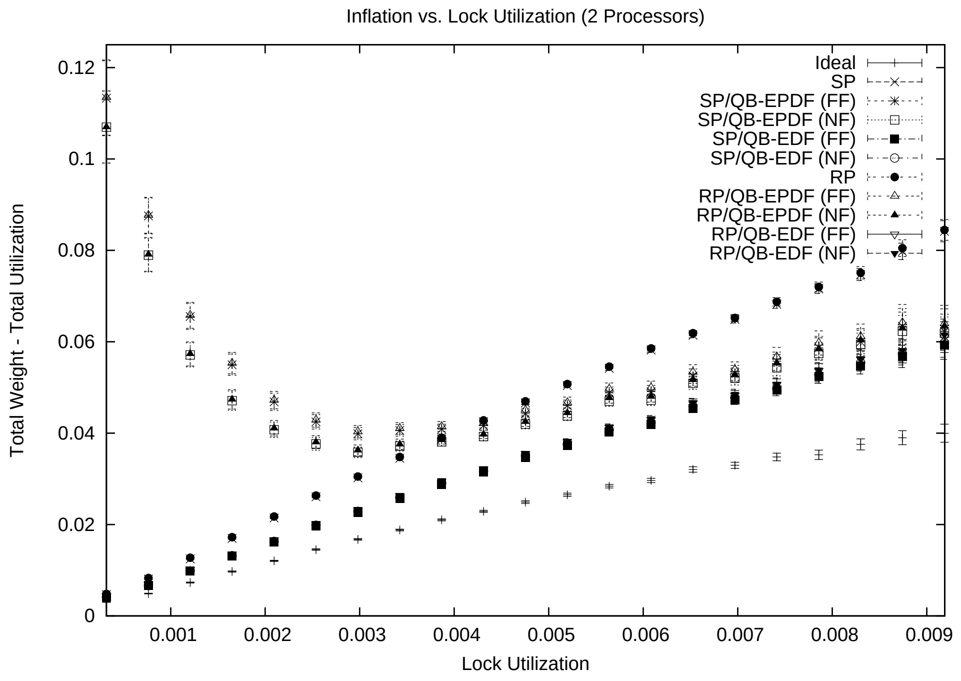


(b) 99% Confidence Interval

Figure 6.46: Plots show how inflation varies as lock utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

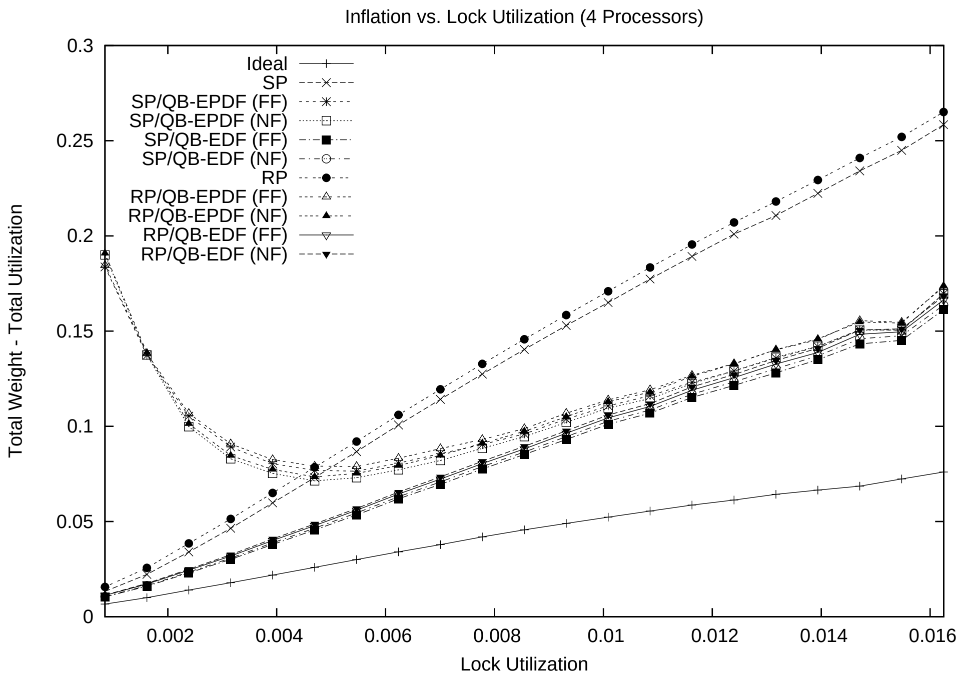


(a) Sample Means Only

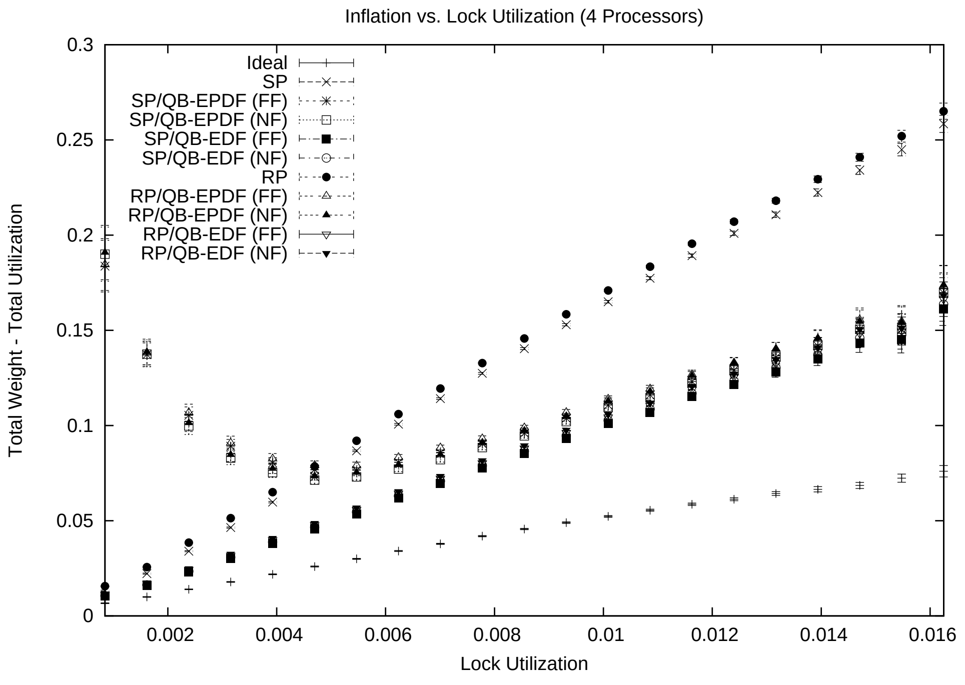


(b) 99% Confidence Interval

Figure 6.47: Plots show how inflation varies as lock utilization is increased when using locking synchronization on two processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

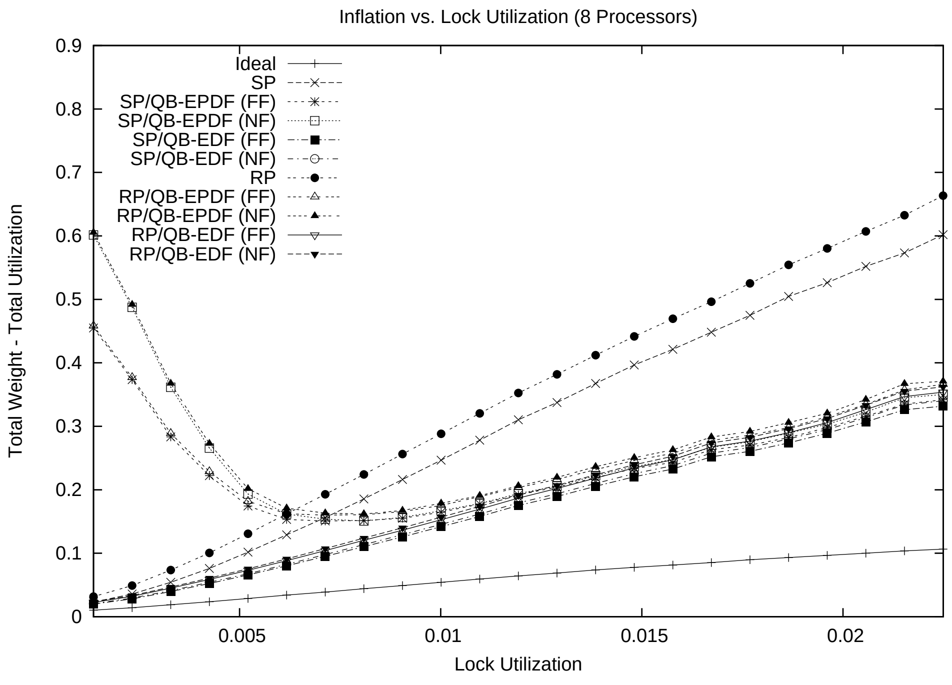


(a) Sample Means Only

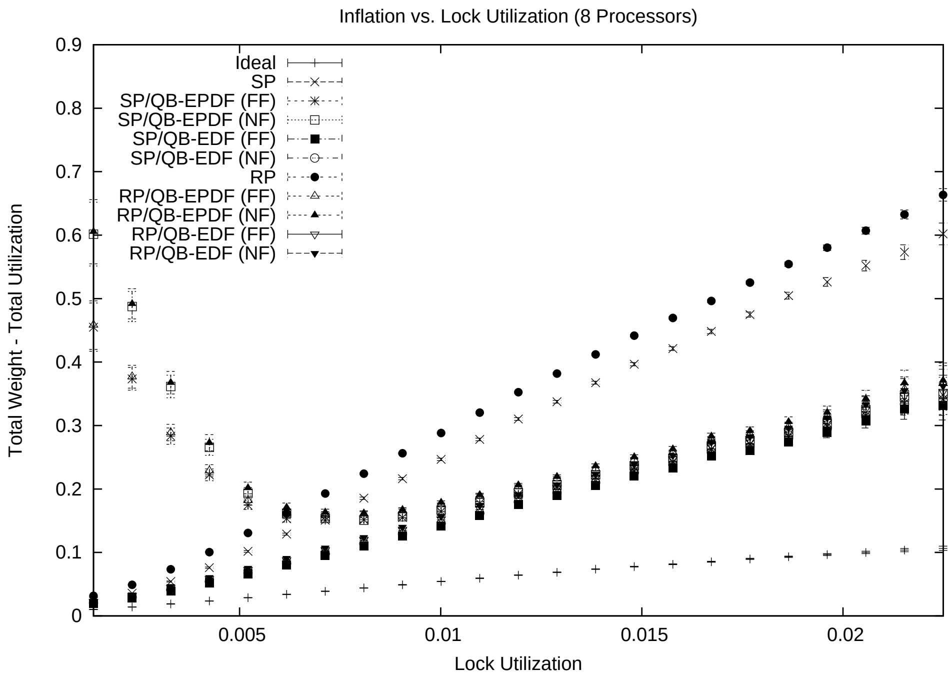


(b) 99% Confidence Interval

Figure 6.48: Plots show how inflation varies as lock utilization is increased when using locking synchronization on four processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

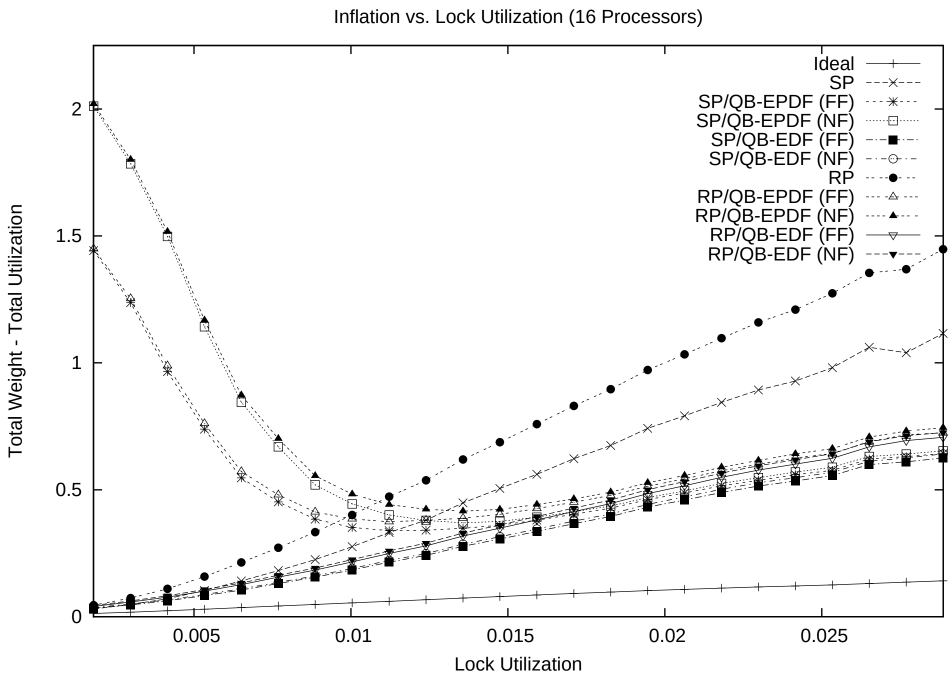


(a) Sample Means Only

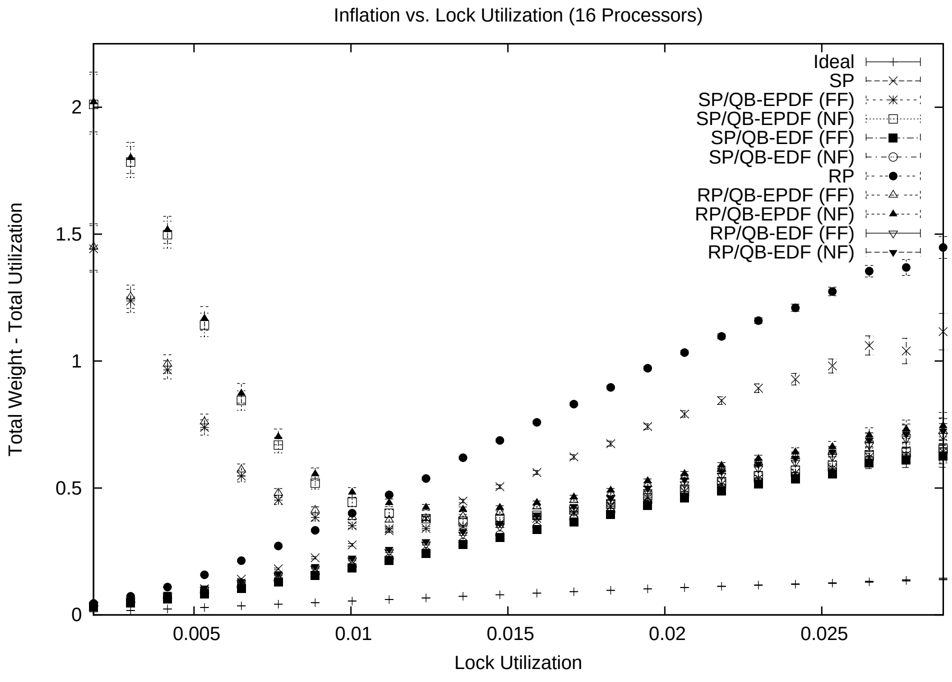


(b) 99% Confidence Interval

Figure 6.49: Plots show how inflation varies as lock utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

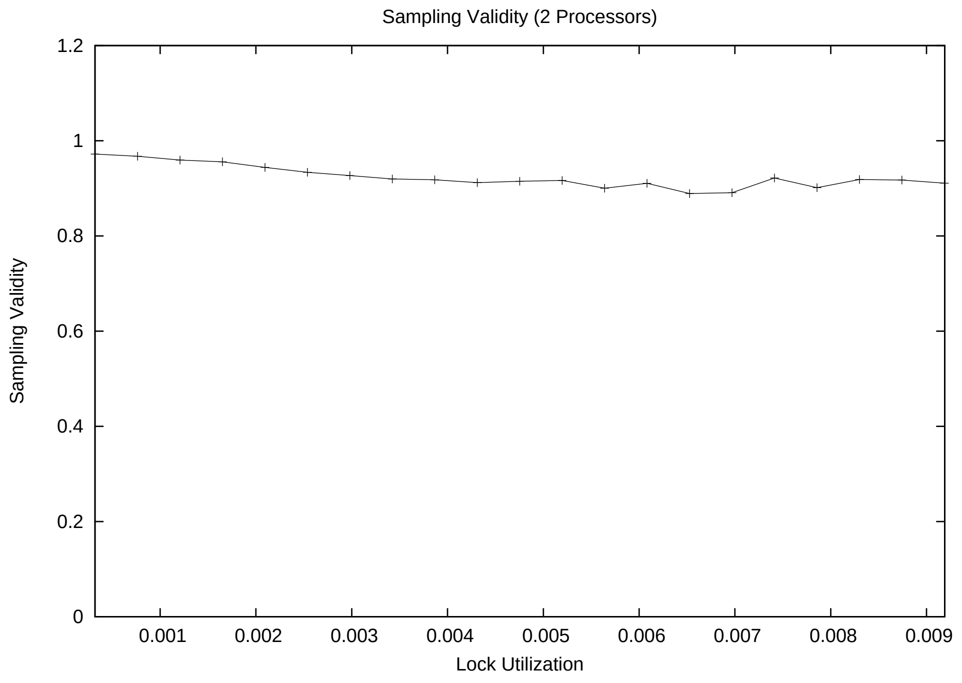


(a) Sample Means Only

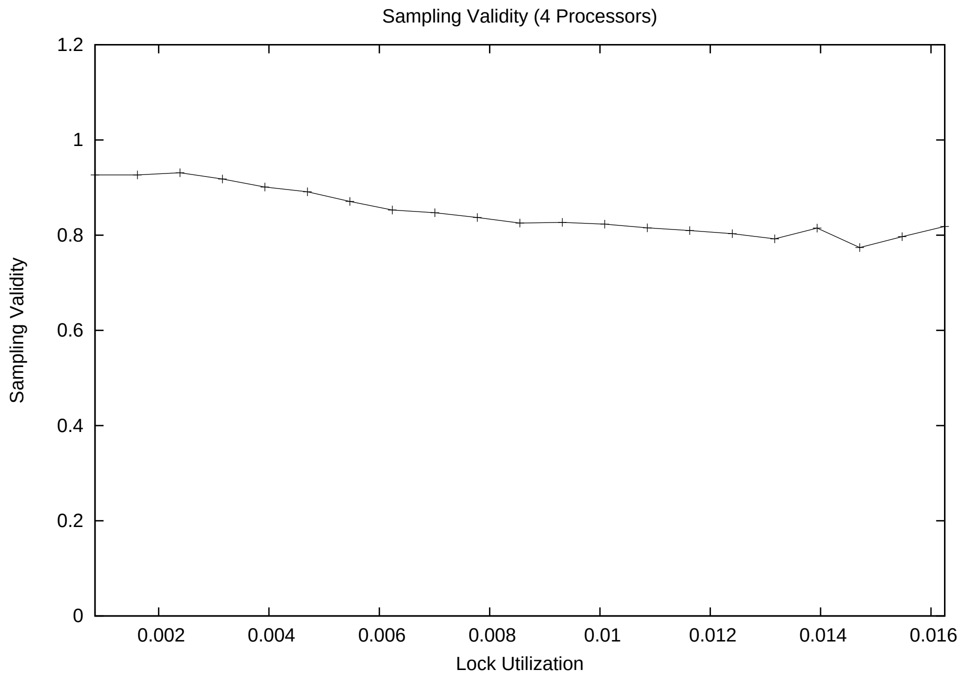


(b) 99% Confidence Interval

Figure 6.50: Plots show how inflation varies as lock utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R3). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

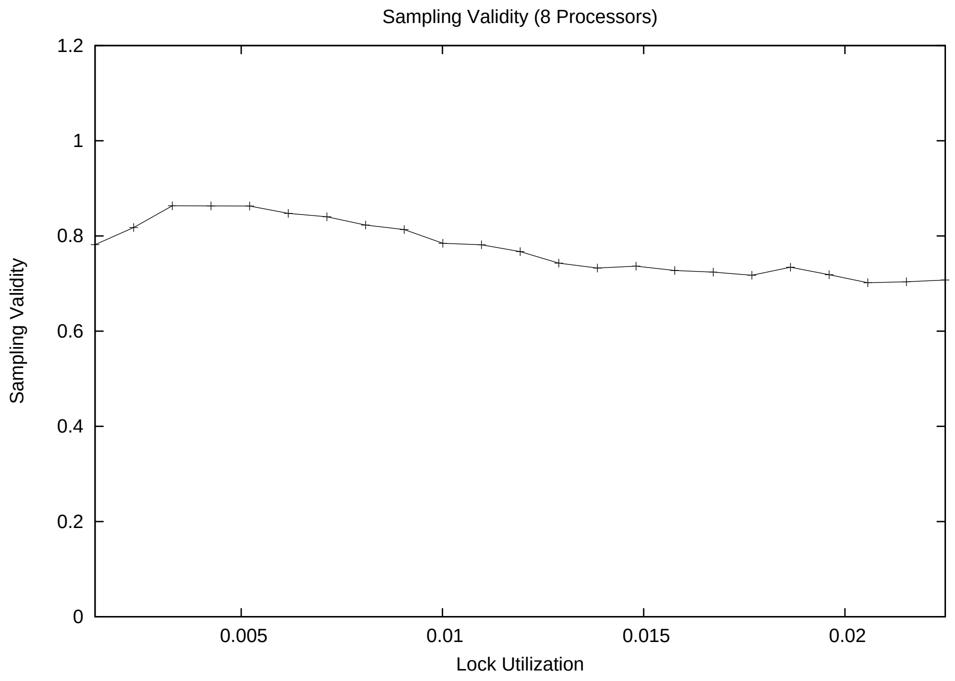


(a) $M = 2$

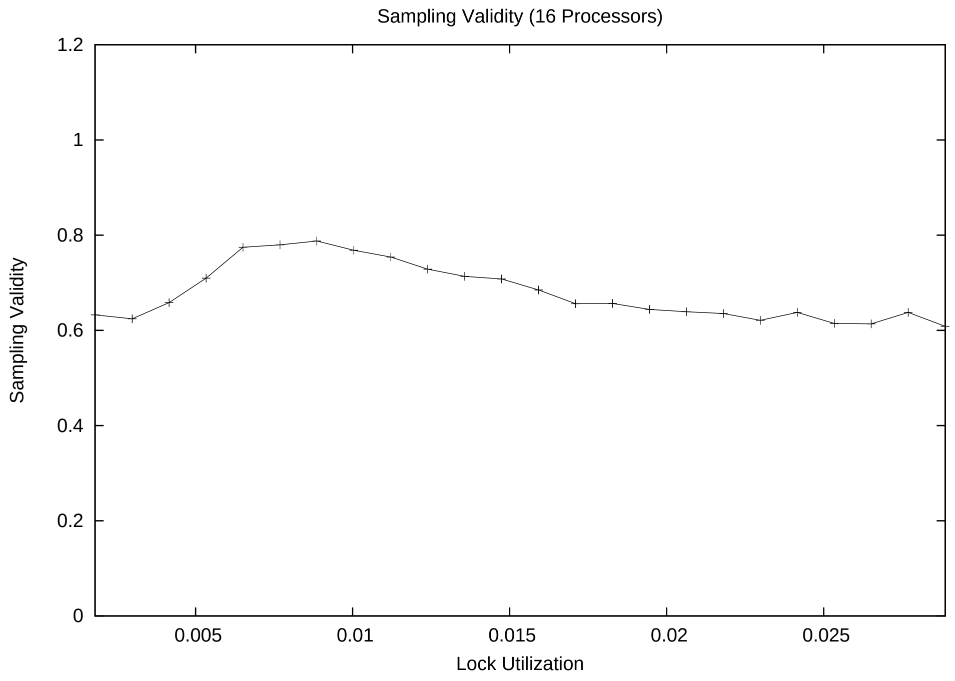


(b) $M = 4$

Figure 6.51: Plots show the sampling validity for the graphs showing inflation plotted against lock utilization when (R3) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



(b) $M = 16$

Figure 6.52: Plots show the sampling validity for the graphs showing inflation plotted against lock utilization when (R3) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

requires (R3) to hold, this approach was not considered in this study.

Sample space. For this study, the sample space was defined as follows:

- M is in the range $2, \ldots, 16$;
- $|\tau|$ is in the range $5 \cdot \log_2 M, \ldots, 20 \cdot \log_2 M$;
- $\tau.u$ is in the range $0.2 \cdot M, \ldots, 0.6 \cdot M$;
- $|\Gamma|$ is in the range $\log_2 M, \ldots, 10 \cdot \log_2 M$;
- $T.p$ is in the range $50, \ldots, 2000$;
- C equals 0.25, *i.e.*, 25% of a quantum.

Recall that C is the maximum allowed critical section length. The above C value was selected arbitrarily. (The next study focuses on how performance scales with C.) Because locks that satisfy (R1) instead of (R3) tends to produce more locking overhead, the sample space had to be more restricted in this study than in the previous one. Specifically, the upper bounds on $|\tau|$ and $\tau.u$ were decreased. This was necessary to produce reasonably high sampling validity.

Sampling. Samples were collected as follows:

- M was set to each value in the set $\{\ 2^x \mid 1 \leq x \leq 4\ \}$;
- $|\tau|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 5 \leq x \leq 20\ \}$;
- $\tau.u$ was set to each value in the set $\{\ 0.025x \cdot M \mid 8 \leq x \leq 24\ \}$;
- $|\Gamma|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 1 \leq x \leq 10\ \}$.

$M = 2$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.011659	0.006782	0.00010594	*N/A*
SWSP	3.389903	2.257611	0.03526242	0.000%
SP	0.313764	0.259619	0.00405508	0.011%
SP/Ideal QB (FF)	0.044573	0.041110	0.00064211	*N/A*
SP/Ideal QB (NF)	0.046804	0.046594	0.00072777	*N/A*
SP/QB-EPDF (FF)	0.074769	0.076085	0.00118841	*N/A*
SP/QB-EPDF (NF)	0.072638	0.073335	0.00114544	*N/A*
SP/QB-EDF (FF)	0.044931	0.041219	0.00064381	10.636%
SP/QB-EDF (NF)	0.047097	0.046648	0.00072861	0.849%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	88.504%

Figure 6.53: Summary of the $M = 2$ results of the second locking experiment in which all locks respect (R1).

For each combination of the above parameter assignments, ten valid task sets were generated and evaluated. The sampling validity criteria is identical to that used by the previous study except for the fact that locks were only required to satisfy (R1) in this study.

Measurement. This study is based on the same measurements as the previous study, except for the omission of RP measurements. Since the SWSP's behavior does not depend on (R1) or (R3), its trends should resemble those already observed in the previous study. For this reason, we primarily focus on the performance of the SP and its variants. (The SWSP measurement was taken to provide a reference point that links the (R3) and (R1) studies.)

Overall Performance. The tables shown in Figures 6.53–6.56 summarize the overall results of this study. Again, halflength and standard deviations measurements appear unremarkable. Hence, we ignore them in the discussion that follows.

In the $M = 2$ case, results resemble those given in the (R3) experiment in all aspects except the SP measurement. The inflation produced by the SP is an order of magnitude higher than in the previous study. The fact that changing C impacted the SWSP and the

$M = 4$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.022880	0.011000	0.00017181	*N/A*
SWSP	7.429091	4.458942	0.06964580	0.423%
SP	0.796303	0.609360	0.00951782	0.000%
SP/Ideal QB (FF)	0.395274	0.413159	0.00645327	*N/A*
SP/Ideal QB (NF)	0.412658	0.426754	0.00666562	*N/A*
SP/QB-EPDF (FF)	0.428270	0.409922	0.00640272	*N/A*
SP/QB-EPDF (NF)	0.439943	0.423876	0.00662067	*N/A*
SP/QB-EDF (FF)	0.395724	0.413243	0.00645459	65.140%
SP/QB-EDF (NF)	0.413041	0.426811	0.00666652	7.860%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	26.577%

Figure 6.54: Summary of the $M = 4$ results of the second locking experiment in which all locks respect (R1).

$M = 8$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.034526	0.015297	0.00023893	*N/A*
SWSP	11.798888	6.601403	0.10310966	2.283%
SP	1.381273	1.121643	0.01751933	0.000%
SP/Ideal QB (FF)	1.078198	0.980108	0.01530865	*N/A*
SP/Ideal QB (NF)	1.099338	0.992138	0.01549655	*N/A*
SP/QB-EPDF (FF)	1.151648	0.962305	0.01503058	*N/A*
SP/QB-EPDF (NF)	1.188704	0.992338	0.01549967	*N/A*
SP/QB-EDF (FF)	1.078965	0.980205	0.01531016	81.853%
SP/QB-EDF (NF)	1.100113	0.992318	0.01549937	13.460%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	2.404%

Figure 6.55: Summary of the $M = 8$ results of the second locking experiment in which all locks respect (R1).

supertasking approaches much less has two implications. First, the most significant source of inflation under the SWSP is apparently not a function of C. Since the only other source of inflation is the server weights, it follows that these weights are the primary source of inflation under the SWSP (at least when $M = 2$). Second, increasing C by an order of magnitude should significantly impact the performance of each approach. The fact that this change does not appear to impact the SWSP significantly (compare Figures 6.9 and 6.53) suggests that the impact of active blocking is being minimized by the use of supertasks, which can only be

$M = 16$	**Mean**	**Standard Deviation**	**Halflength**	**Best**
Ideal	0.045768	0.019647	0.00030687	*N/A*
SWSP	16.471846	8.689446	0.13572355	3.426%
SP	2.006823	1.767475	0.02760682	0.246%
SP/Ideal QB (FF)	1.775394	1.653127	0.02582078	*N/A*
SP/Ideal QB (NF)	1.796558	1.665604	0.02601566	*N/A*
SP/QB-EPDF (FF)	2.034739	1.607010	0.02510046	*N/A*
SP/QB-EPDF (NF)	2.226956	1.696965	0.02650549	*N/A*
SP/QB-EDF (FF)	1.776957	1.653090	0.02582019	81.493%
SP/QB-EDF (NF)	1.798841	1.665892	0.02602015	14.695%
SP/QB-EDF (Both)	*N/A*	*N/A*	*N/A*	0.140%

Figure 6.56: Summary of the $M = 16$ results of the second locking experiment in which all locks respect (R1).

the case if sharing between supertasks is rare.

As M increases, supertasking becomes increasingly less effective. Indeed, QB-EPDF supertasking performs worse than SP alone in the $M = 16$ case. In addition, the performance of the SP also degrades. This trend is indicated not only by the inflation measurements, but also by the Best percentages. In the $M = 4$, 8, and 16 cases, the fraction of task sets favoring the use of the SWSP over all variants of the SP was 0.423%, 2.283%, and 3.426%, respectively. Although the majority of samples continue to favor the use of the SP (with supertasks), this increasing minority of samples clearly indicates that zone-based protocols are not always a better choice than server-based protocols. (See Appendix B for a more detailed summary of which approach was favored by the tested samples.)

In the remainder of this section, we present graphs to show how inflation scales with the task set parameters. In all graphs, the “Ideal QB (FF),” “Ideal QB (NF),” “SP/QB-EDF (FF),” and ‘SP/QB-EDF (NF)” curves are virtually co-linear.

Impact of the task count. Figures 6.57, 6.58, 6.59, and 6.60 show how inflation varies with the task count on two, four, eight, and sixteen processors, respectively, at the scale of the

SWSP's inflation. Figures 6.61, 6.62, 6.63, and 6.64 show the remaining approaches. Plots of the sampling validity can be seen in Figures 6.65 and 6.66.

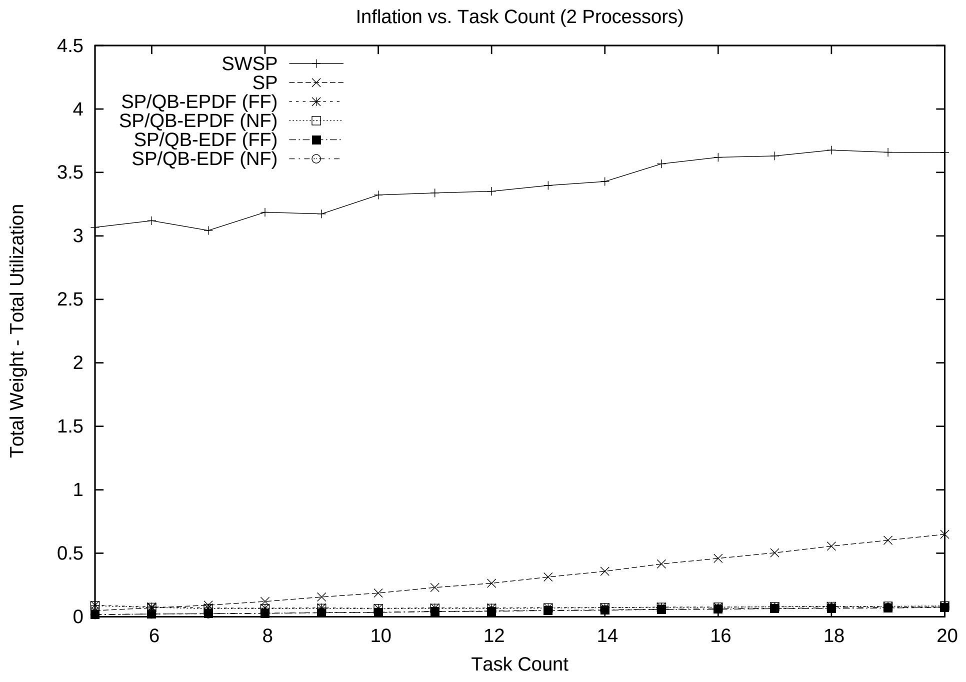
Observation 6.10 *The benefit of supertasking appears to diminish quickly as M increases, but the reliability of this observation is uncertain.*

Explanation: Figures 6.61–6.64 clearly show that using supertasks with the SP provides less improvement for larger values of M. However, the decline of sampling validity as M increases, shown in Figures 6.65 and 6.66, could be increasingly skewing the means as M increases. Since the evidence does not clearly indicate which of these possible explanations is correct, further study is needed.

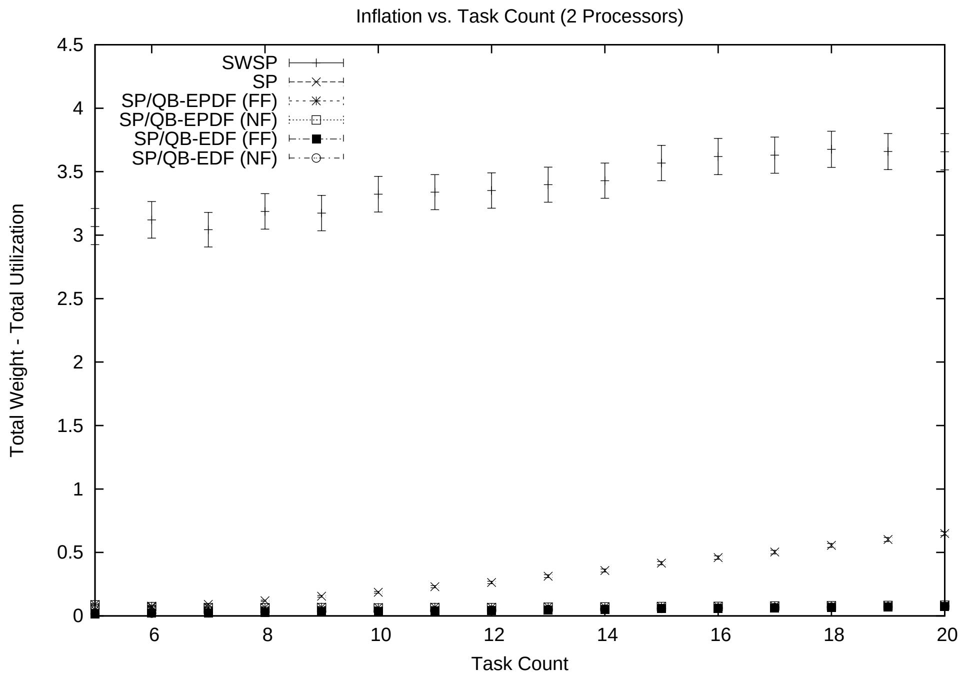
Impact of the lock count. Figures 6.67, 6.68, 6.69, and 6.70 show how inflation varies with the lock count on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.71, 6.72, 6.73, and 6.74 show the remaining approaches. Plots of the sampling validity are shown in Figures 6.75 and 6.76.

These graphs simply repeat trends that have already been observed. Specifically, the benefit of supertasking diminishes as M increases (see Observation 6.10). In addition, these graphs clearly show that coarse-grained locking is more costly than fine-grained locking (see Observation 6.5).

Impact of utilization. Figures 6.77, 6.78, 6.79, and 6.80 show how inflation varies with the total utilization on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.81, 6.82, 6.83, and 6.84 show the remaining approaches. Plots of the sampling validity are shown in Figures 6.85 and 6.86.

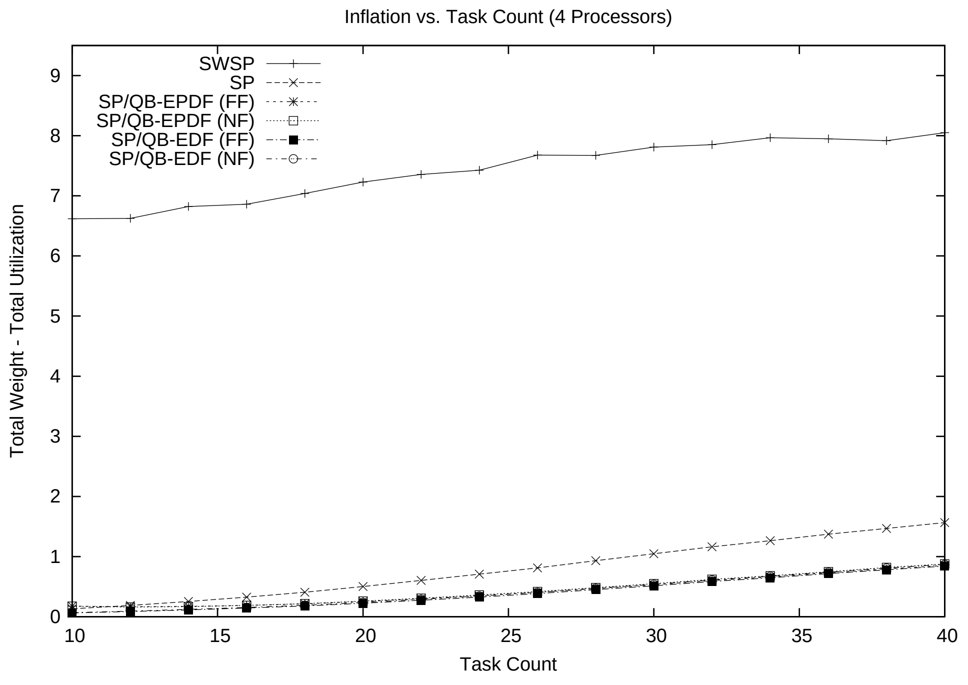


(a) Sample Means Only

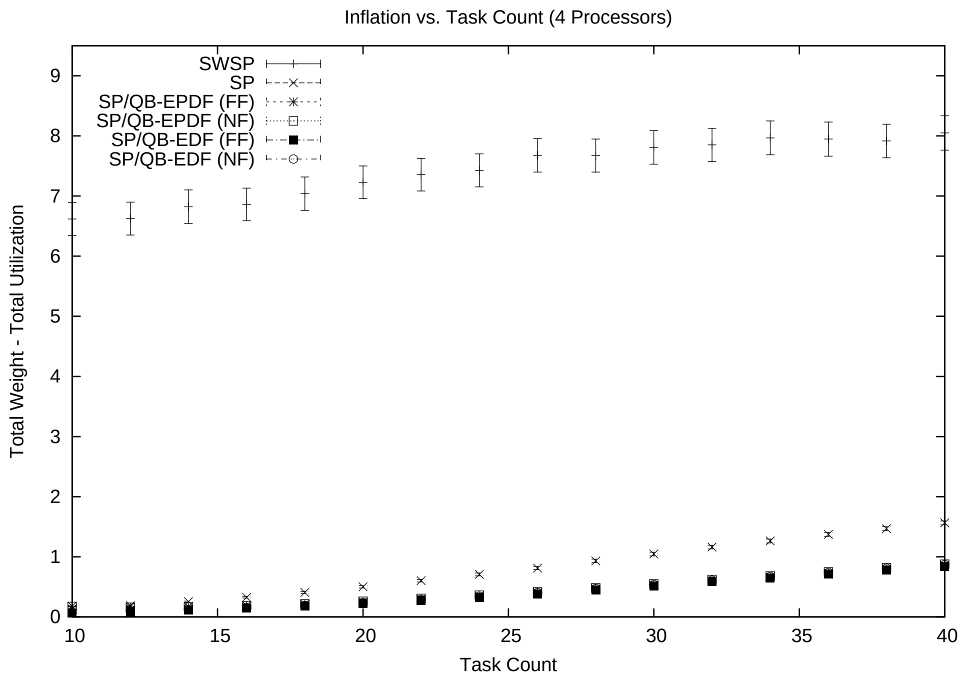


(b) 99% Confidence Interval

Figure 6.57: Plots show how inflation varies as the task count is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

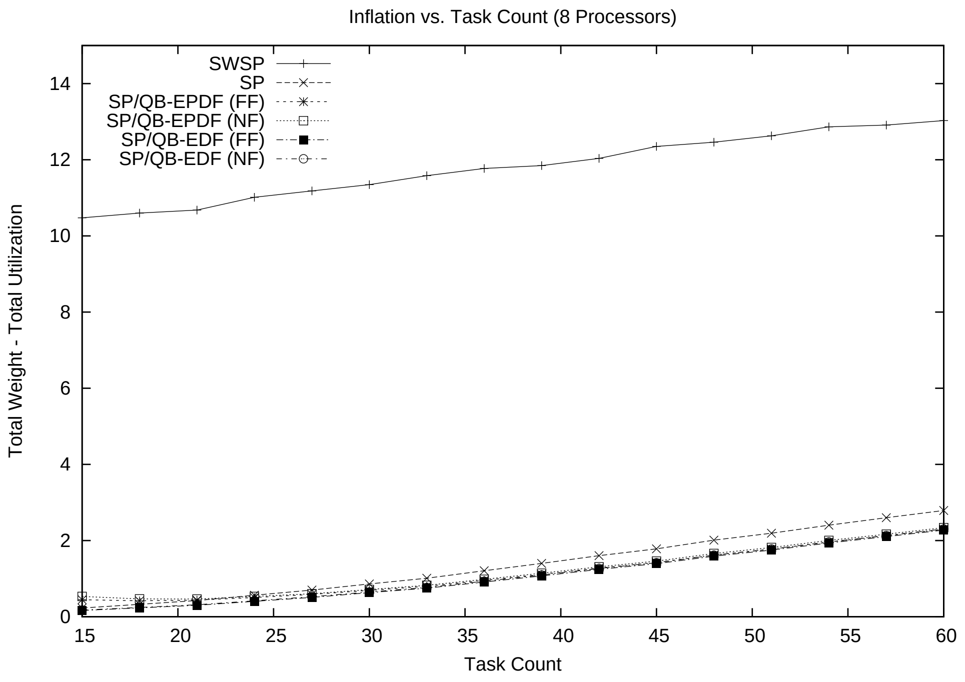


(a) Sample Means Only

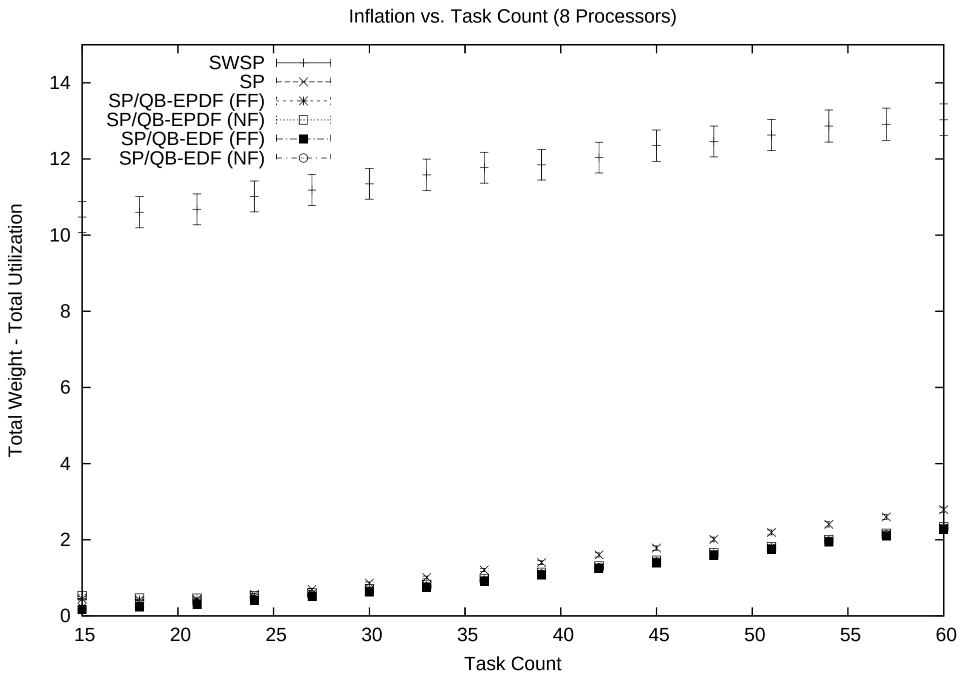


(b) 99% Confidence Interval

Figure 6.58: Plots show how inflation varies as the task count is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

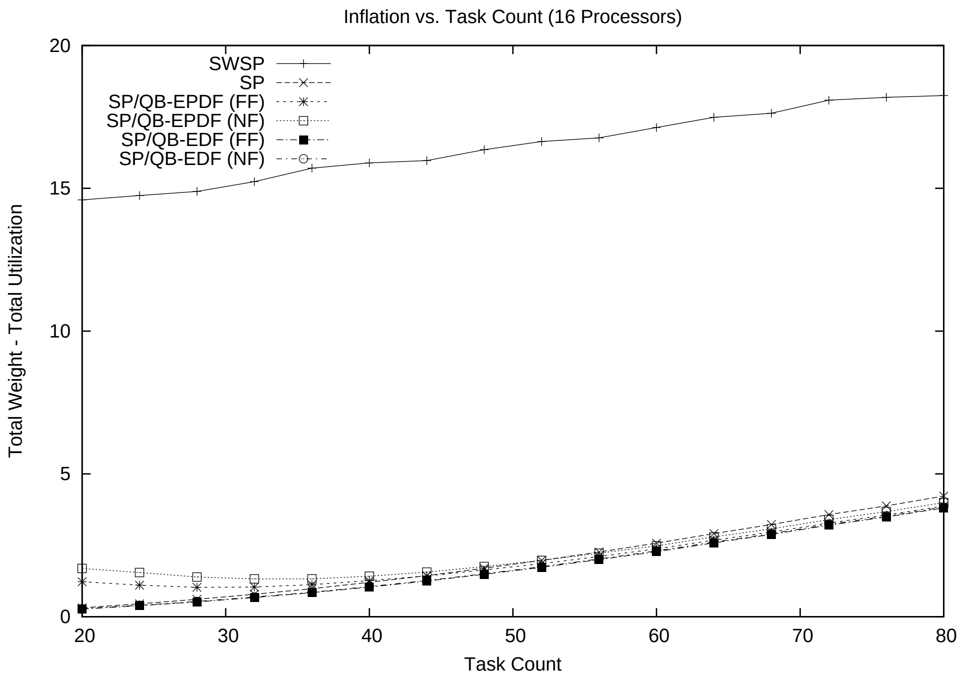


(a) Sample Means Only

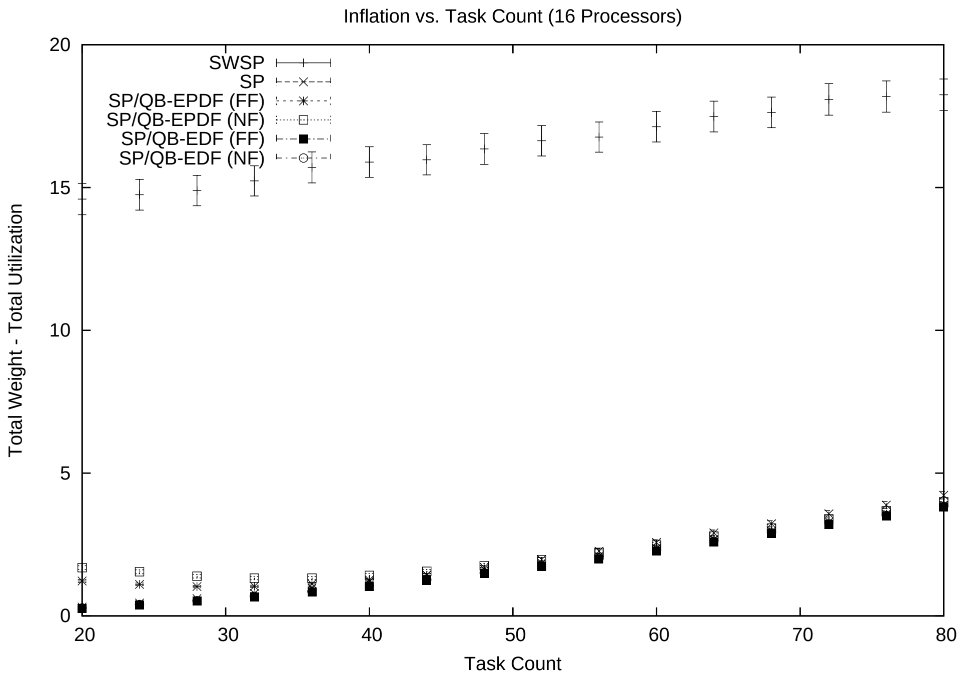


(b) 99% Confidence Interval

Figure 6.59: Plots show how inflation varies as the task count is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

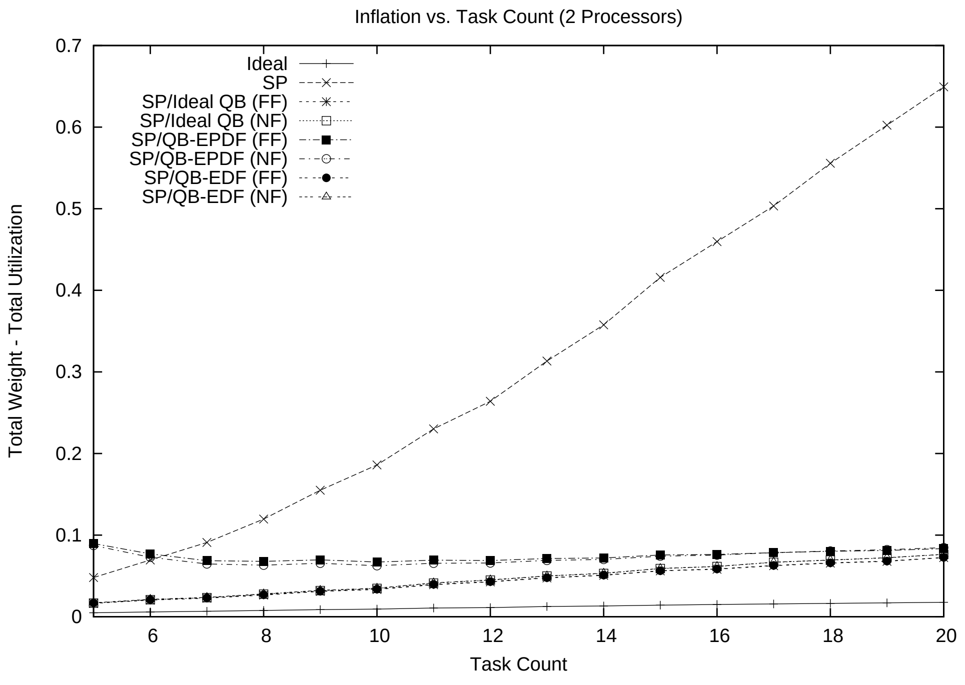


(a) Sample Means Only

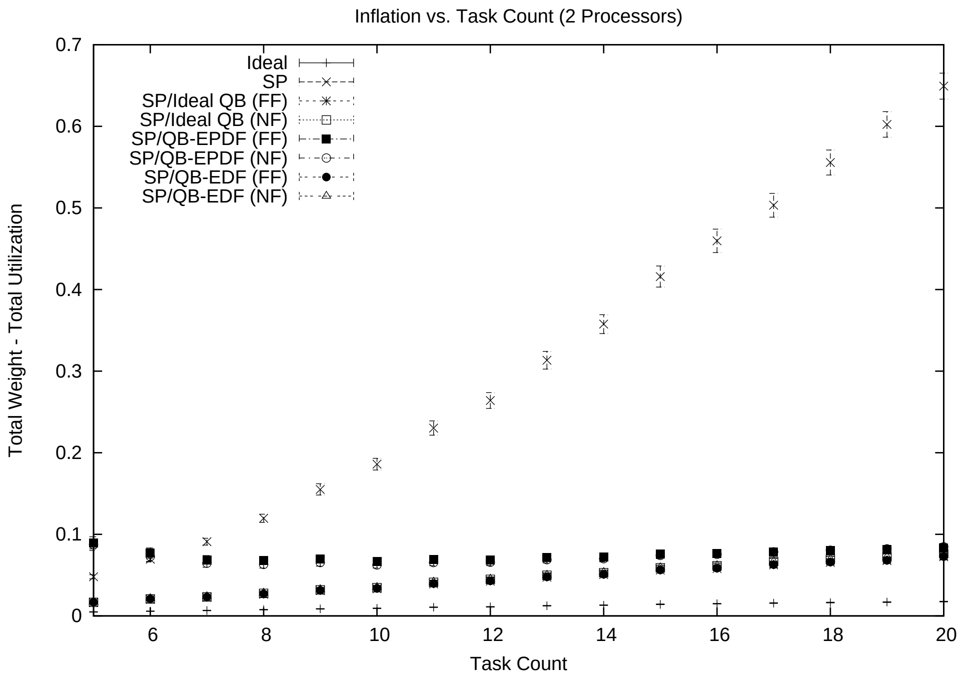


(b) 99% Confidence Interval

Figure 6.60: Plots show how inflation varies as the task count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

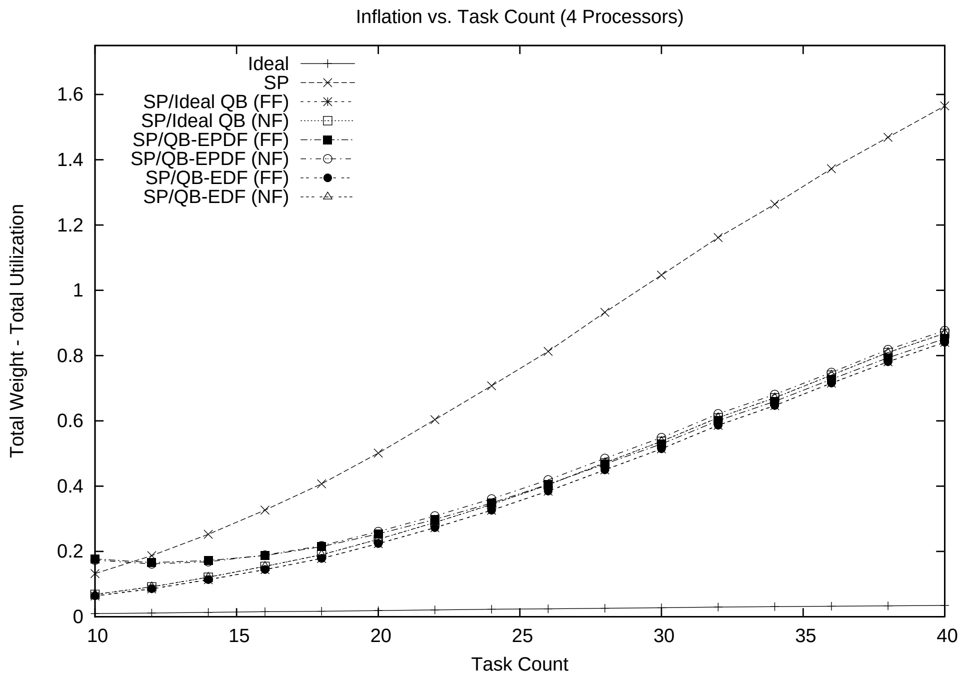


(a) Sample Means Only

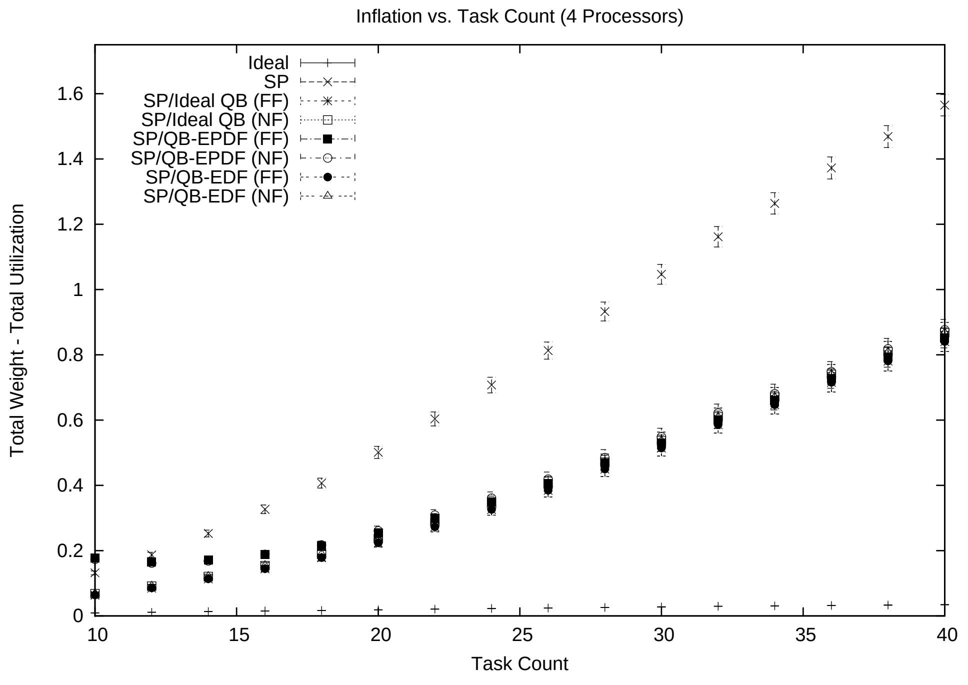


(b) 99% Confidence Interval

Figure 6.61: Plots show how inflation varies as the task count is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

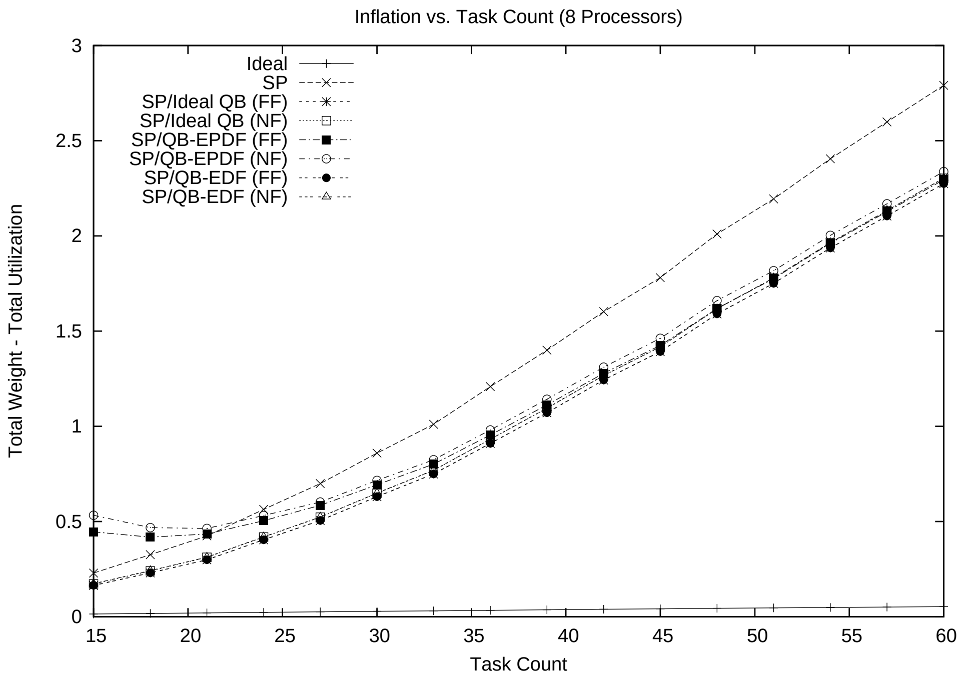


(a) Sample Means Only

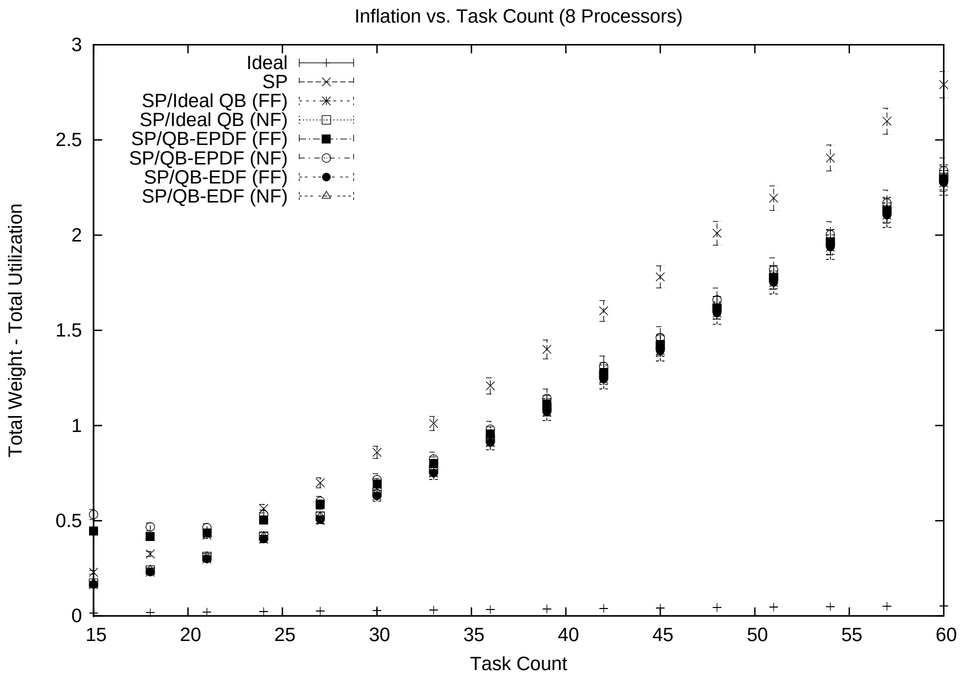


(b) 99% Confidence Interval

Figure 6.62: Plots show how inflation varies as the task count is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

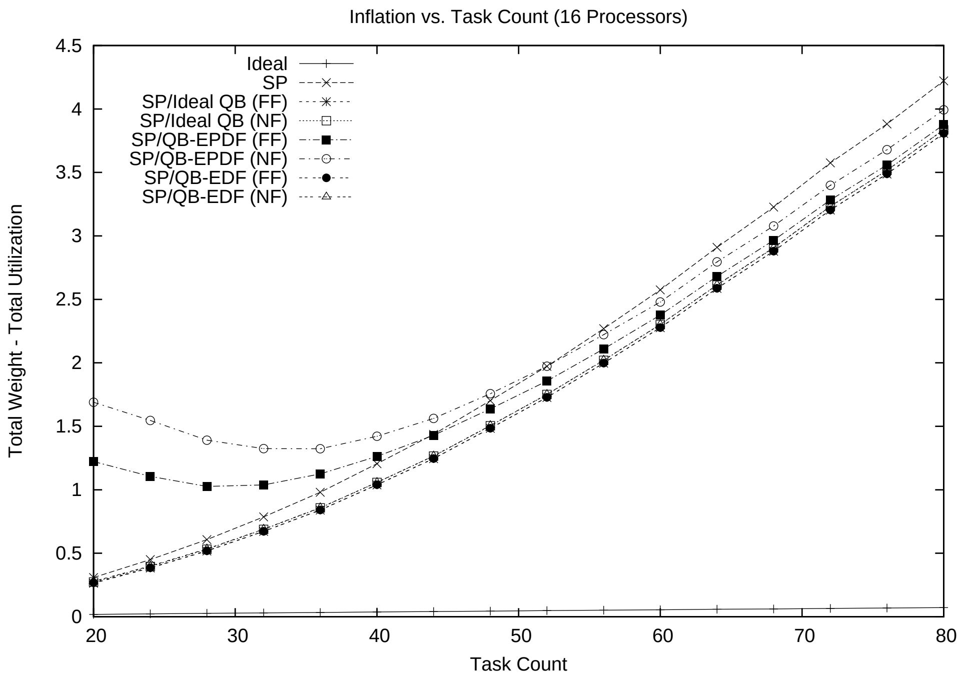


(a) Sample Means Only

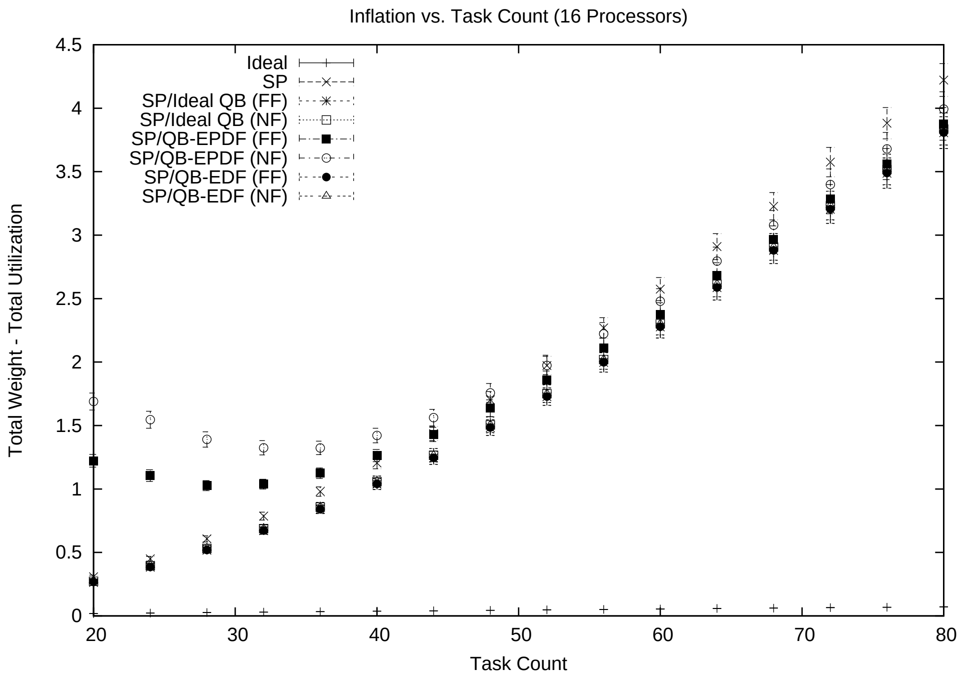


(b) 99% Confidence Interval

Figure 6.63: Plots show how inflation varies as the task count is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

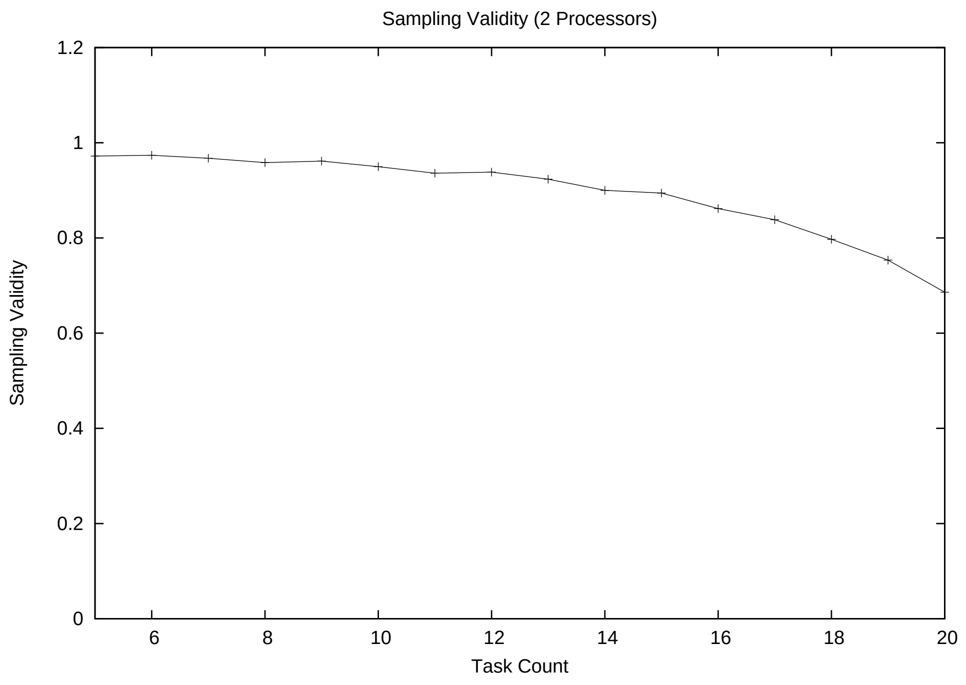


(a) Sample Means Only

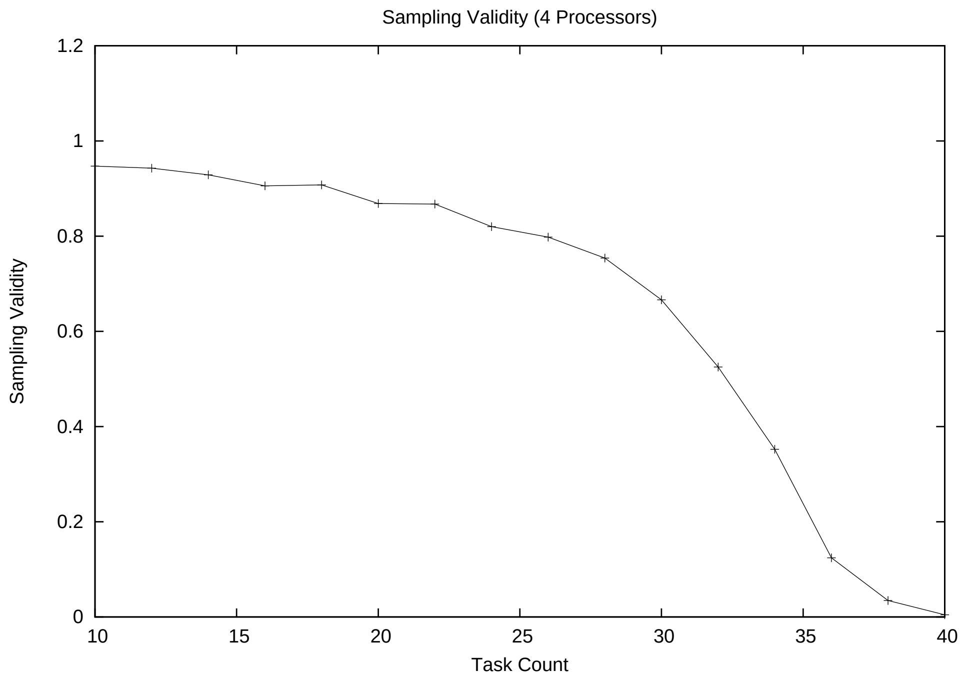


(b) 99% Confidence Interval

Figure 6.64: Plots show how inflation varies as the task count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

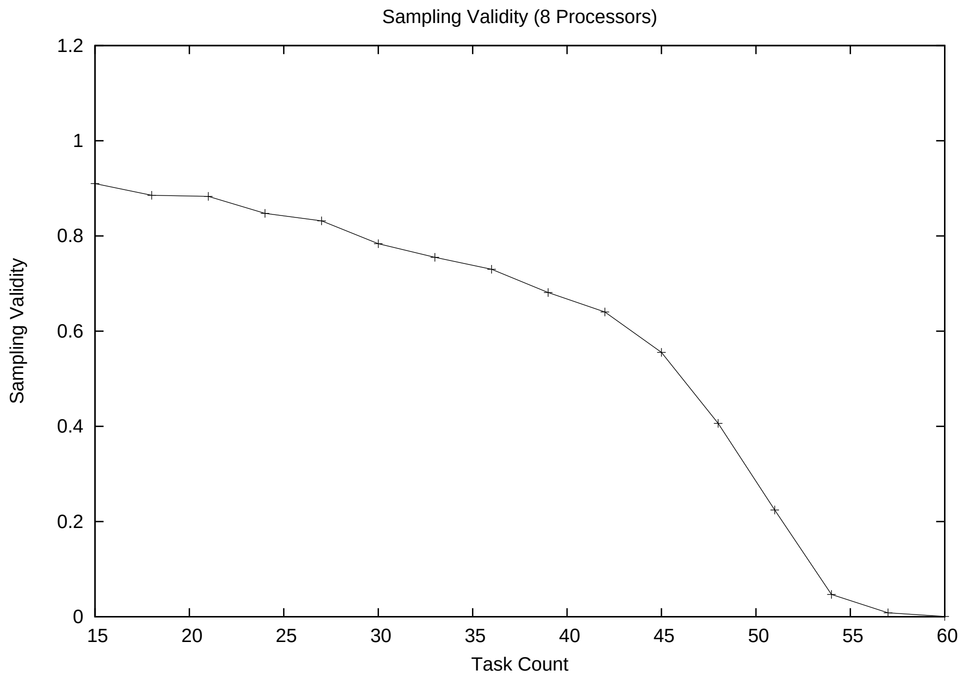


(a) $M = 2$

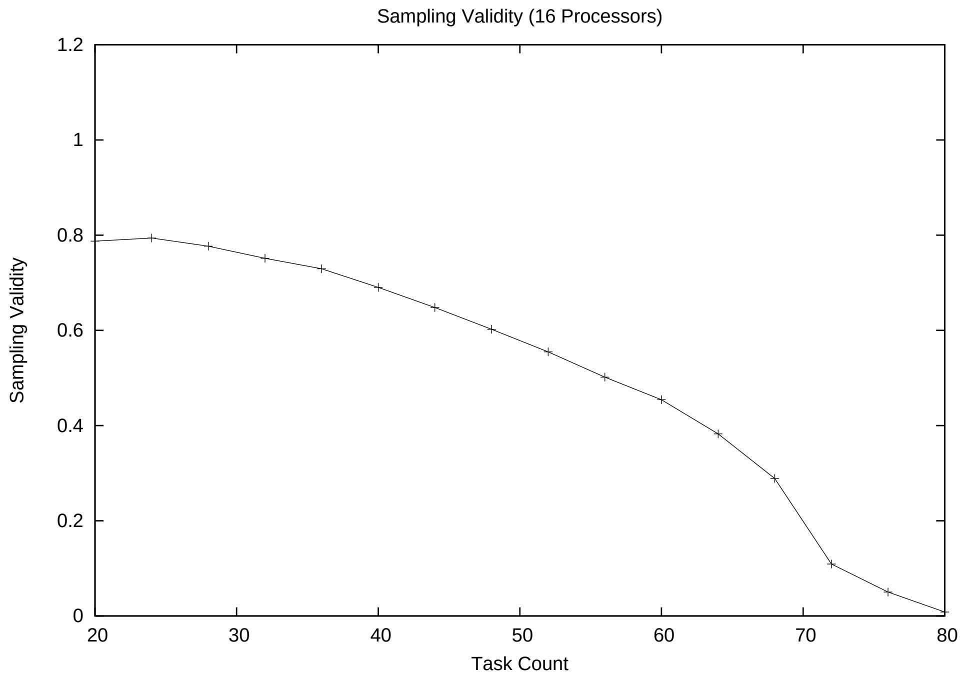


(b) $M = 4$

Figure 6.65: Plots show the sampling validity for the graphs showing inflation plotted against the task count when (R1) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.

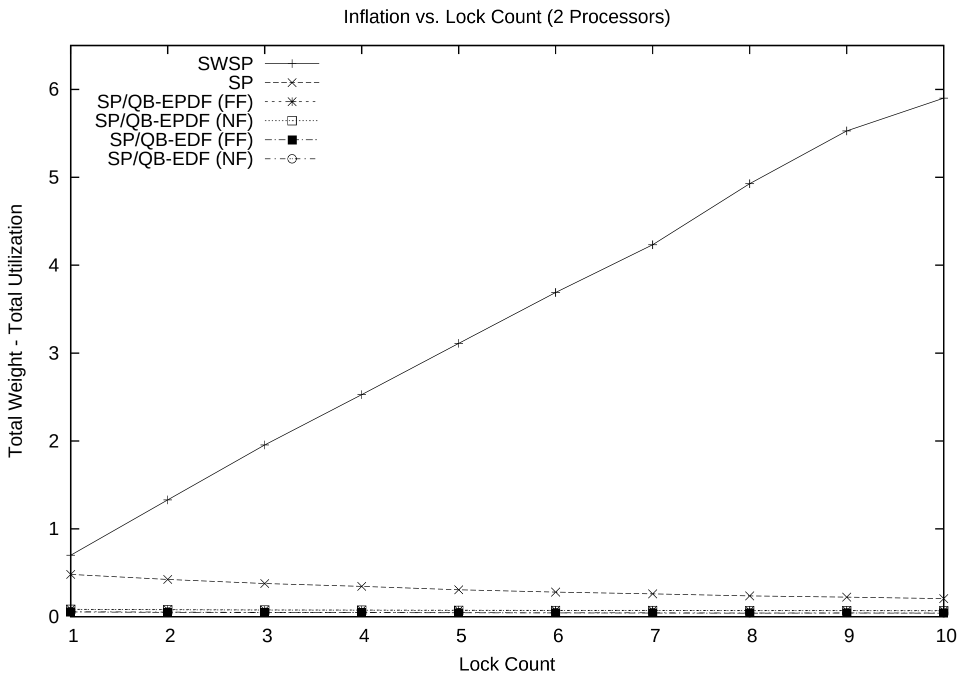


(a) $M = 8$

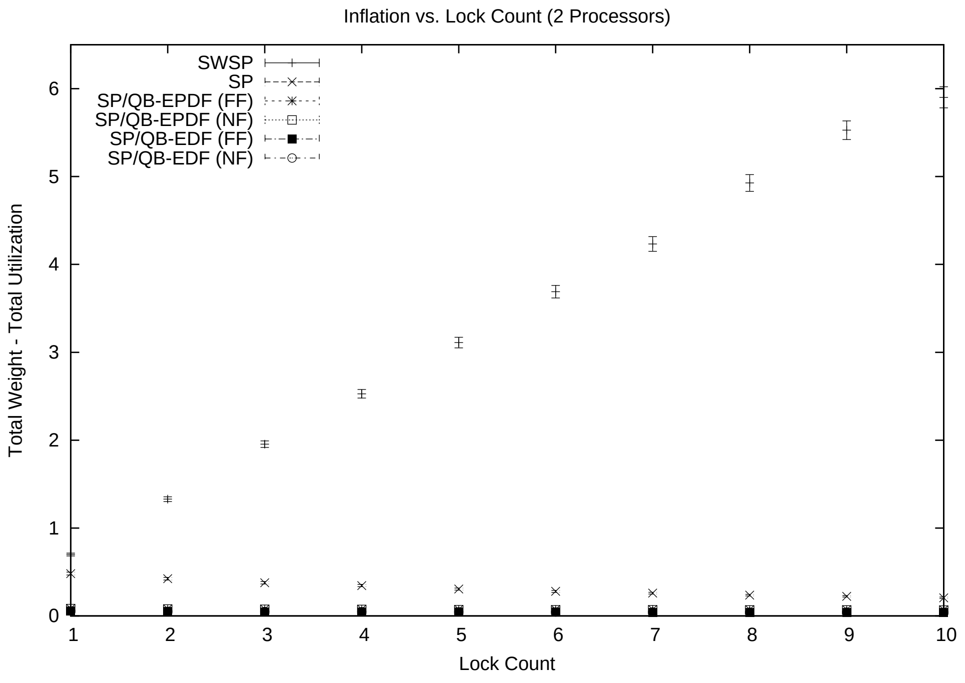


(b) $M = 16$

Figure 6.66: Plots show the sampling validity for the graphs showing inflation plotted against the task count when (R1) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

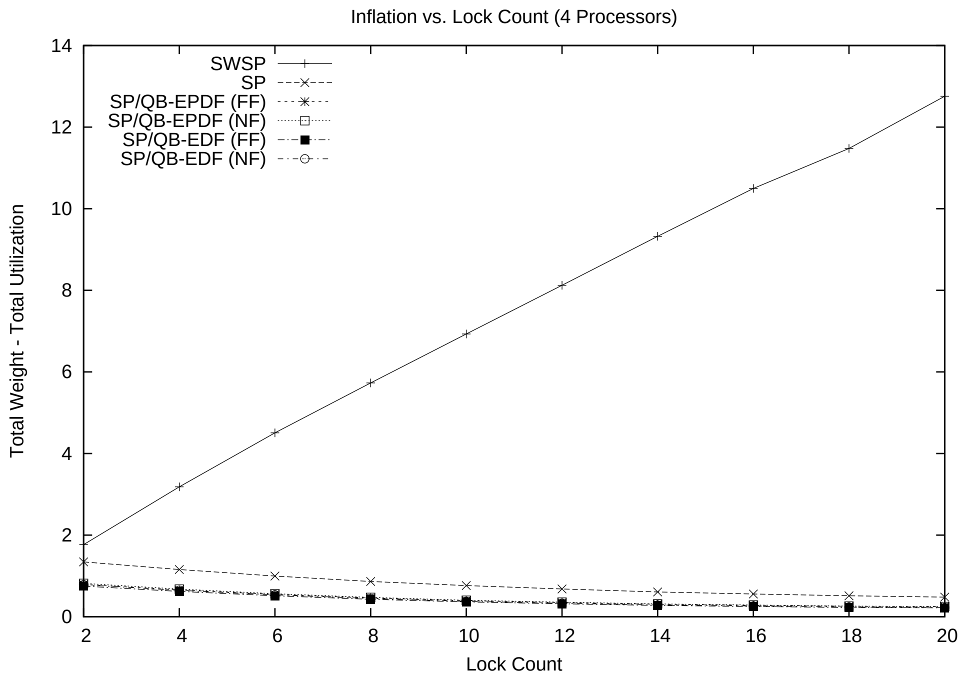


(a) Sample Means Only

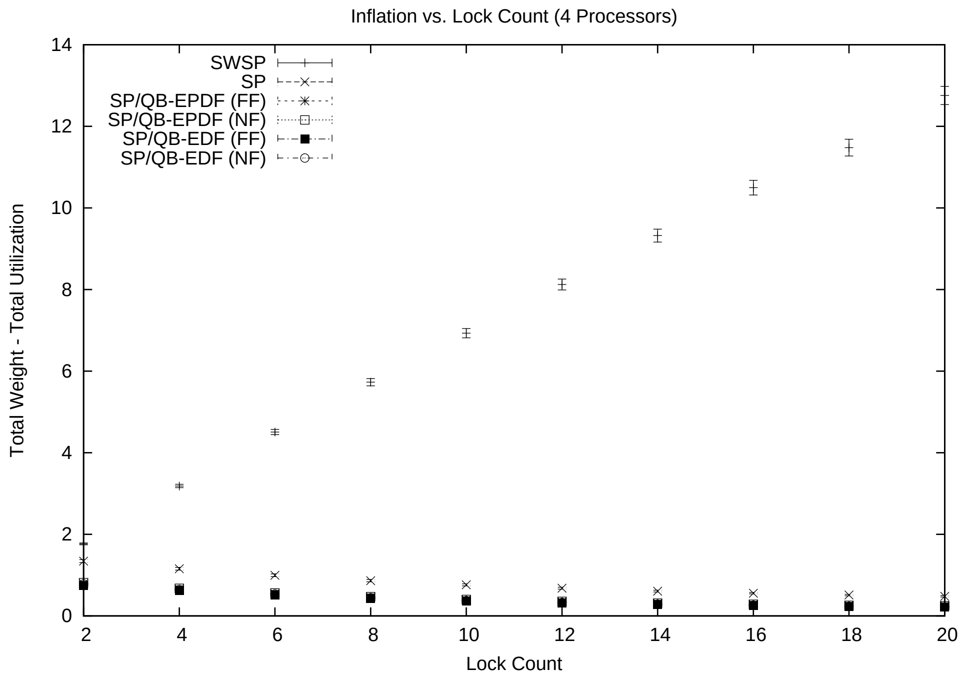


(b) 99% Confidence Interval

Figure 6.67: Plots show how inflation varies as the lock count is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

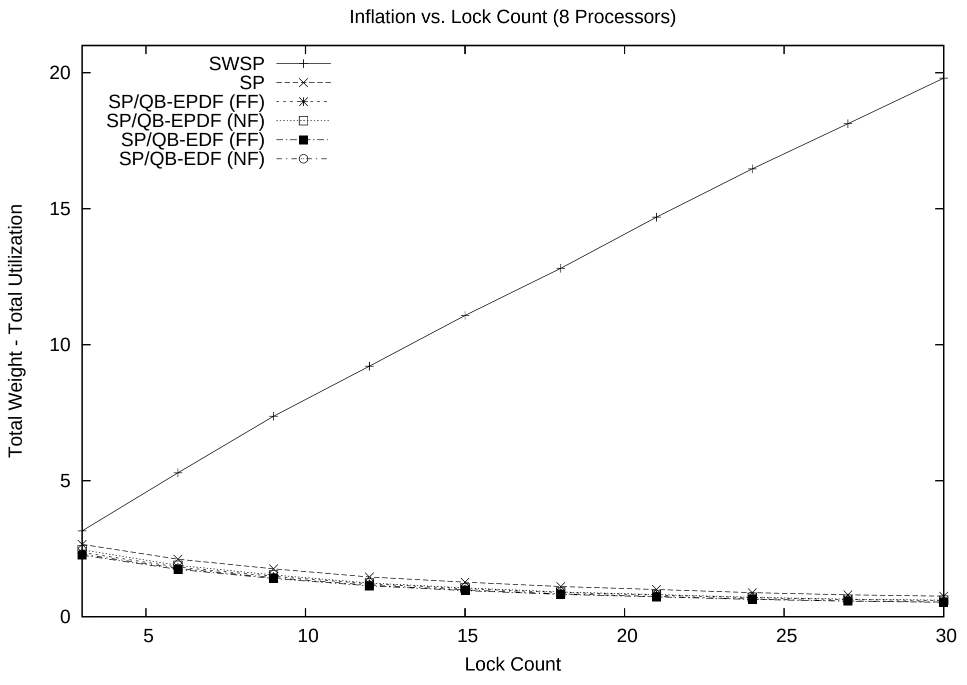


(a) Sample Means Only

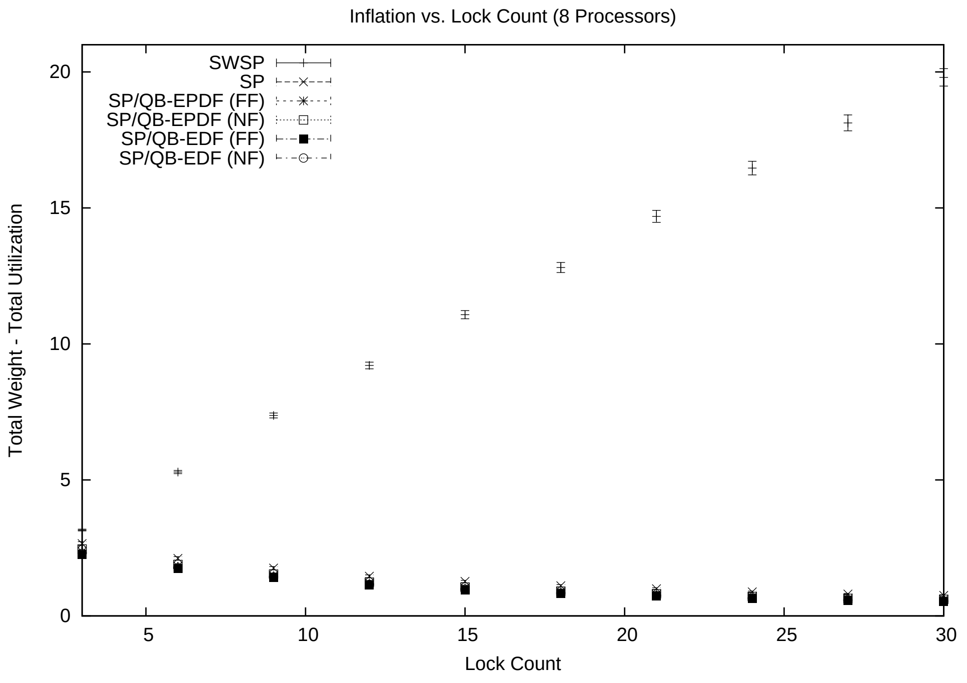


(b) 99% Confidence Interval

Figure 6.68: Plots show how inflation varies as the lock count is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

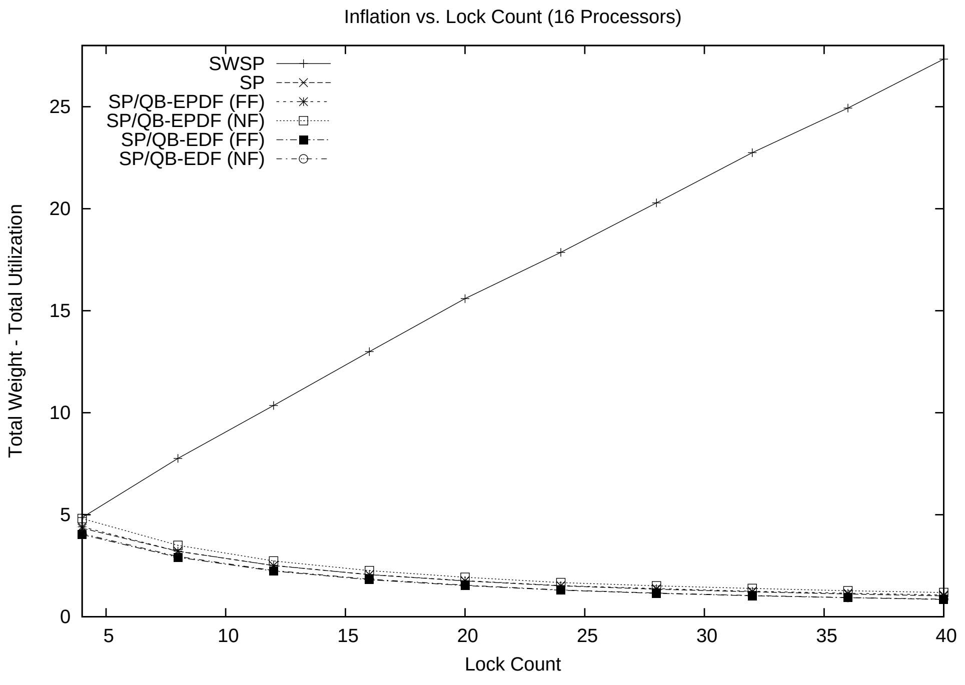


(a) Sample Means Only

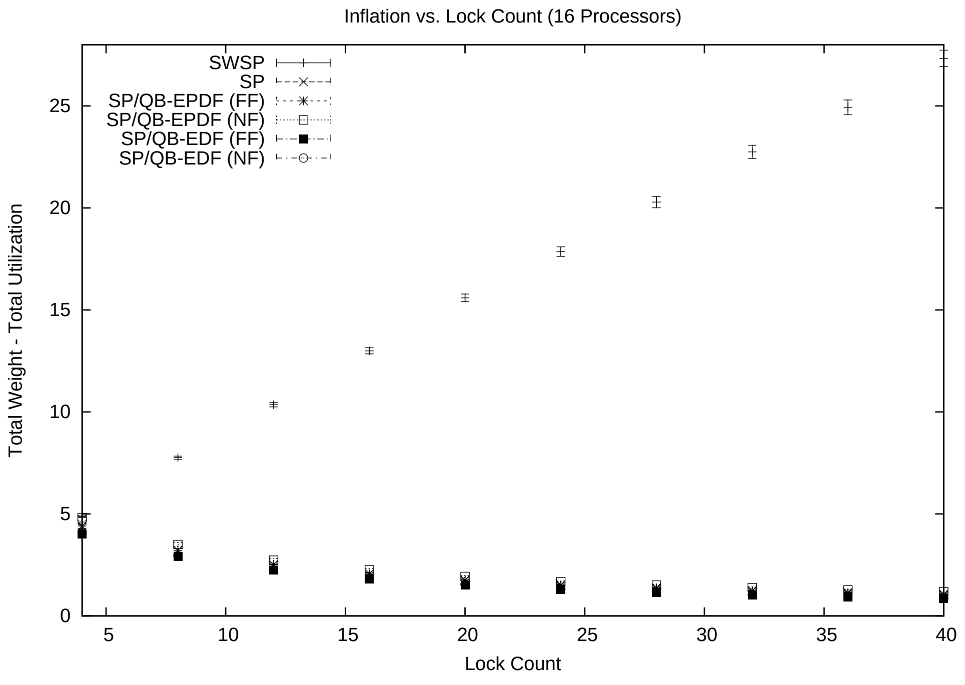


(b) 99% Confidence Interval

Figure 6.69: Plots show how inflation varies as the lock count is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

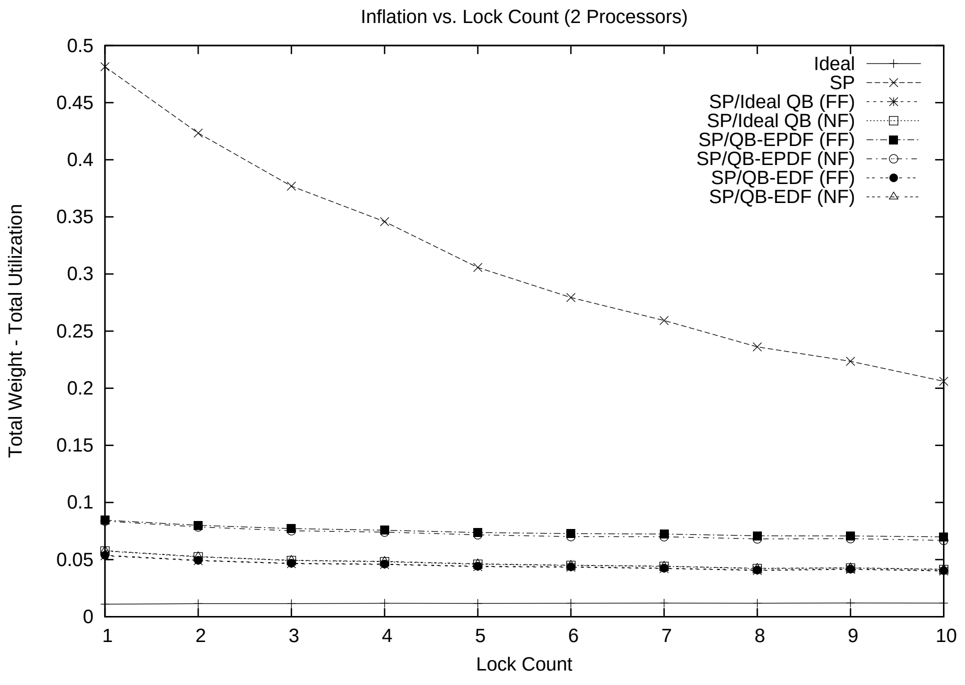


(a) Sample Means Only

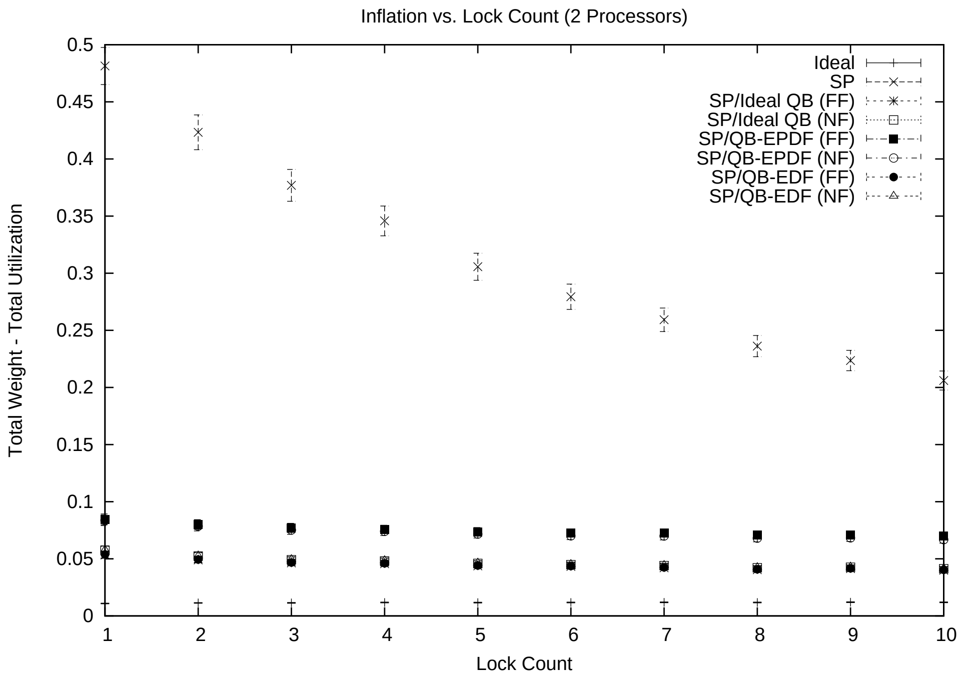


(b) 99% Confidence Interval

Figure 6.70: Plots show how inflation varies as the lock count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

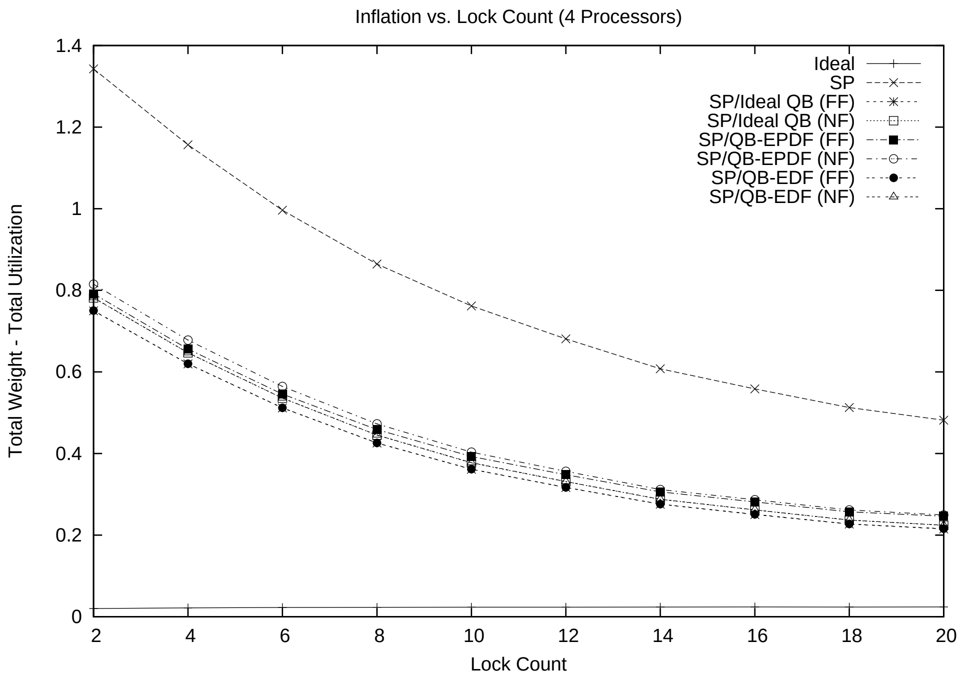


(a) Sample Means Only

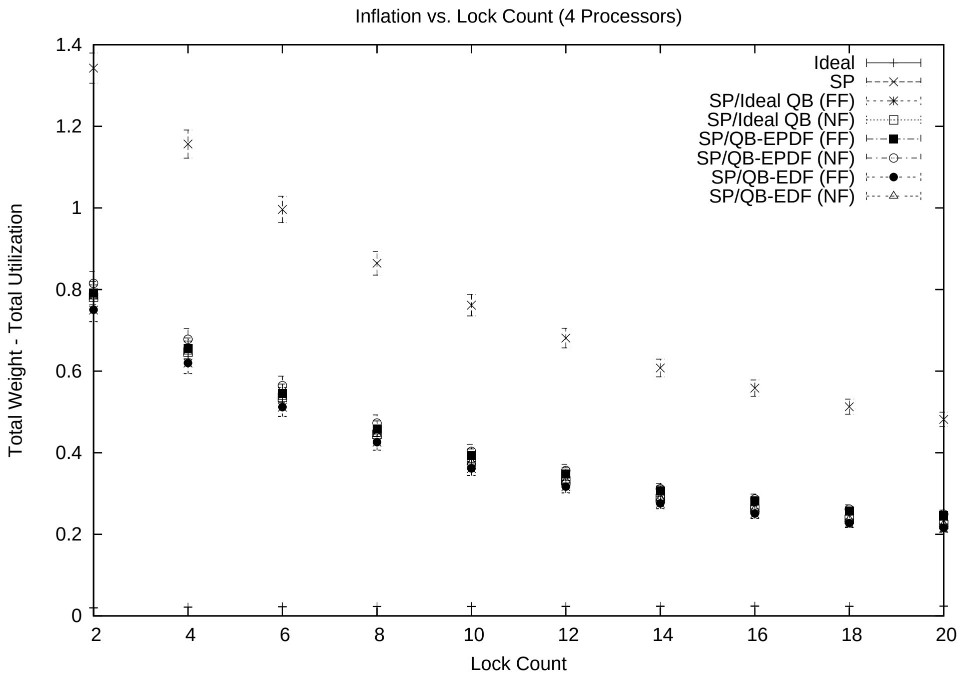


(b) 99% Confidence Interval

Figure 6.71: Plots show how inflation varies as the lock count is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

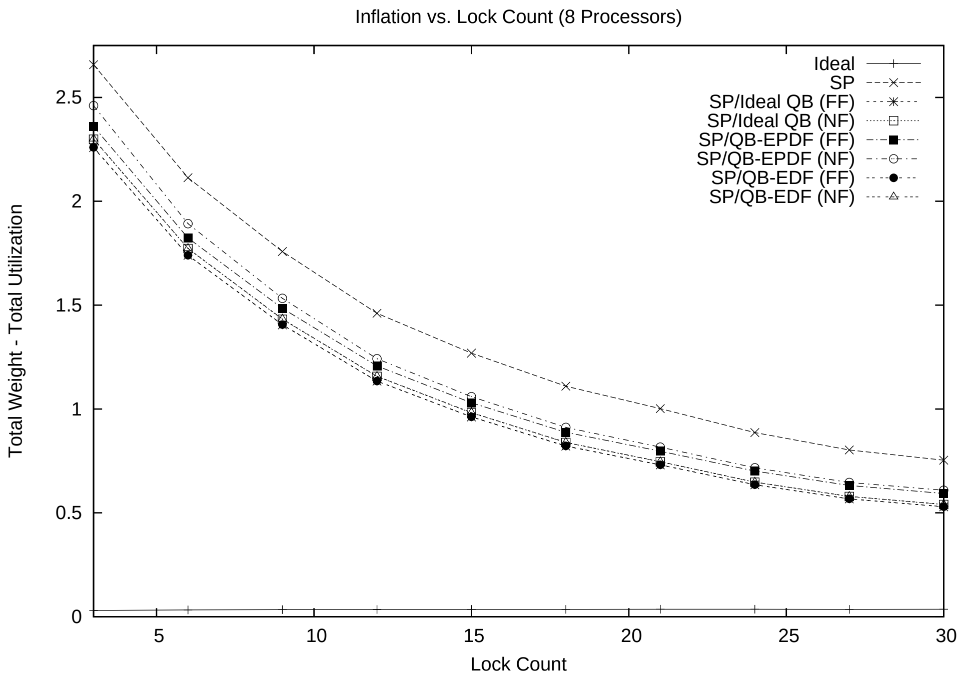


(a) Sample Means Only

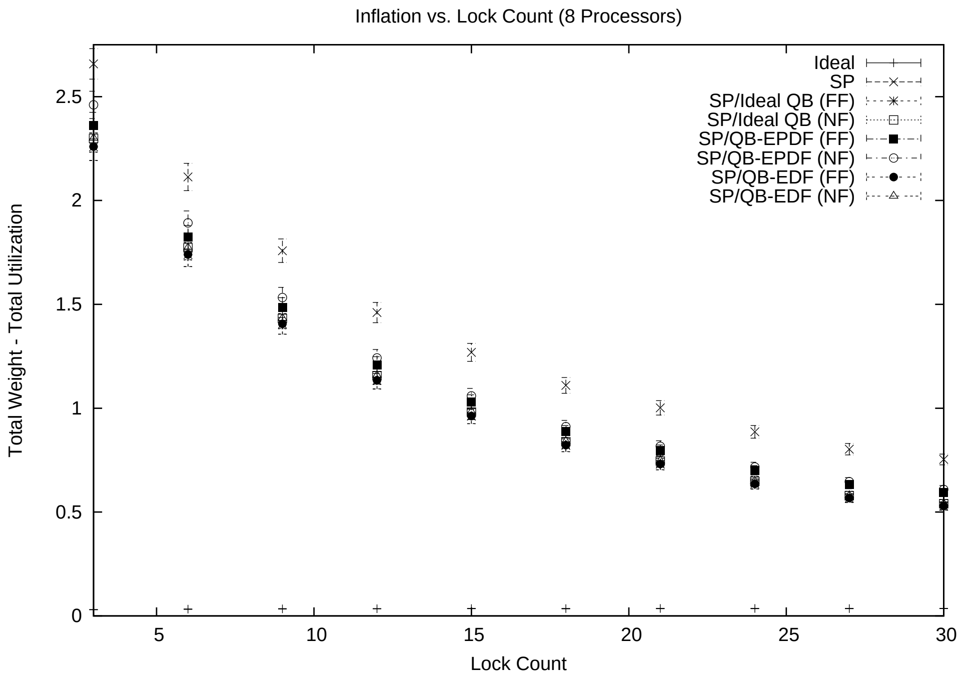


(b) 99% Confidence Interval

Figure 6.72: Plots show how inflation varies as the lock count is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

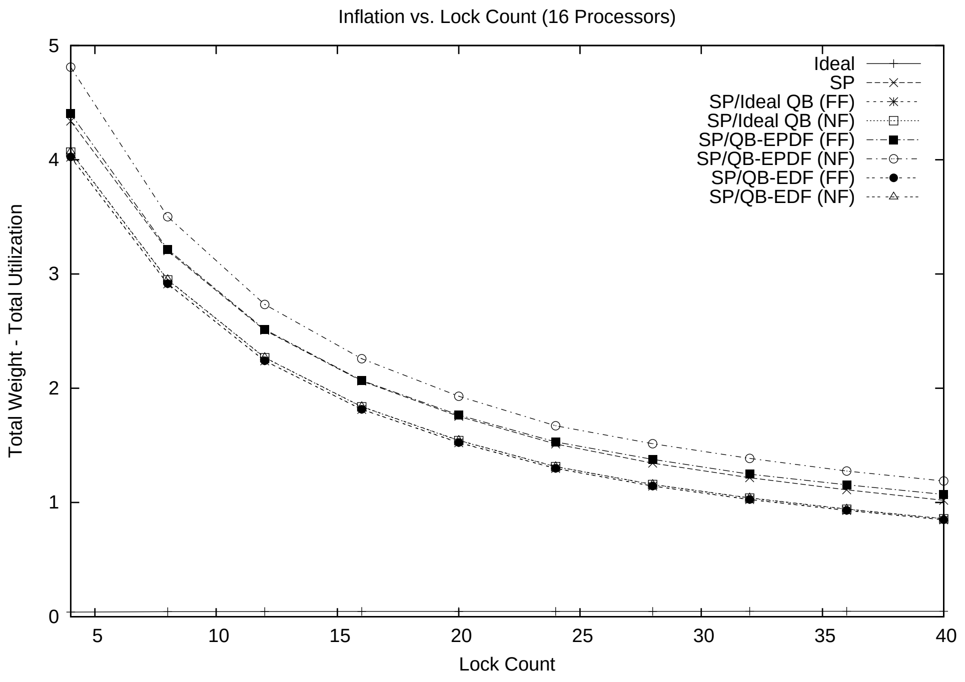


(a) Sample Means Only

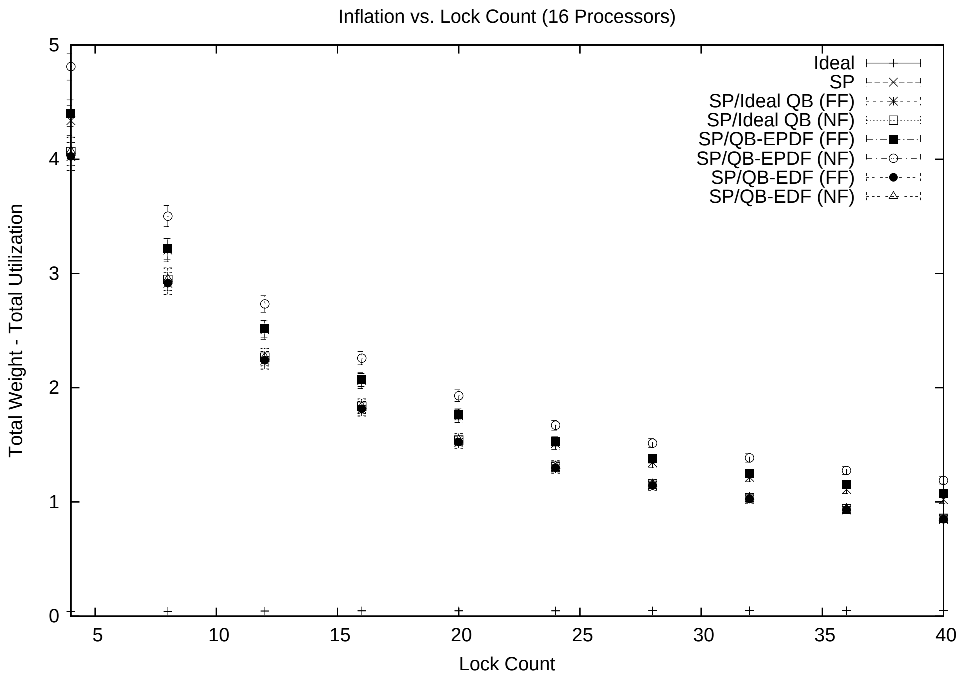


(b) 99% Confidence Interval

Figure 6.73: Plots show how inflation varies as the lock count is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

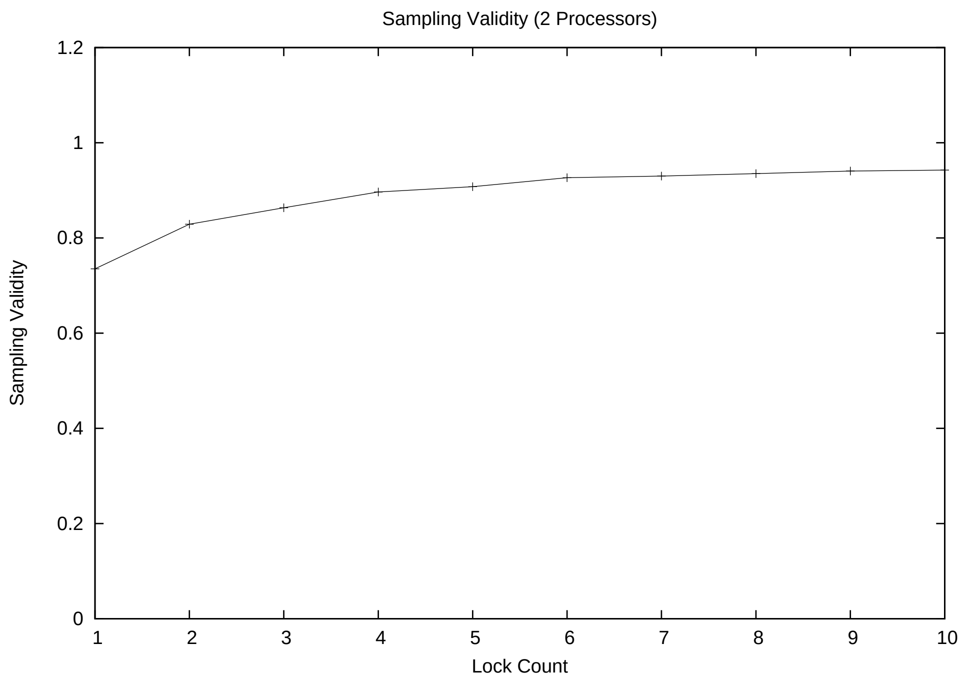


(a) Sample Means Only

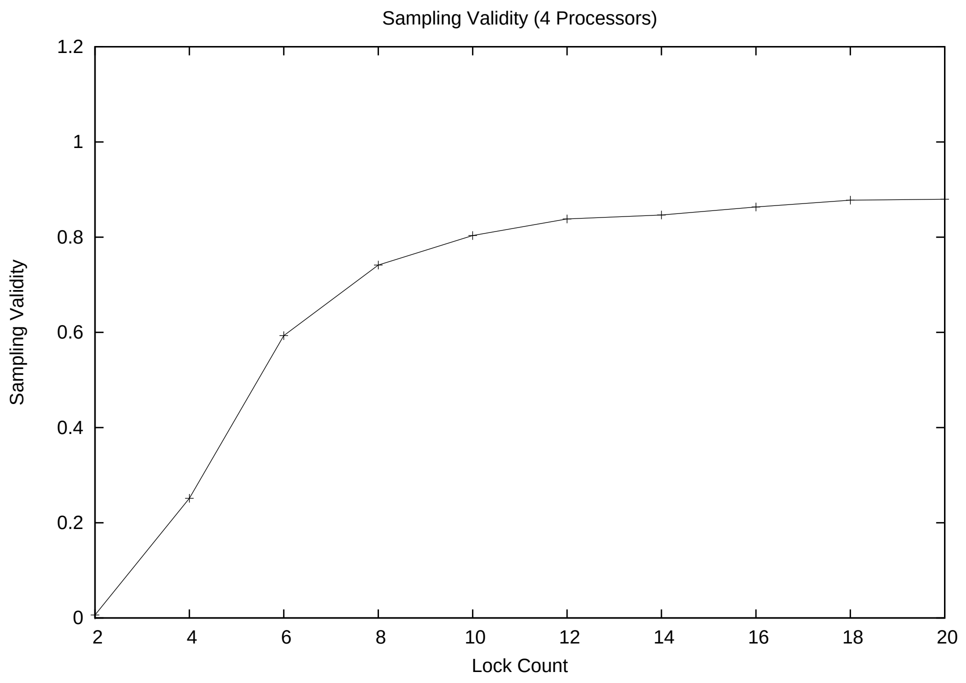


(b) 99% Confidence Interval

Figure 6.74: Plots show how inflation varies as the lock count is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

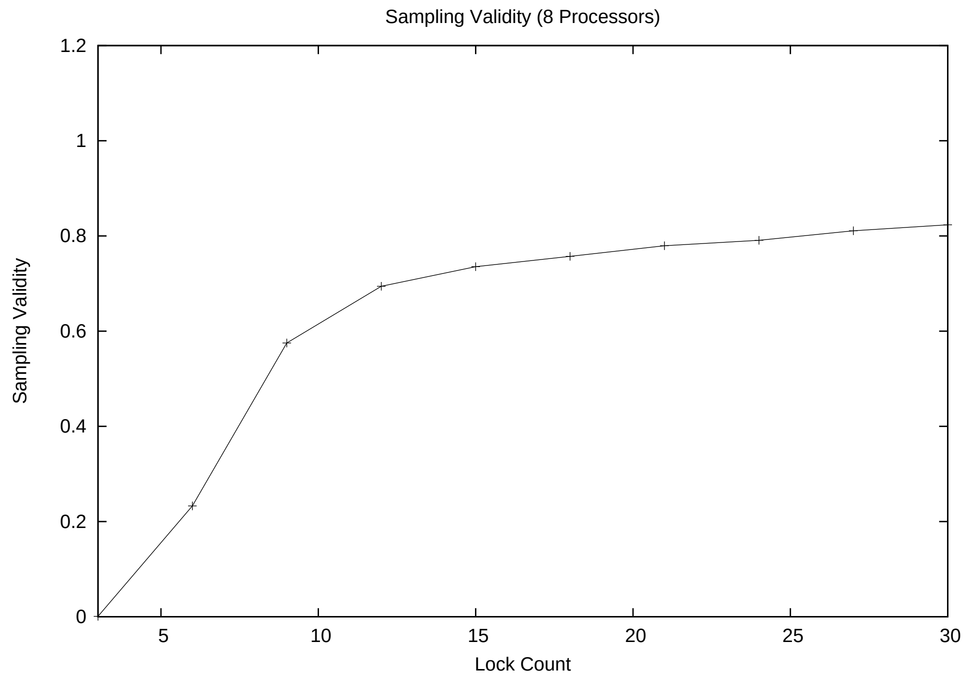


(a) $M = 2$

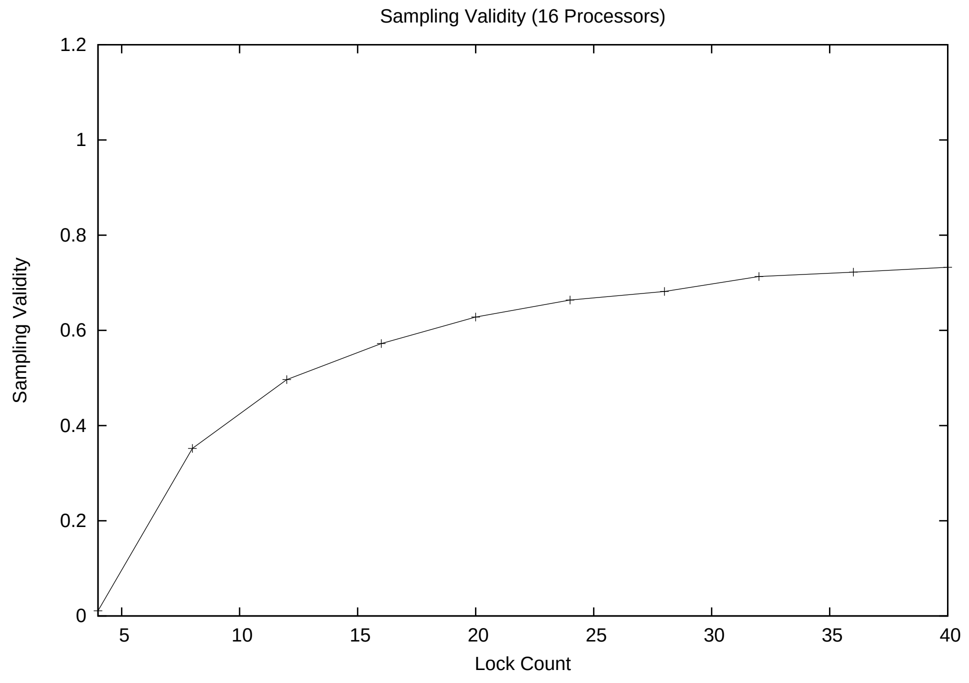


(b) $M = 4$

Figure 6.75: Plots show the sampling validity for the graphs showing inflation plotted against the lock count when (R1) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.

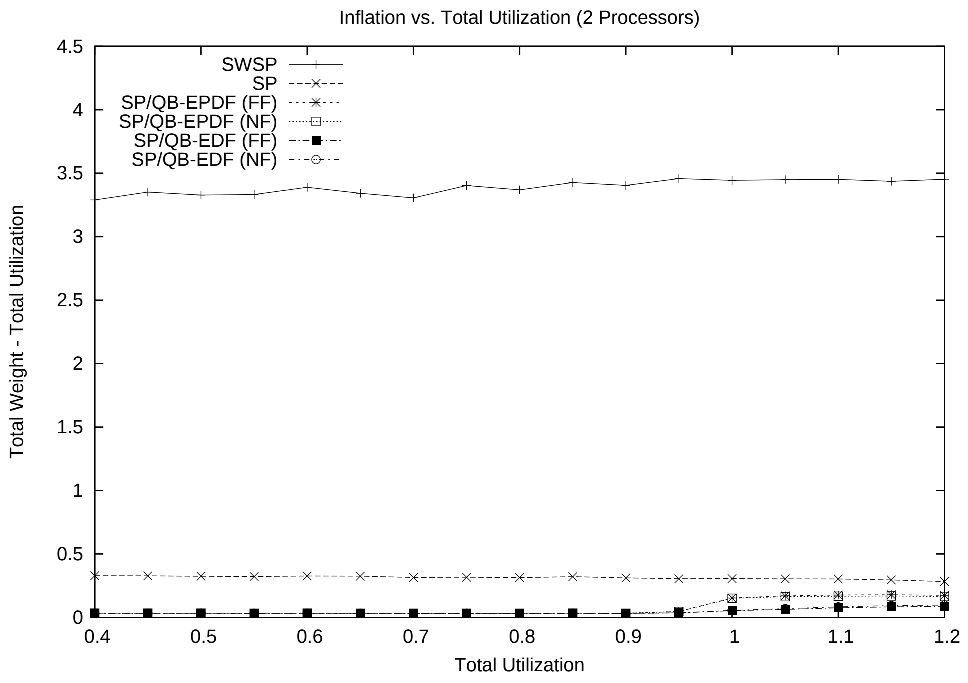


(a) $M = 8$

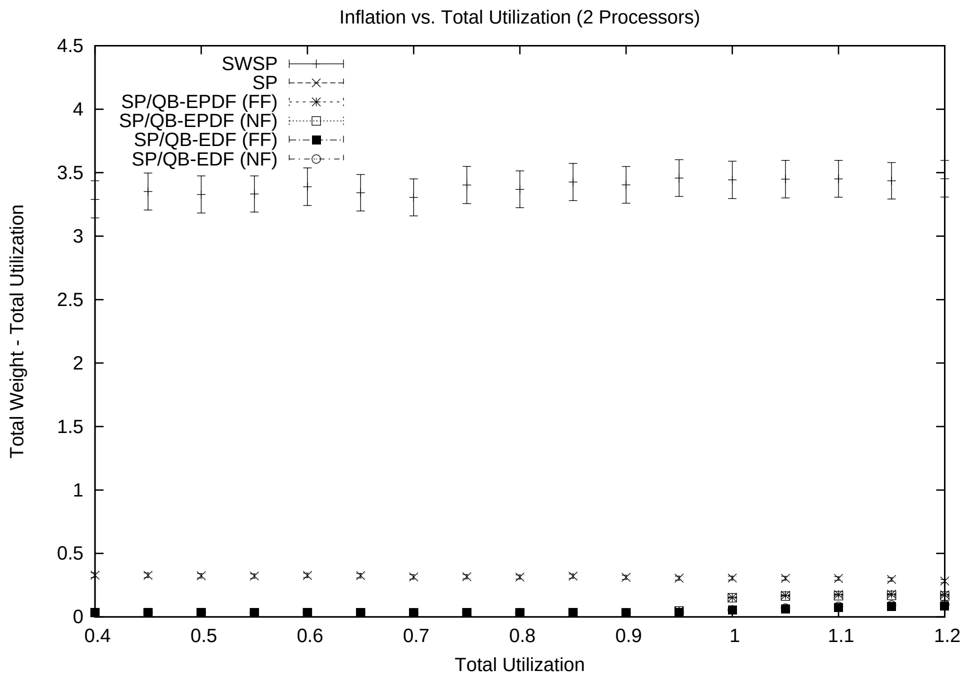


(b) $M = 16$

Figure 6.76: Plots show the sampling validity for the graphs showing inflation plotted against the lock count when (R1) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

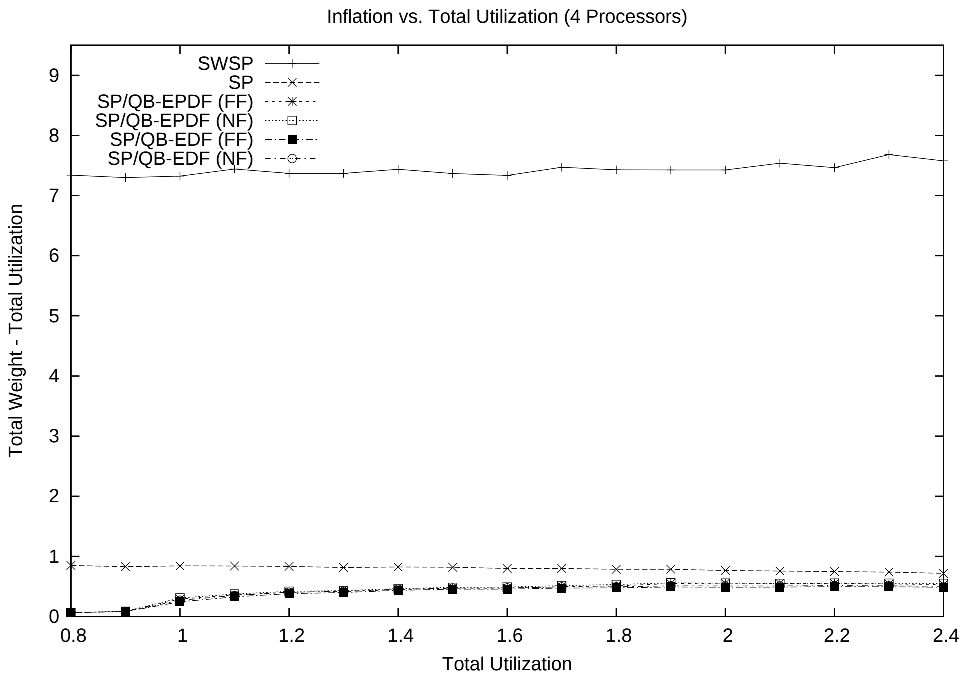


(a) Sample Means Only

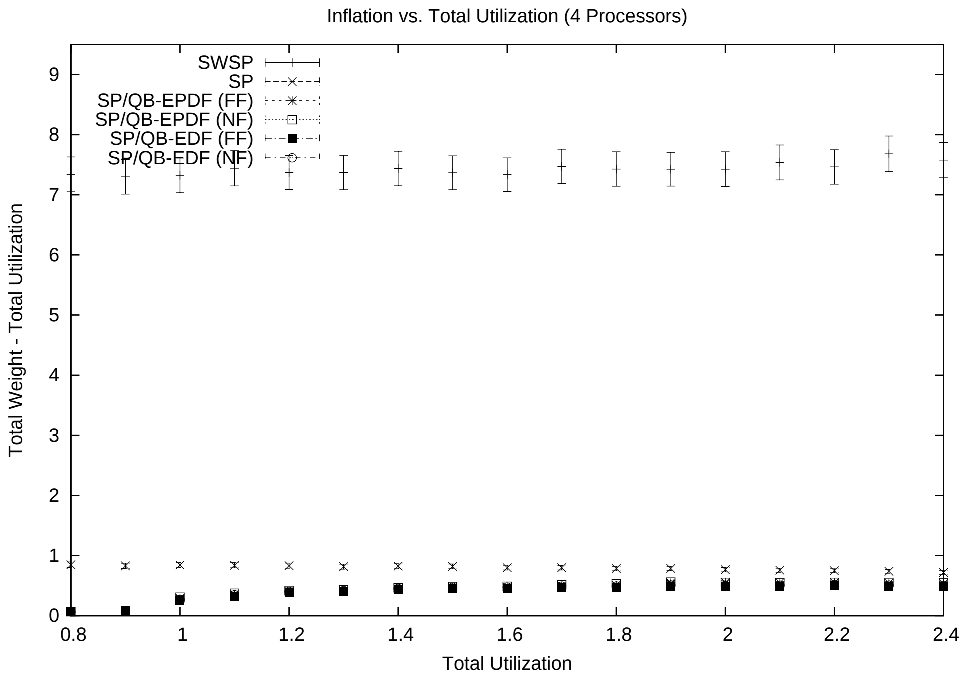


(b) 99% Confidence Interval

Figure 6.77: Plots show how inflation varies as utilization is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

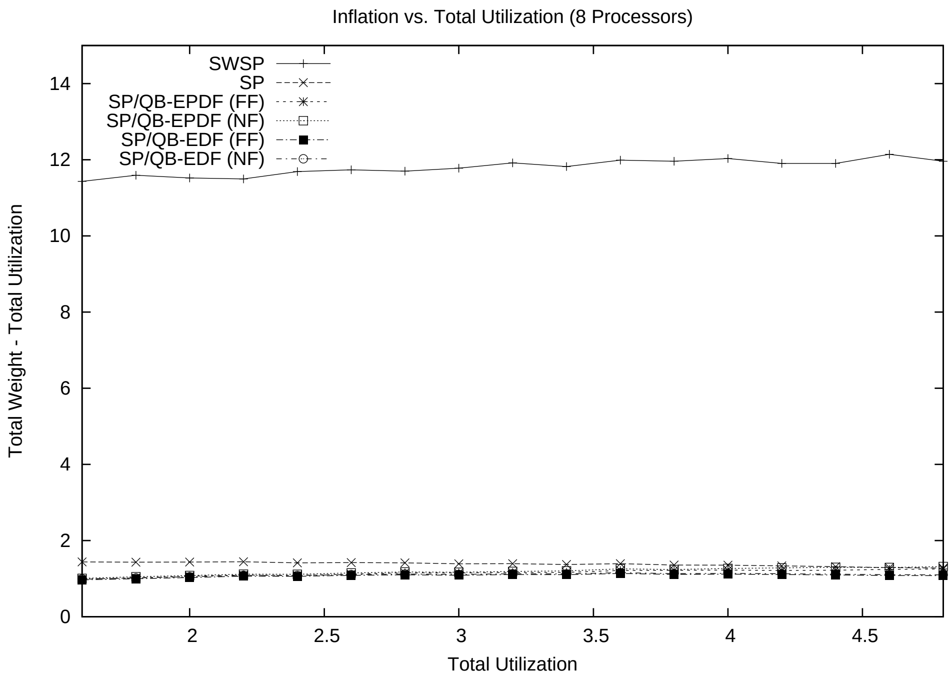


(a) Sample Means Only

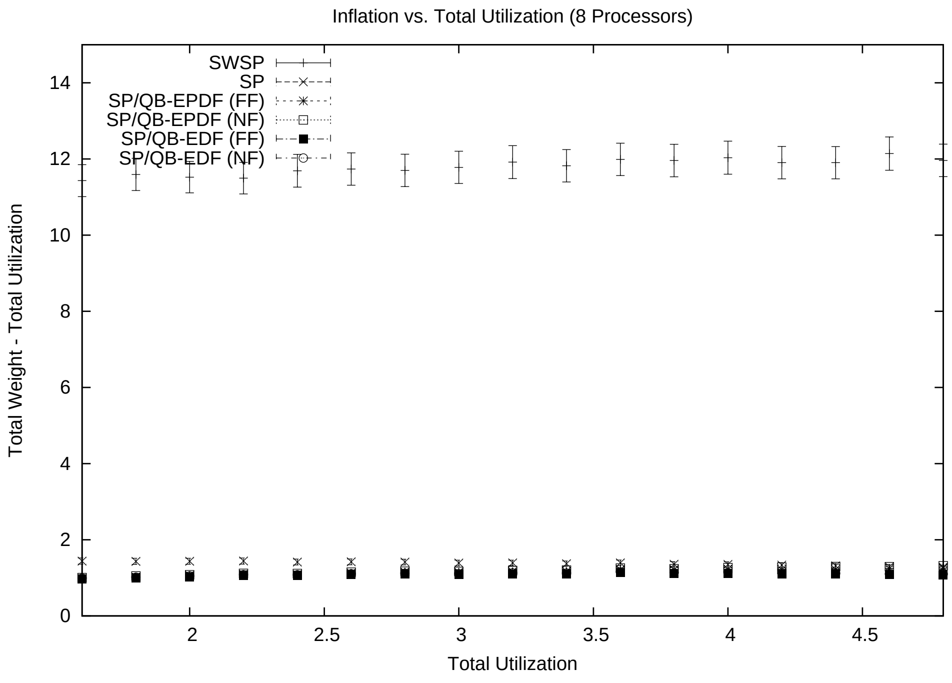


(b) 99% Confidence Interval

Figure 6.78: Plots show how inflation varies as utilization is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

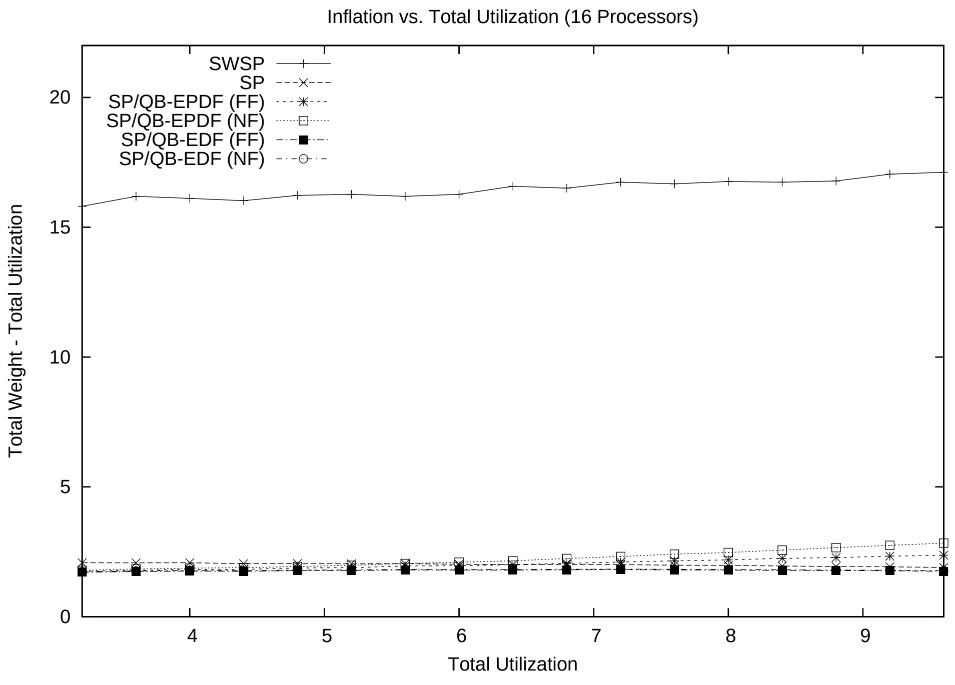


(a) Sample Means Only

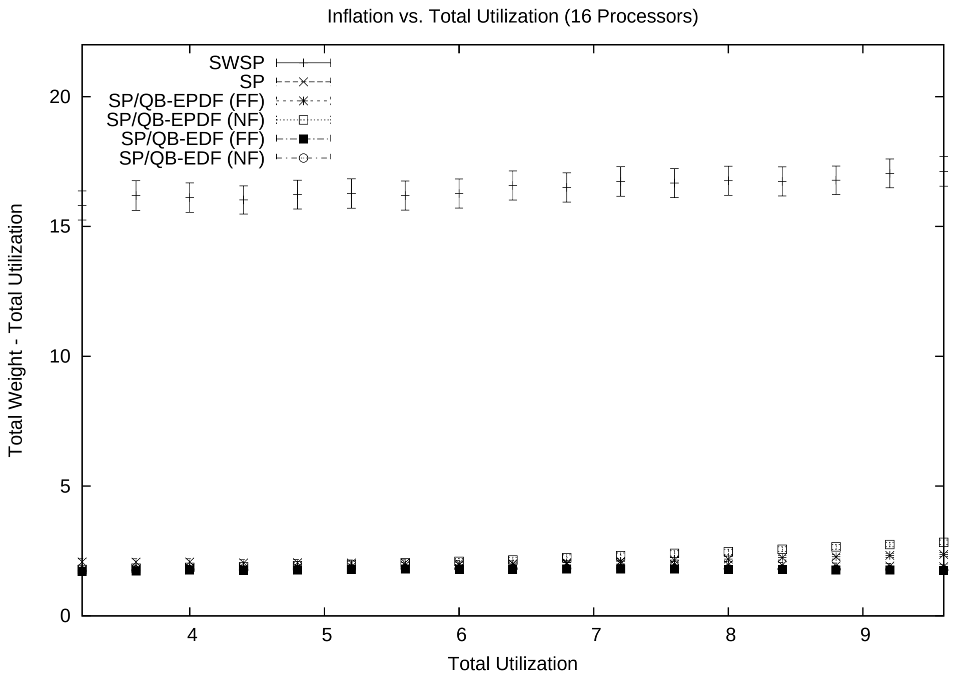


(b) 99% Confidence Interval

Figure 6.79: Plots show how inflation varies as utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

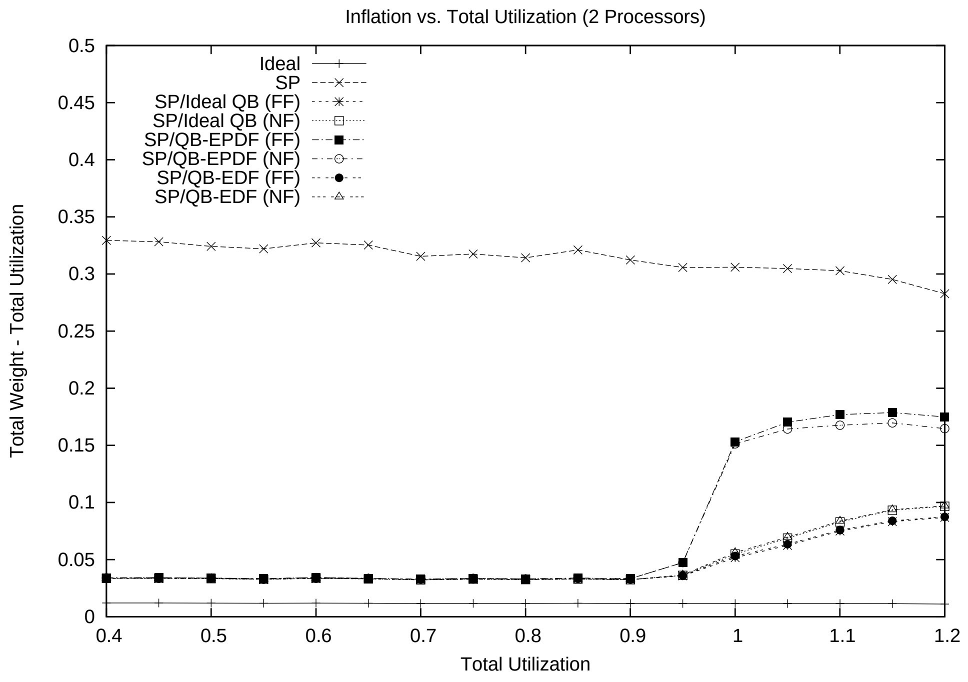


(a) Sample Means Only

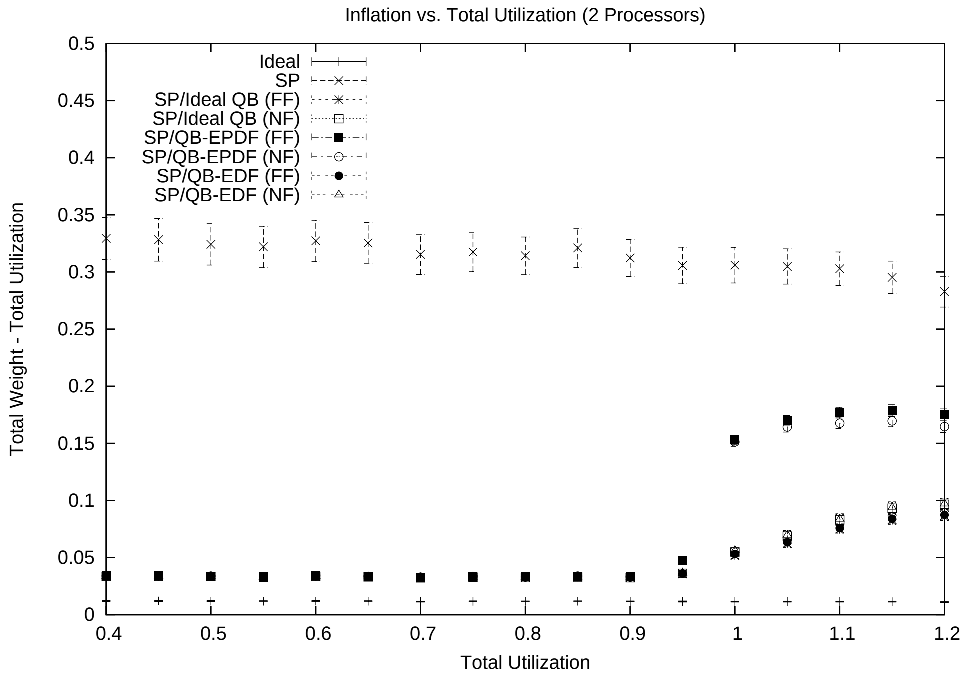


(b) 99% Confidence Interval

Figure 6.80: Plots show how inflation varies as utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

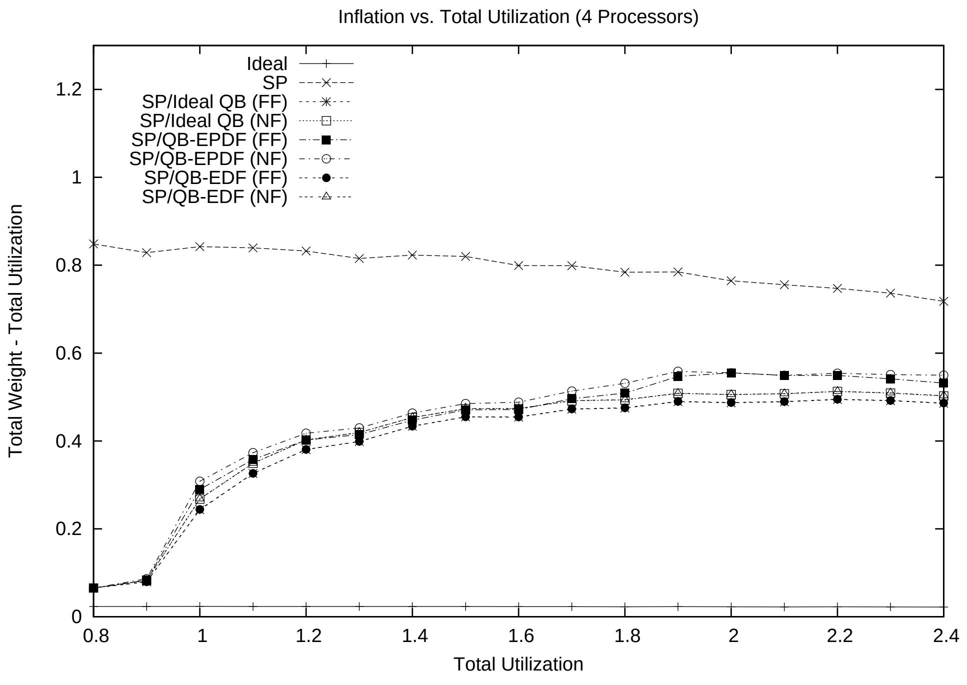


(a) Sample Means Only

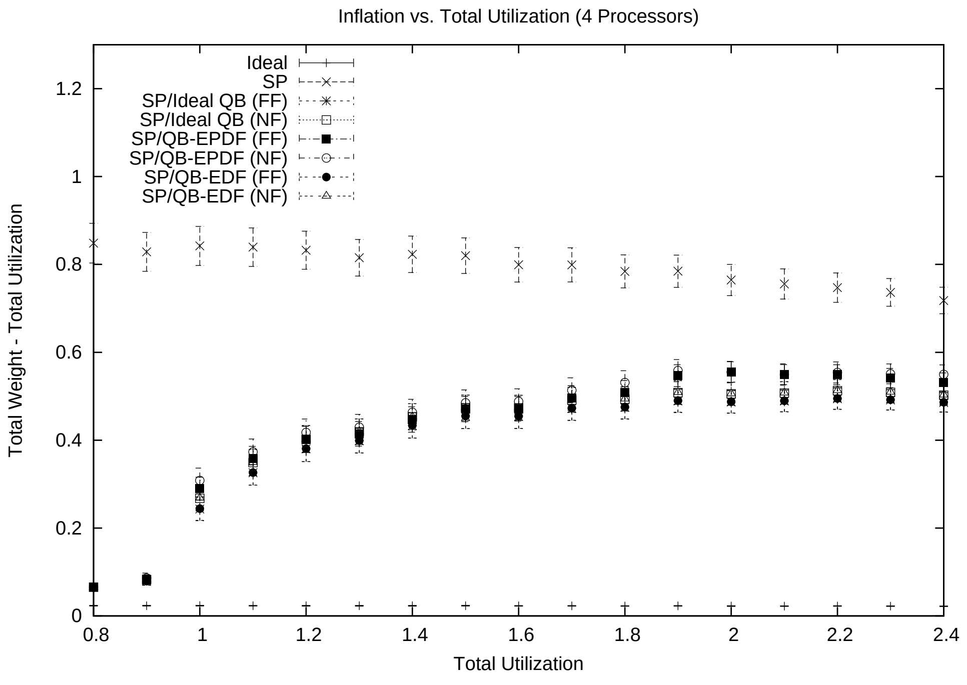


(b) 99% Confidence Interval

Figure 6.81: Plots show how inflation varies as utilization is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

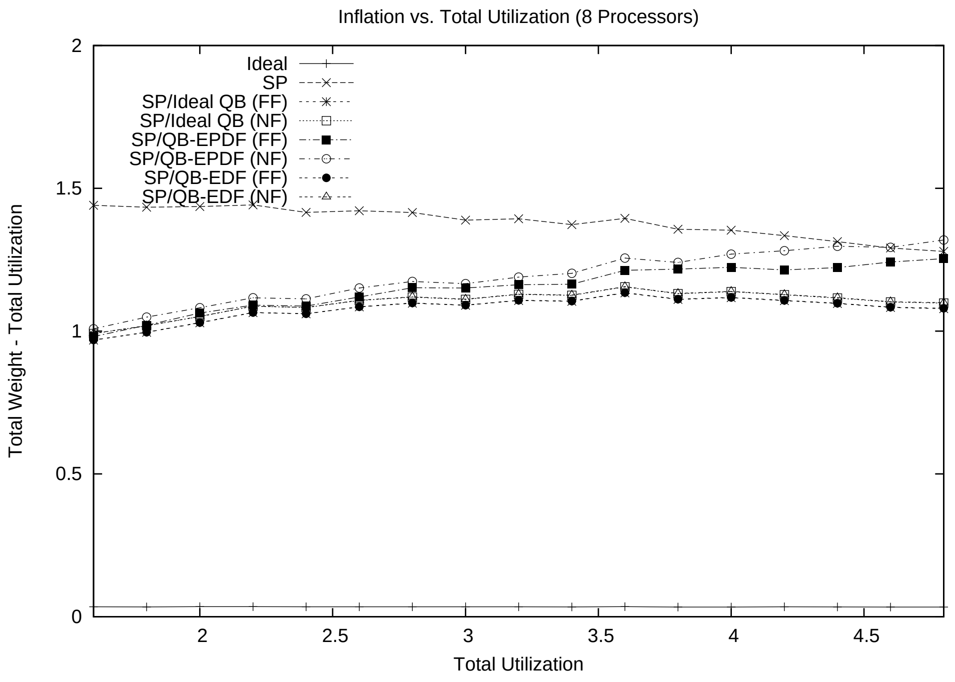


(a) Sample Means Only

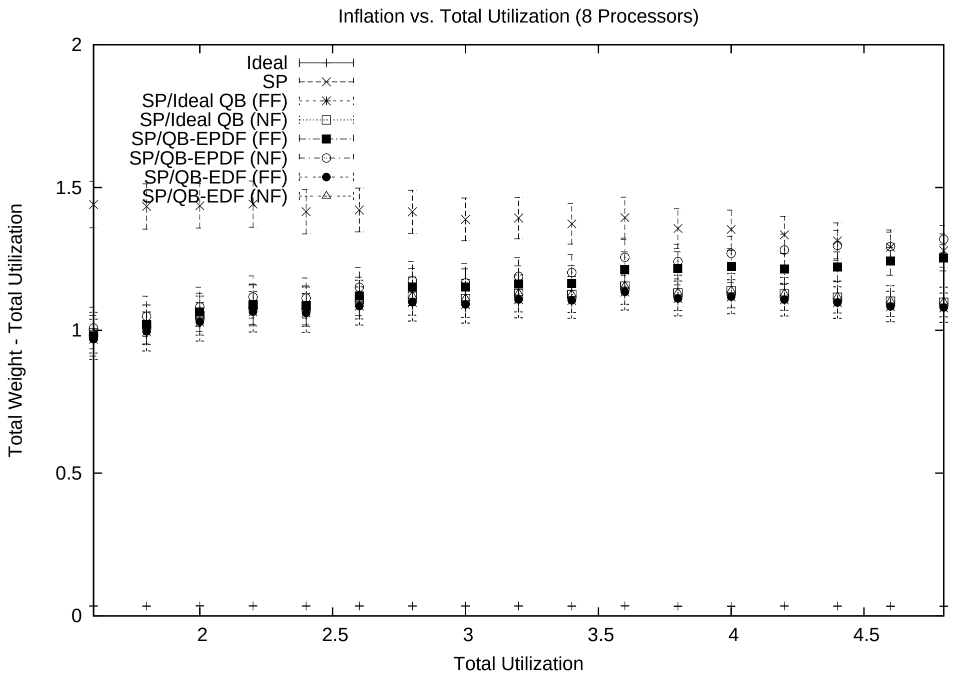


(b) 99% Confidence Interval

Figure 6.82: Plots show how inflation varies as utilization is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

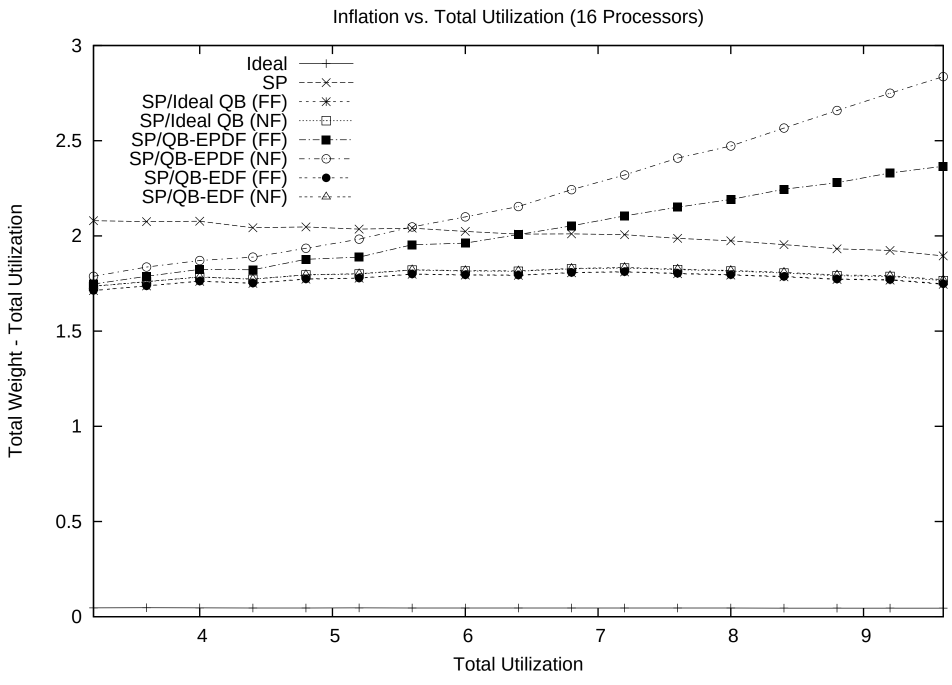


(a) Sample Means Only

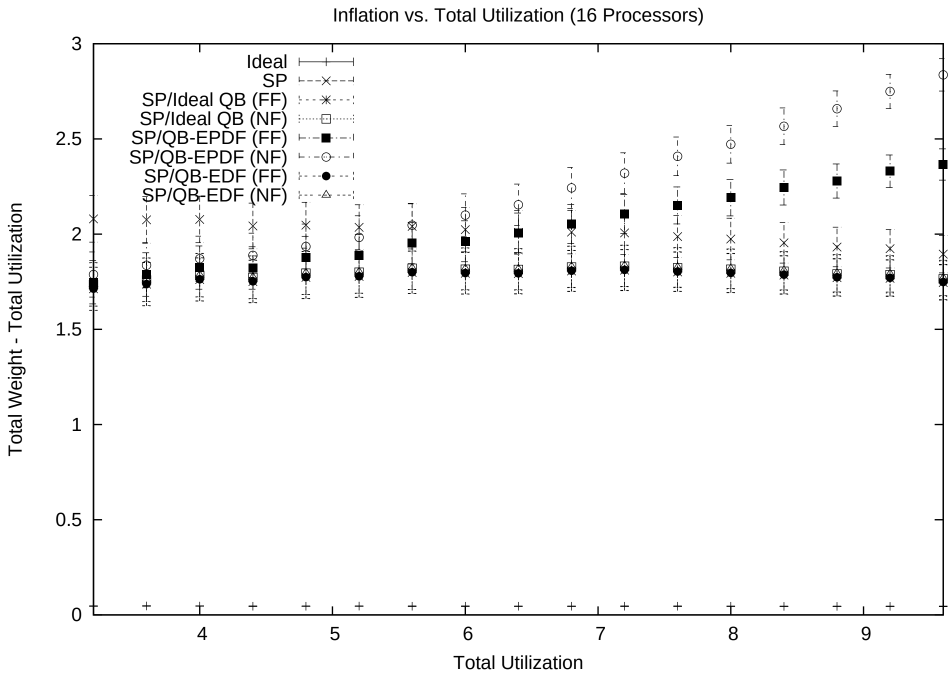


(b) 99% Confidence Interval

Figure 6.83: Plots show how inflation varies as utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

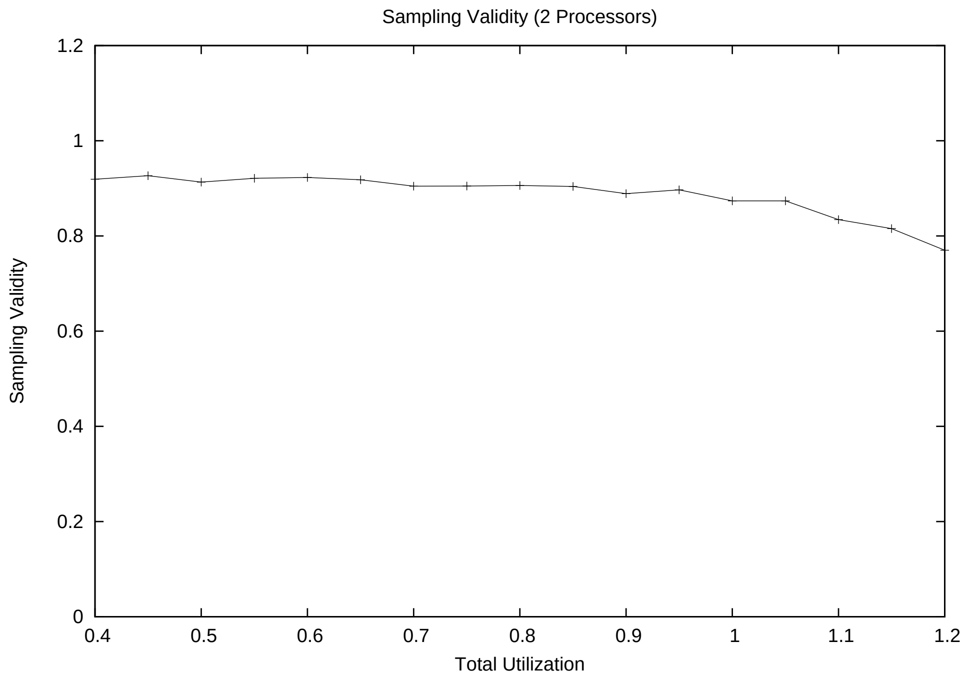


(a) Sample Means Only

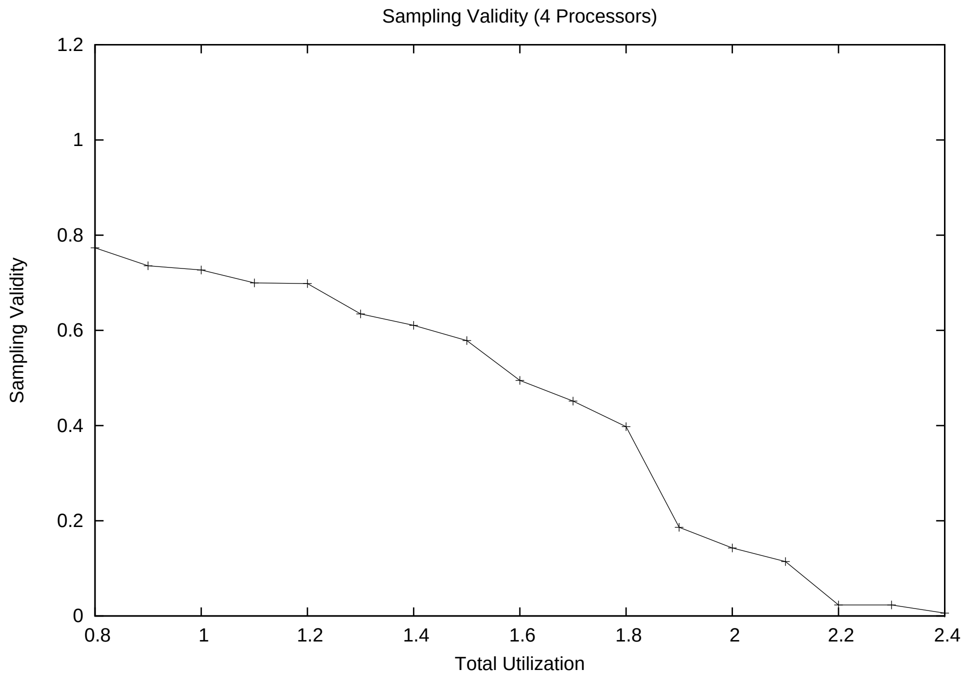


(b) 99% Confidence Interval

Figure 6.84: Plots show how inflation varies as utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

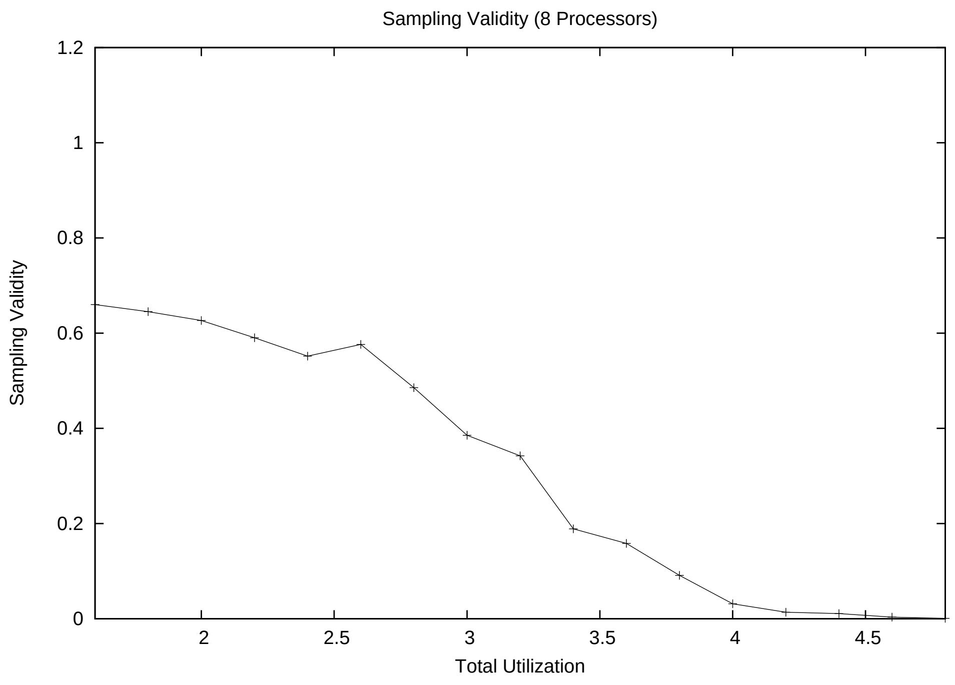


(a) $M = 2$

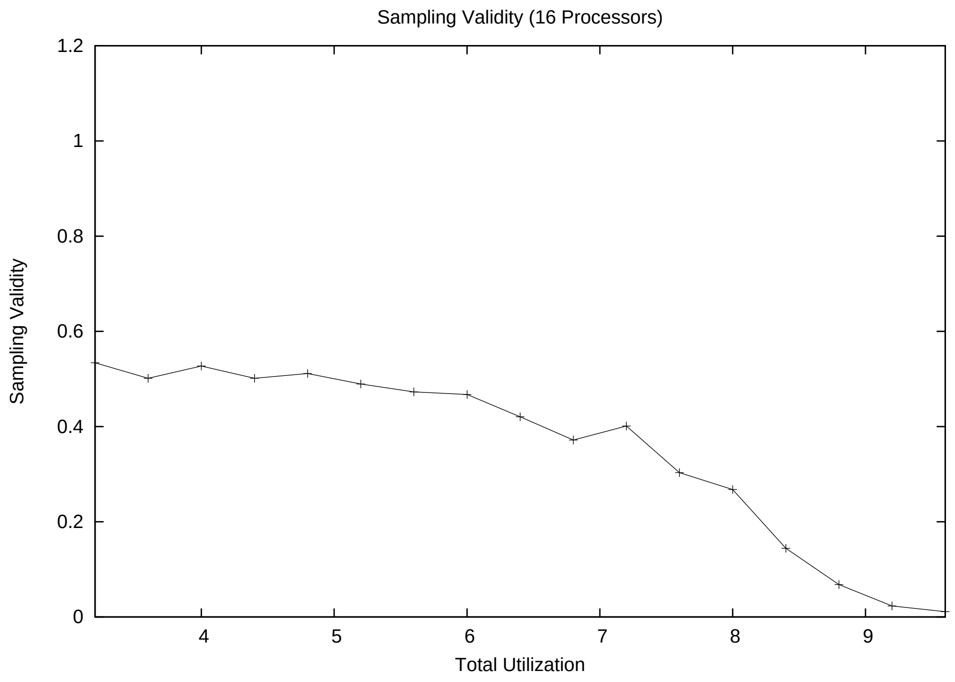


(b) $M = 4$

Figure 6.85: Plots show the sampling validity for the graphs showing inflation plotted against utilization when (R1) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



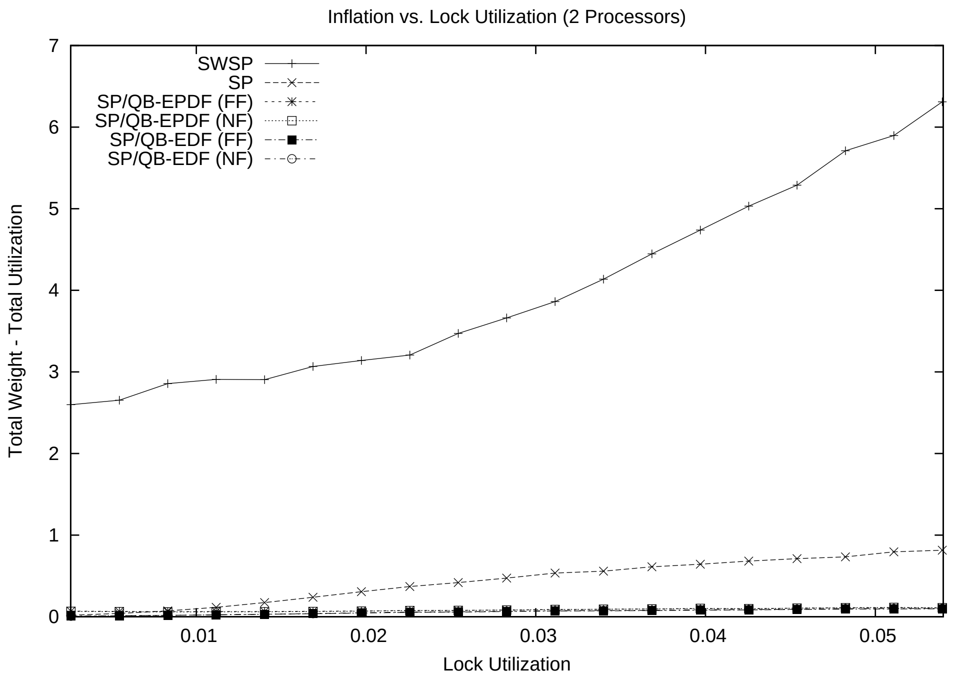
(b) $M = 16$

Figure 6.86: Plots show the sampling validity for the graphs showing inflation plotted against utilization when (R1) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

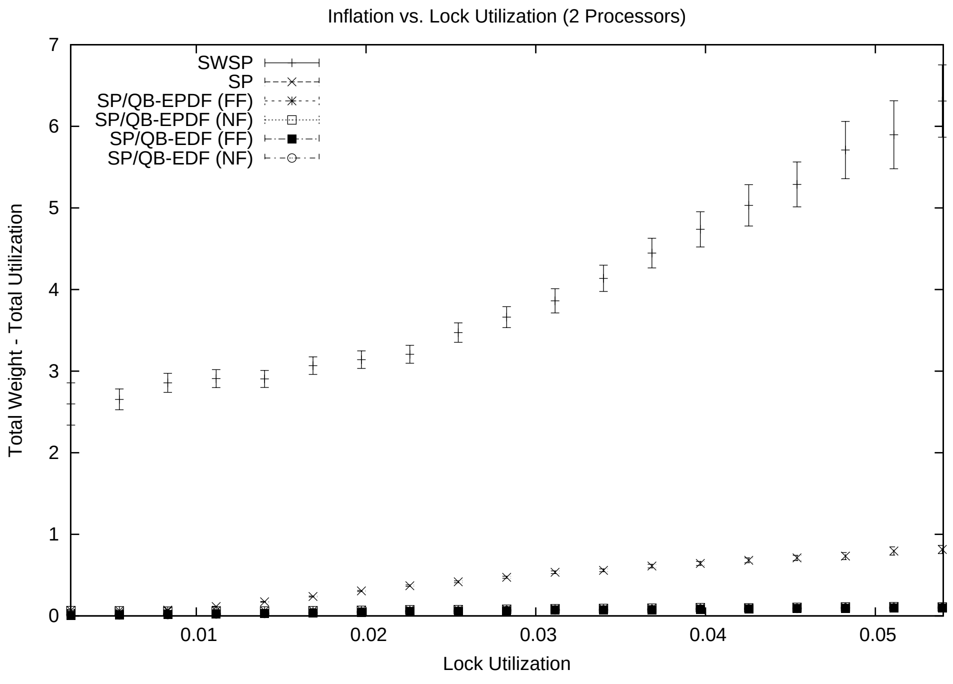
In terms of performance trends, these results resemble those of the first study. (Again, recall that the supertasking measurements vary with the utilization due to reweighting overhead, which was already discussed in Chapter 4.) The only substantial difference between these graphs and those of the first study is that the SP measurement skews toward zero as the utilization increases (compare Figures 6.37–6.40 to Figures 6.81–6.84). Since the SP measurement is the largest measurement (other than the SWSP measurement) in this study, it is the most likely cause of low sampling validity, and hence the measurement that is likely to be most affected by sample filtering. The shape of the sampling validity curves shown in Figures 6.85 and 6.86 appears to correspond to the skew in the SP measurement, which further supports this conclusion.

Impact of lock utilization. Figures 6.87, 6.88, 6.89, and 6.90 show how inflation varies with the lock utilization on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.91, 6.92, 6.93, and 6.94 show the remaining approaches. Plots of the sampling validity are shown in Figures 6.95 and 6.96.

As can be seen in Figures 6.95 and 6.96, the sampling validity is quite unusual for these graphs. The cause of the large drops in sampling validity is unclear. With respect to interpreting the results, the dips in sampling validity appear to correspond to the regions over which inflation increases most rapidly. The sampling validity suggests that the actual increase in inflation is even more rapid than the graphs suggest. However, it is also unclear why the sampling validity increases again as the lock utilization increases. Further investigation is needed to better understand this unusual behavior.

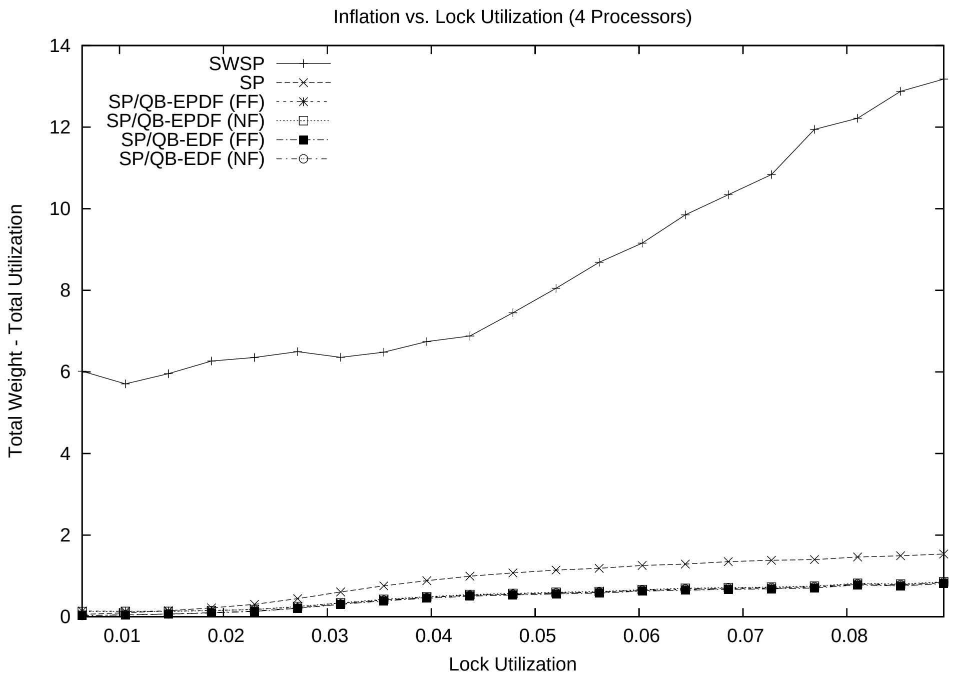


(a) Sample Means Only

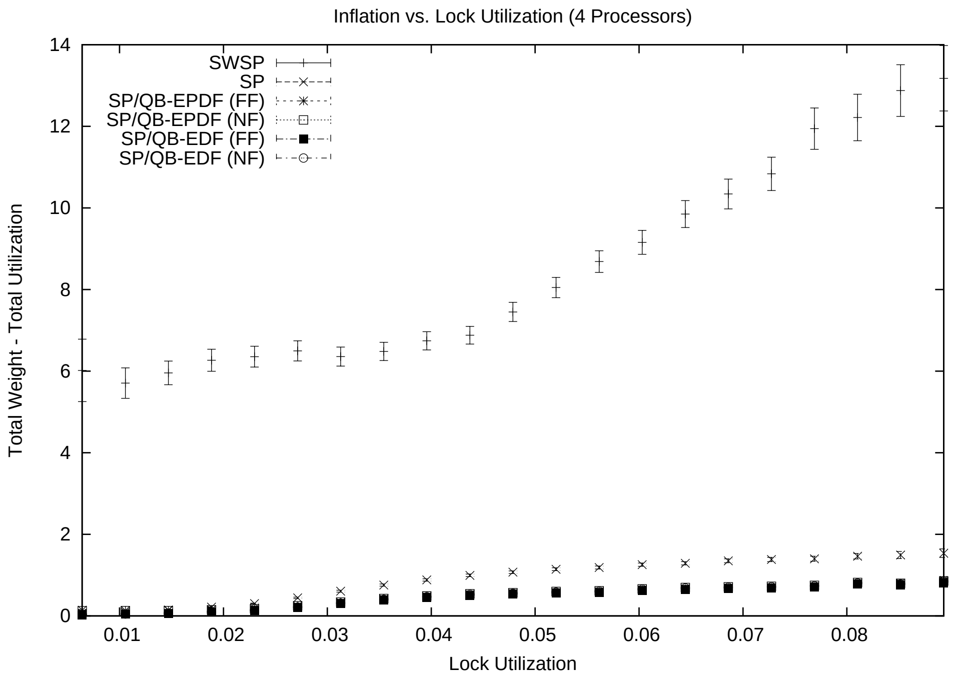


(b) 99% Confidence Interval

Figure 6.87: Plots show how inflation varies as lock utilization is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

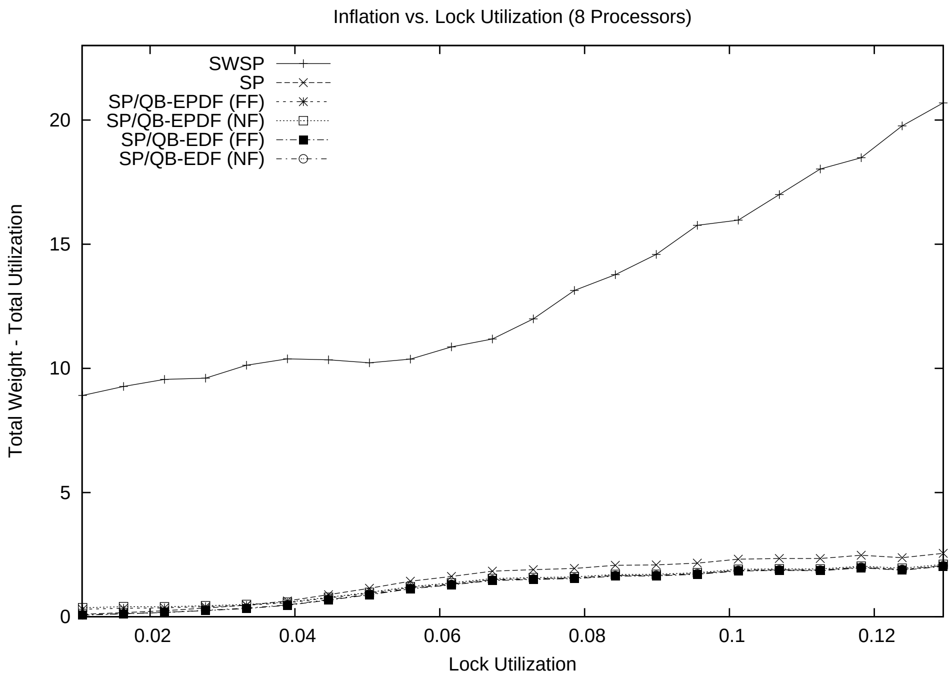


(a) Sample Means Only

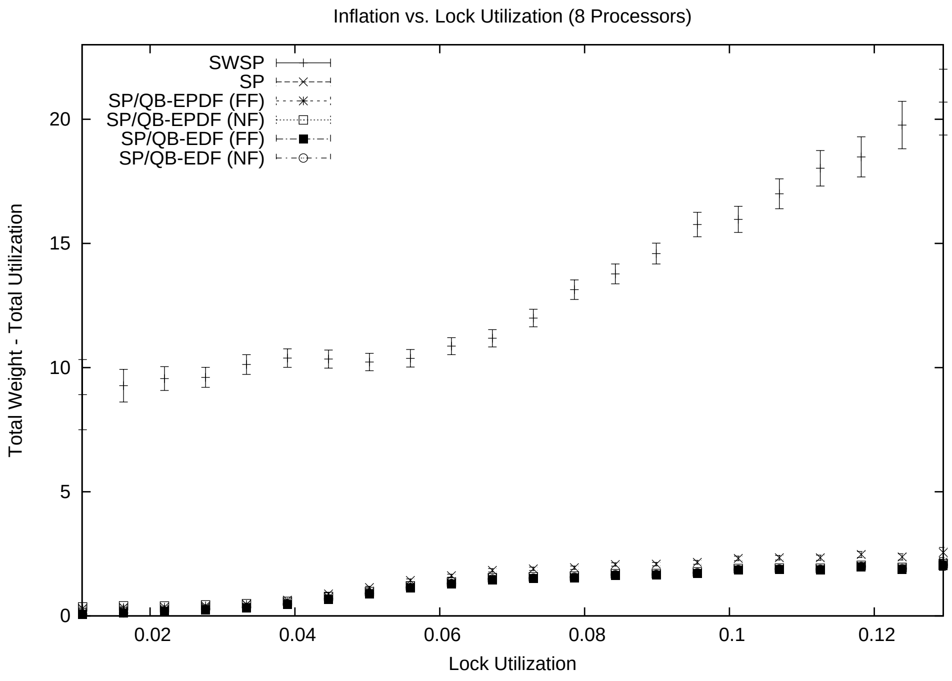


(b) 99% Confidence Interval

Figure 6.88: Plots show how inflation varies as lock utilization is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

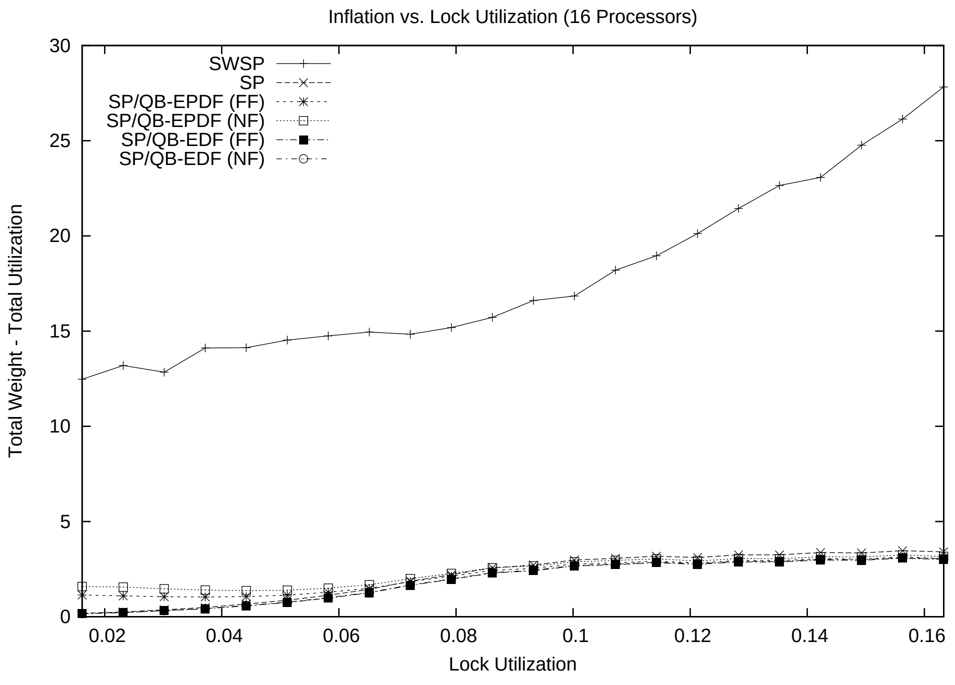


(a) Sample Means Only

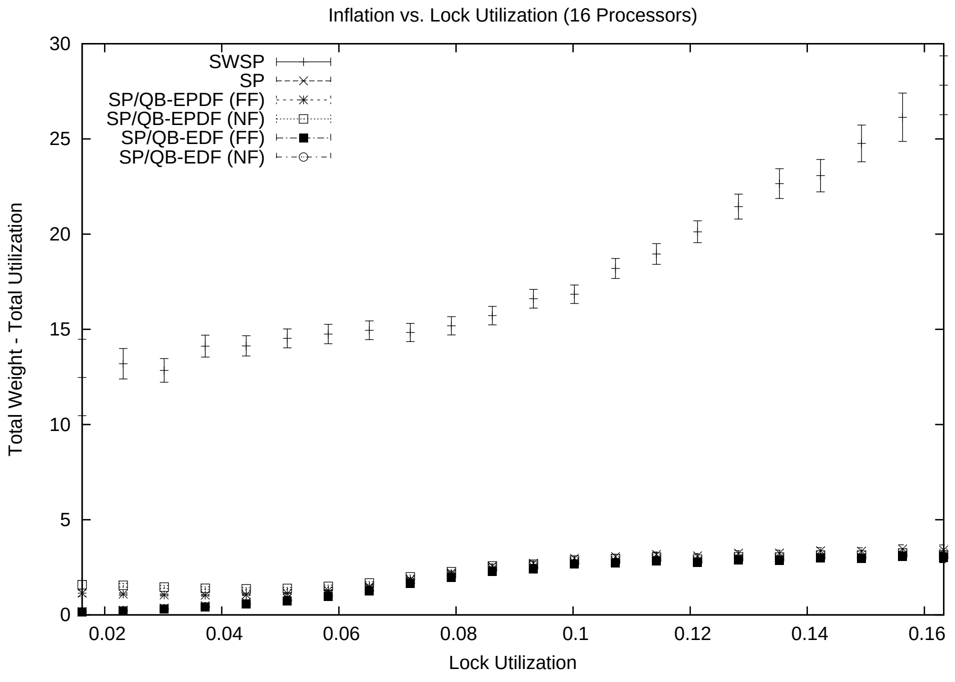


(b) 99% Confidence Interval

Figure 6.89: Plots show how inflation varies as lock utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

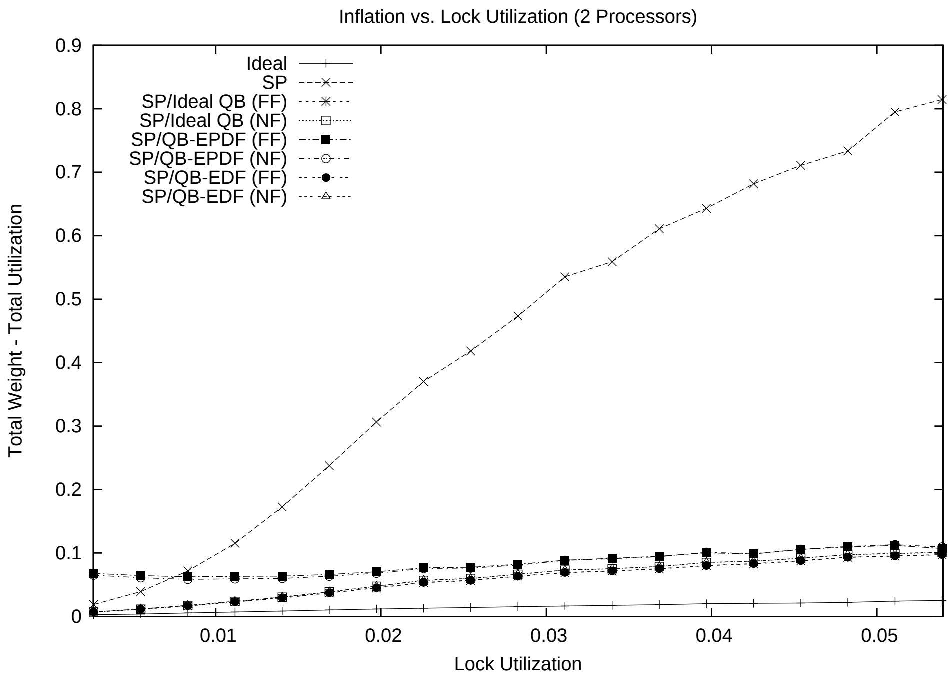


(a) Sample Means Only

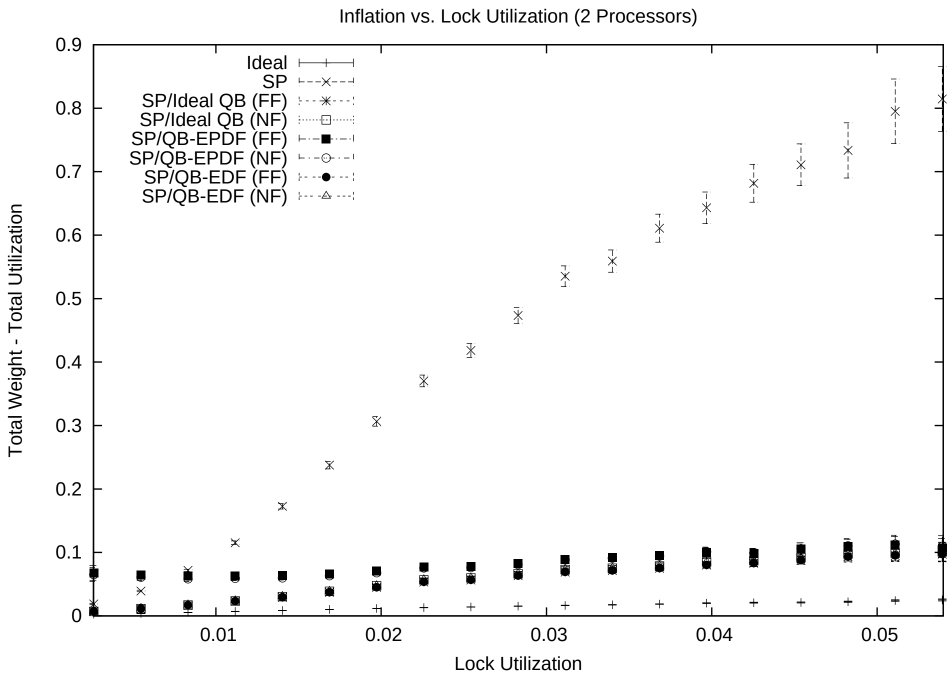


(b) 99% Confidence Interval

Figure 6.90: Plots show how inflation varies as lock utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

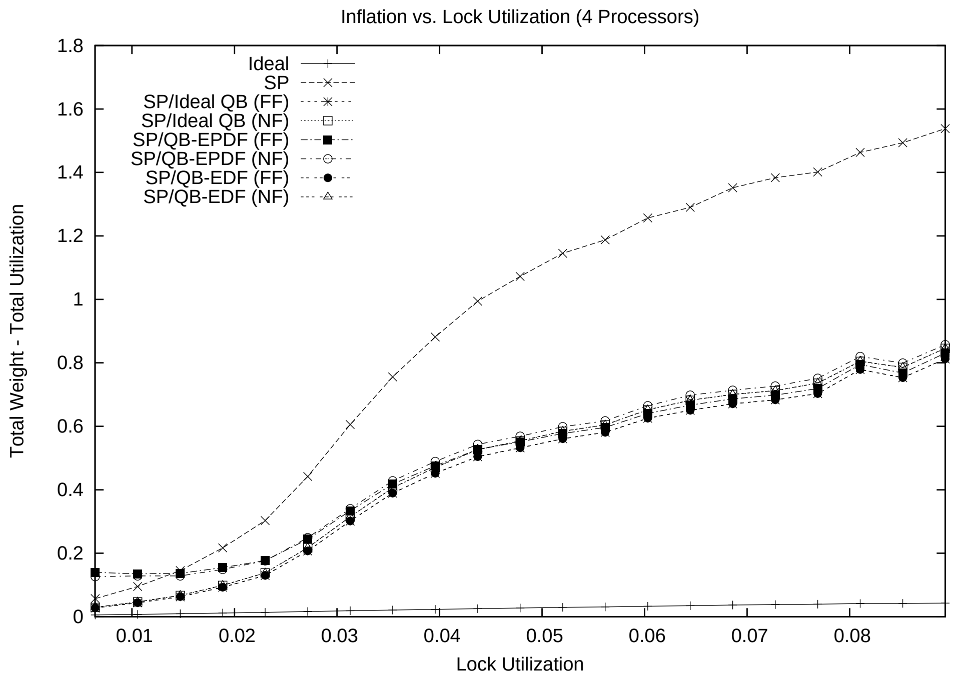


(a) Sample Means Only

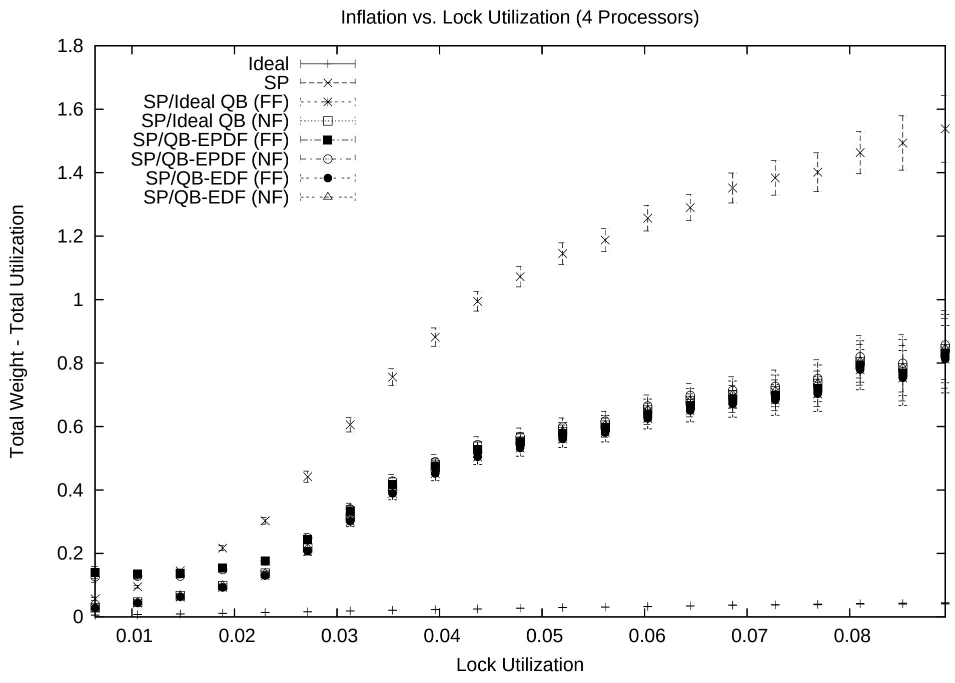


(b) 99% Confidence Interval

Figure 6.91: Plots show how inflation varies as lock utilization is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

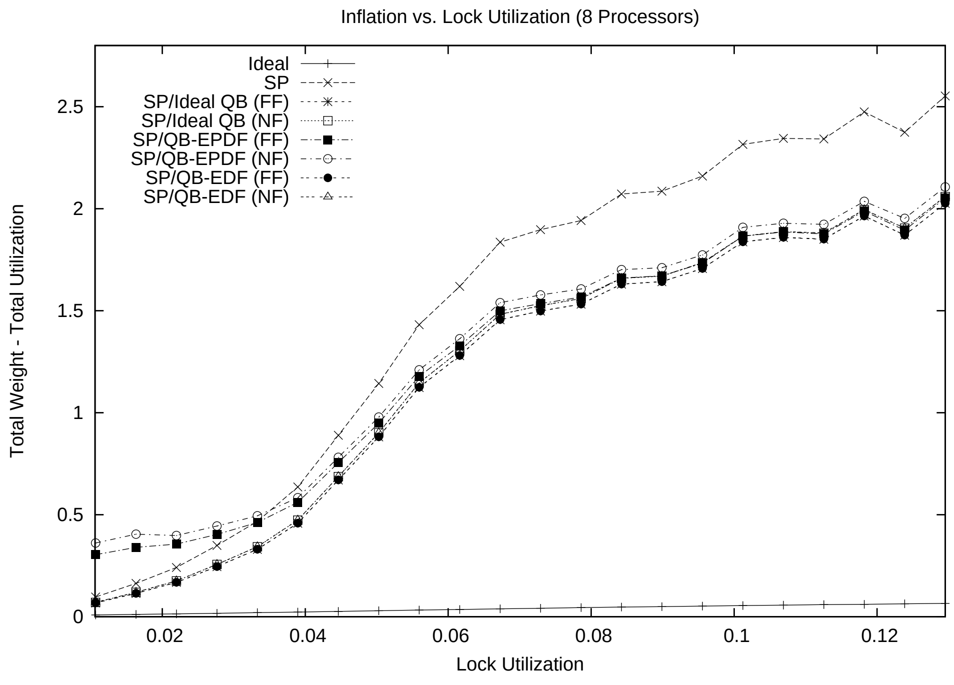


(a) Sample Means Only

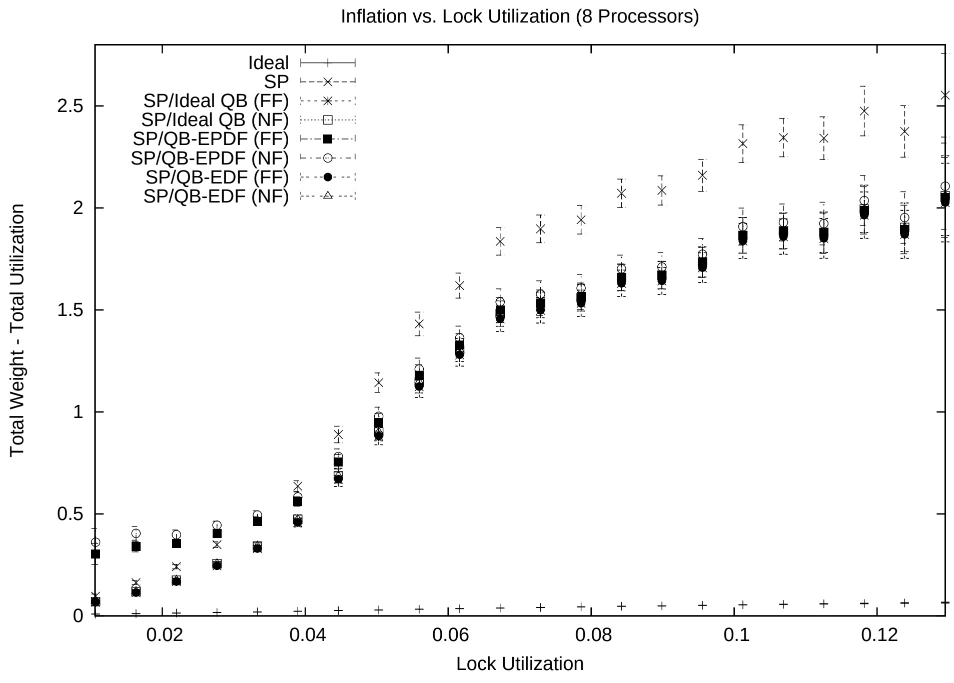


(b) 99% Confidence Interval

Figure 6.92: Plots show how inflation varies as lock utilization is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

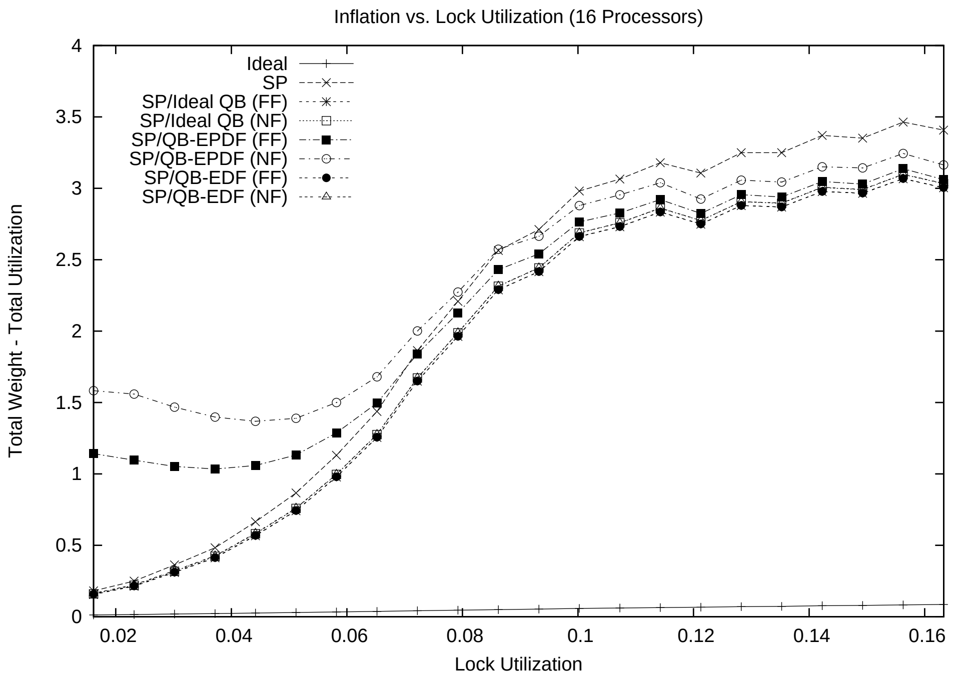


(a) Sample Means Only

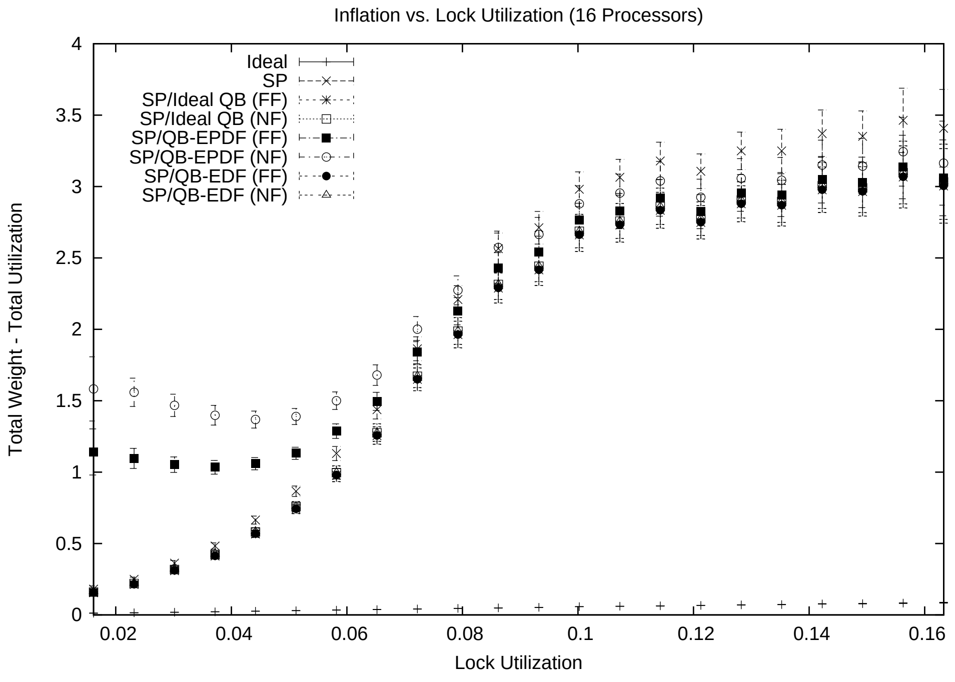


(b) 99% Confidence Interval

Figure 6.93: Plots show how inflation varies as lock utilization is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

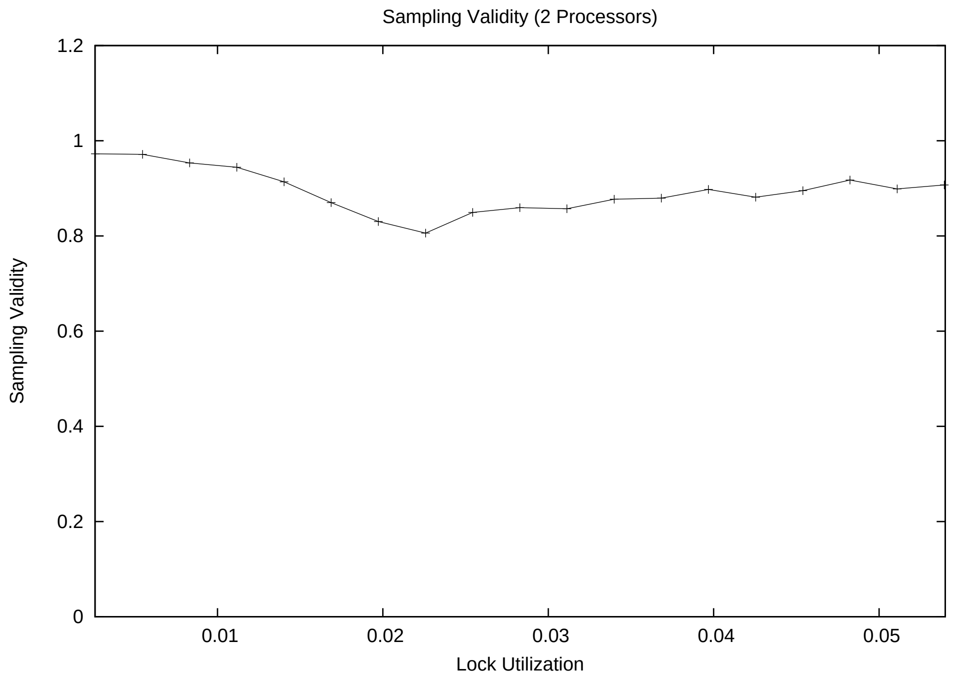


(a) Sample Means Only

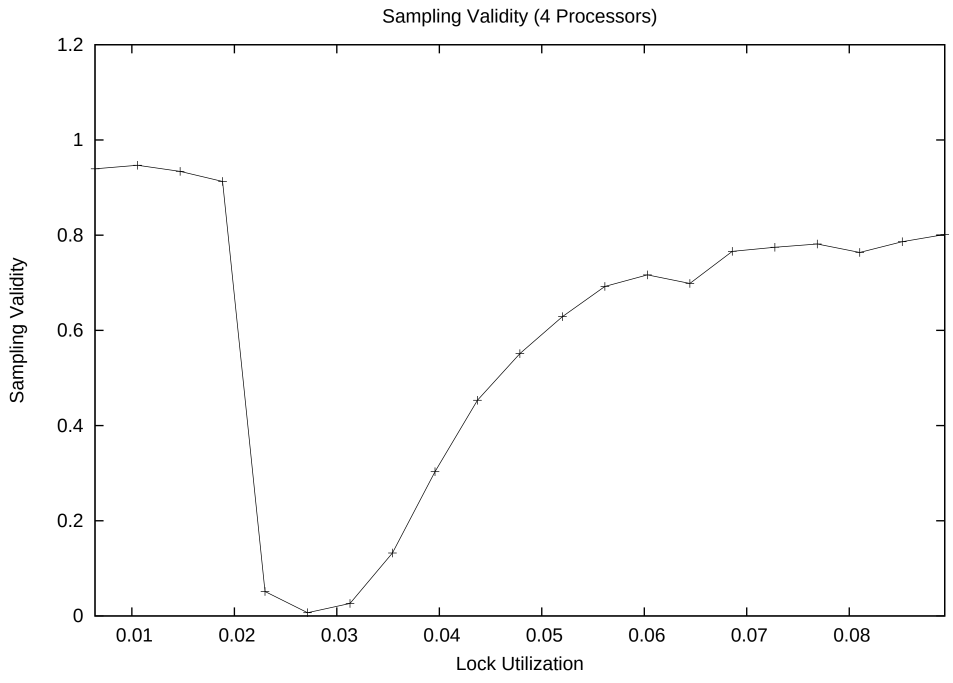


(b) 99% Confidence Interval

Figure 6.94: Plots show how inflation varies as lock utilization is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

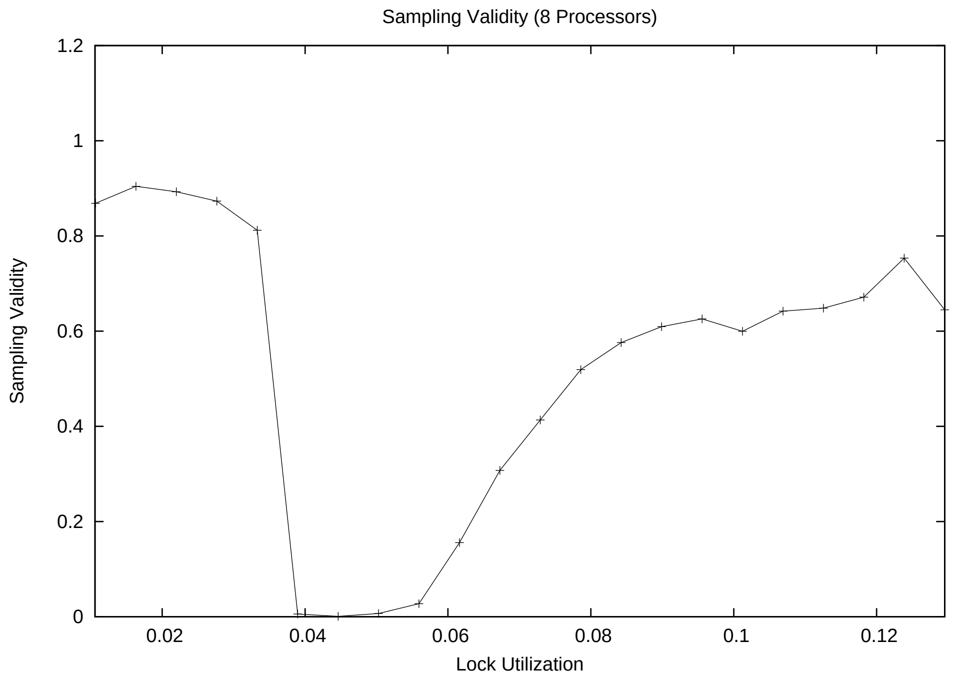


(a) $M = 2$

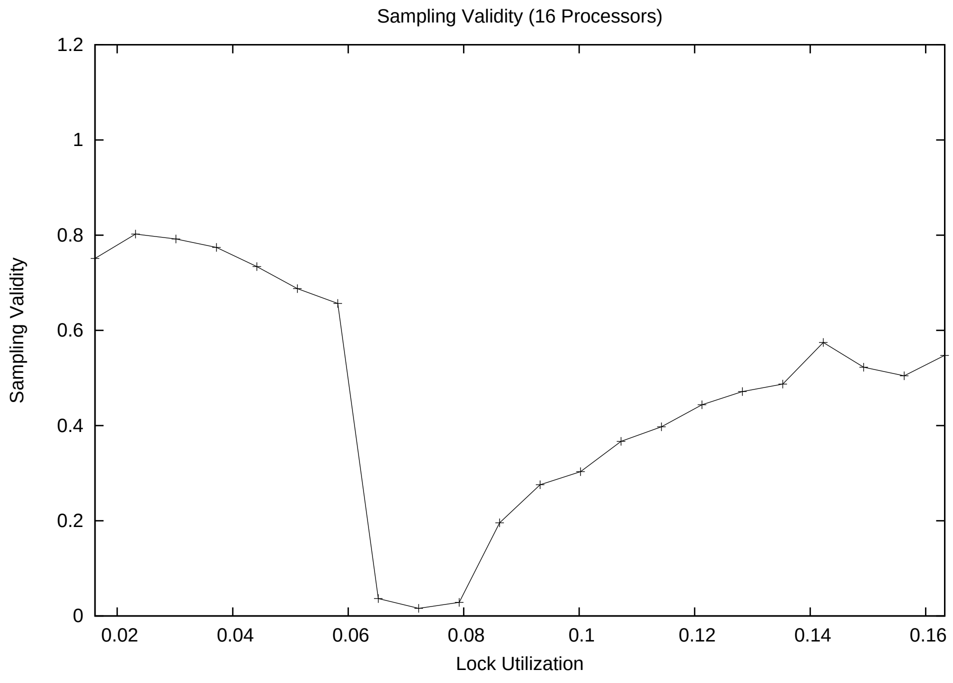


(b) $M = 4$

Figure 6.95: Plots show the sampling validity for the graphs showing inflation plotted against lock utilization when (R1) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



(b) $M = 16$

Figure 6.96: Plots show the sampling validity for the graphs showing inflation plotted against lock utilization when (R1) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

6.6.3 Study 3: Performance under (R1) as a Function of C

In this section, we present the details and results of the third study. This study measured the performance of each locking approach while varying C. As in the previous study, only task sets satisfying (R1) were considered.

Sample space. For this study, the sample space was defined as follows:

- M is in the range $2, \ldots, 16$;
- $|\tau|$ is in the range $5 \cdot \log_2 M, \ldots, 15 \cdot \log_2 M$;
- $\tau.u$ is in the range $0.2 \cdot M, \ldots, 0.5 \cdot M$;
- $|\Gamma|$ is in the range $4 \cdot \log_2 M, \ldots, 16 \cdot \log_2 M$;
- $T.p$ is in the range $50, \ldots, 2000$;
- C is in the range $0.1, \ldots, 0.9$.

Because this study allows for more contention and hence more inflation, the sample space is again more restricted than in the previous study. Specifically, the upper bound of $|\tau|$ and $\tau.u$ have been increased. In addition, the $|\Gamma|$ range has been shifted to larger counts in order to avoid cases in which only one or two locks are shared by a large number of tasks.

Sampling. Samples were collected as follows:

- M was set to each value in the set $\{\ 2^x \mid 1 \leq x \leq 4\ \}$;
- $|\tau|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 5 \leq x \leq 15\ \}$;
- $\tau.u$ was set to each value in the set $\{\ 0.025x \cdot M \mid 8 \leq x \leq 20\ \}$;

- $|\Gamma|$ was set to each value in the set $\{\ x \cdot \log_2 M \mid 4 \leq x \leq 16\ \}$;
- C was set to each value in the set $\{\ 0.05x \mid 2 \leq x \leq 18\ \}$.

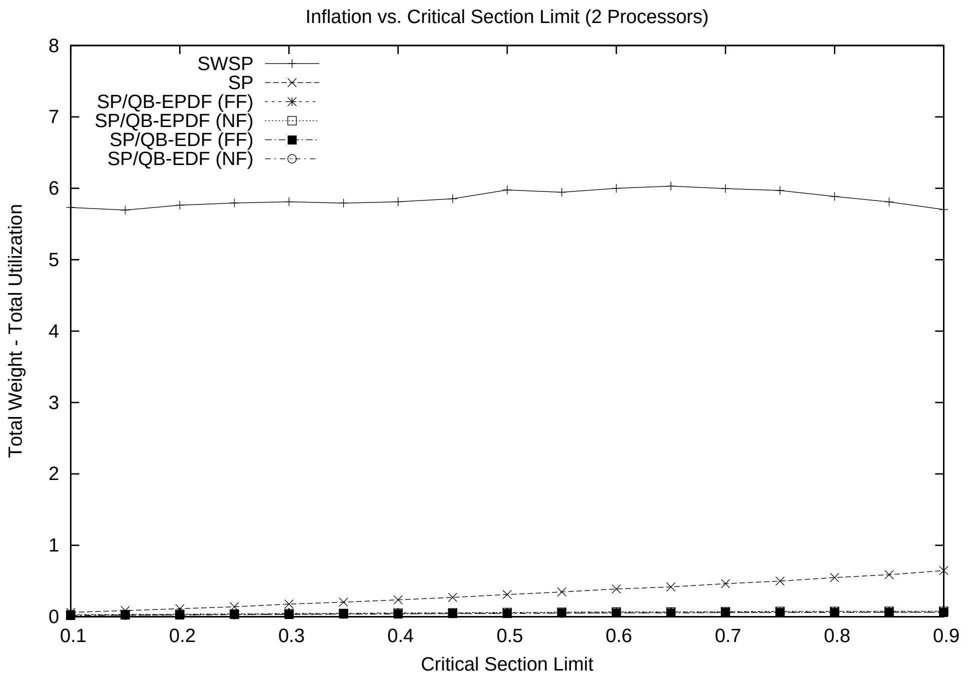
For each combination of the above parameter assignments, one valid task set was generated and evaluated. The sampling validity criteria is identical to that used by the previous study.

Measurement. This study is based on the same measurements as the previous study.

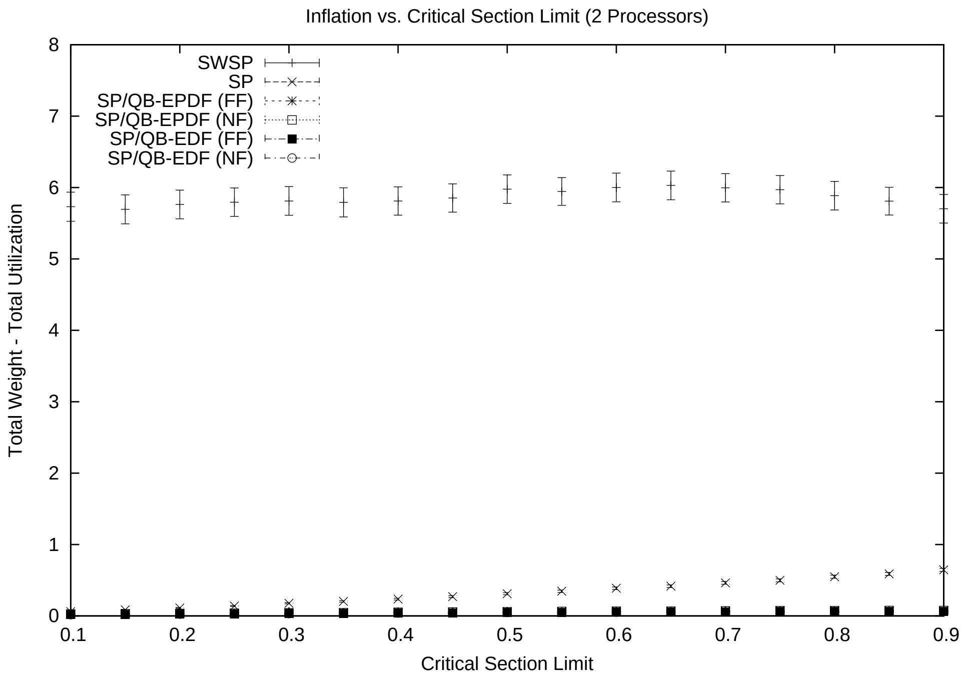
Results. Figures 6.97, 6.98, 6.99, and 6.100 show how inflation varied with C on two, four, eight, and sixteen processors, respectively, at the scale of the SWSP's inflation. Figures 6.101, 6.102, 6.103, and 6.104 show the remaining approaches. Sampling validity is shown in Figures 6.105 and 6.106.

As shown in Figures 6.105 and 6.106, sampling validity is relatively high until around $C = 0.5$, at which point it begins dropping off. This drop off appears to produce a slight skew toward zero in the SWSP measurement shown in Figures 6.97–6.100 over the C range $[0.7, 0.9]$. Since mean inflation should always increase as the critical-section durations increase, any decrease in the mean inflation under any approach must be caused by sample filtering. The other measurements do not appear to be affected as strongly as the SWSP by the decreasing sampling validity. However, it is likely that these measurements are also skewed toward zero since filtering is primarily controlled by the zone-based approaches. Hence, it is likely that these the mean inflation increases at a faster rate than is suggested by Figures 6.101–6.104.

Due to the obvious skew in the SWSP curve, it appears that the SWSP provides the best scalability with C, though its mean inflation is very high. Due to this latter fact, the SWSP will likely perform better than the alternative approaches only in special cases. Supertasking appears to improve scalability when M is low, though performance converges

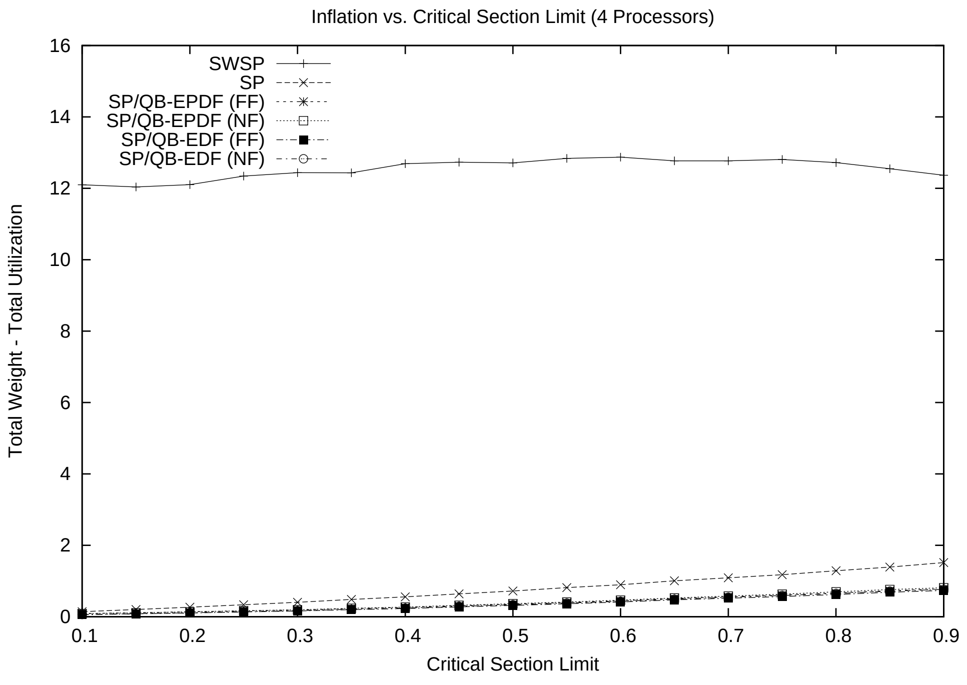


(a) Sample Means Only

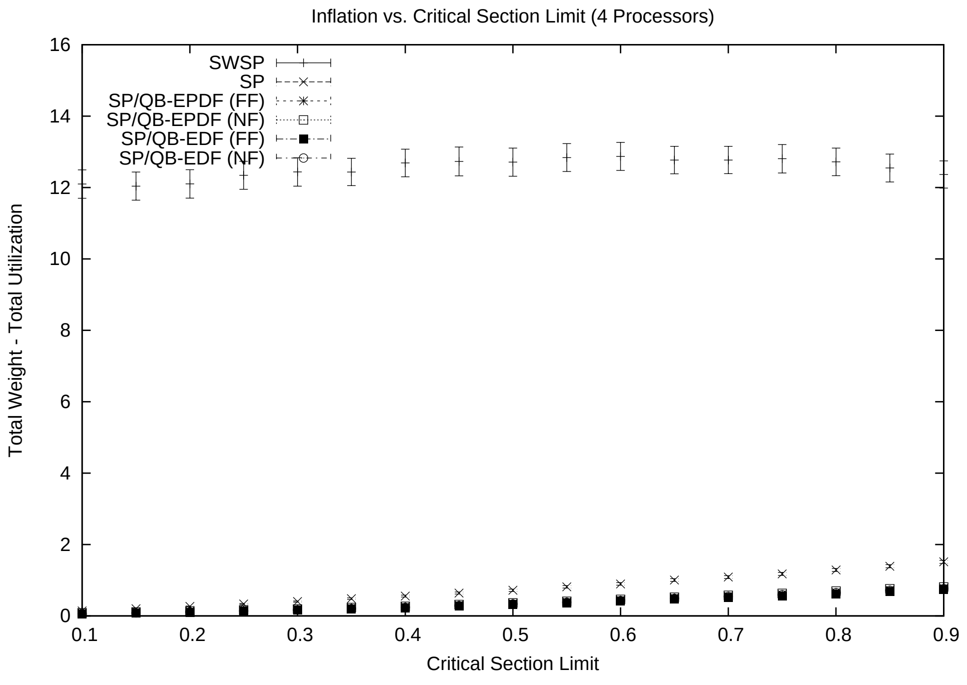


(b) 99% Confidence Interval

Figure 6.97: Plots show how inflation varies as C is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

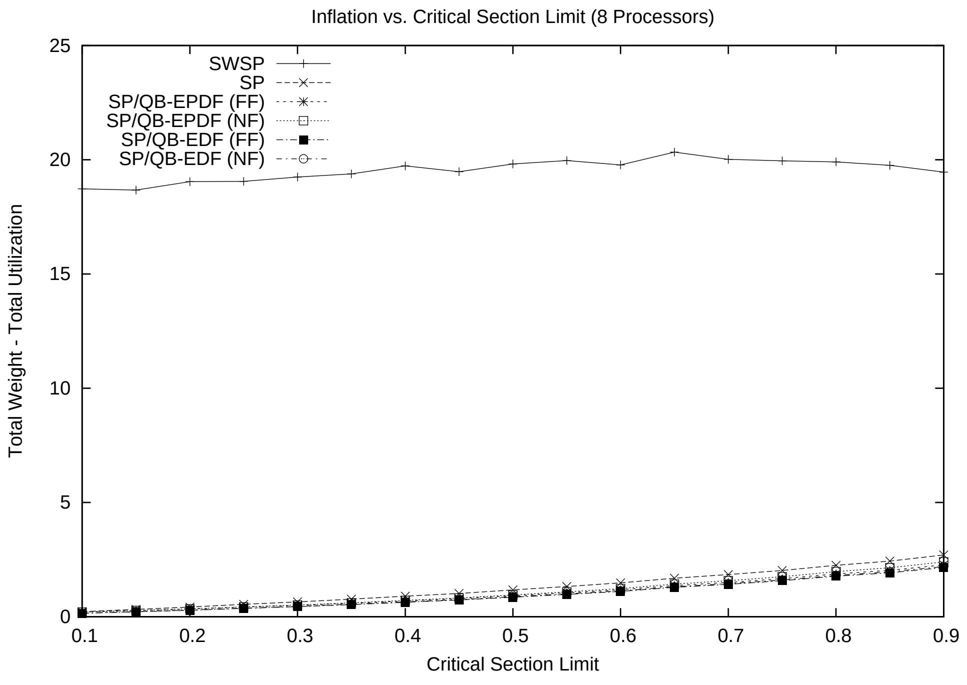


(a) Sample Means Only

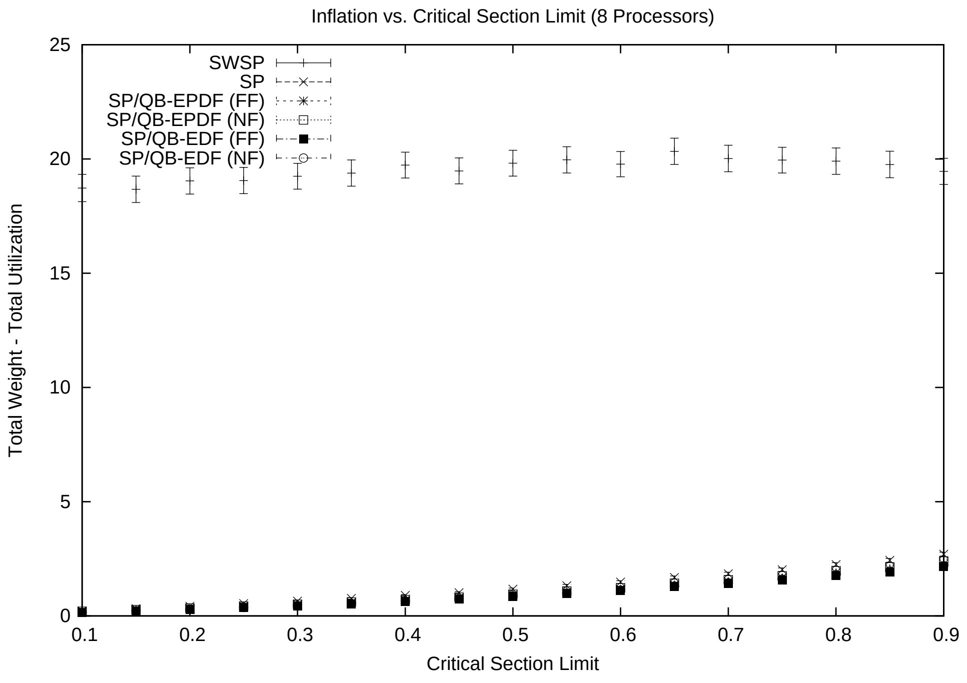


(b) 99% Confidence Interval

Figure 6.98: Plots show how inflation varies as C is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

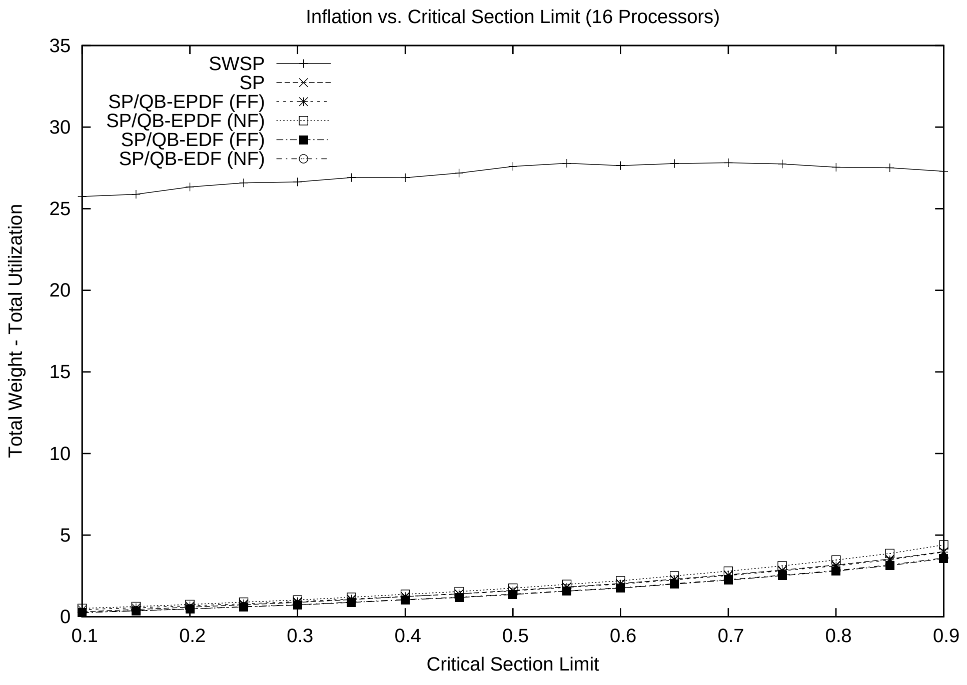


(a) Sample Means Only

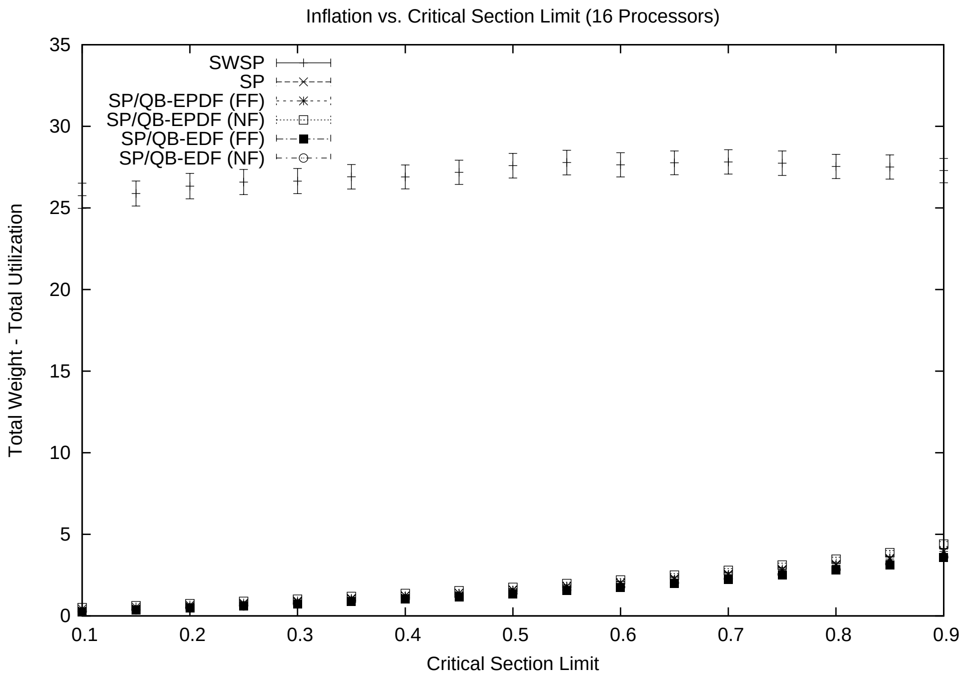


(b) 99% Confidence Interval

Figure 6.99: Plots show how inflation varies as C is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

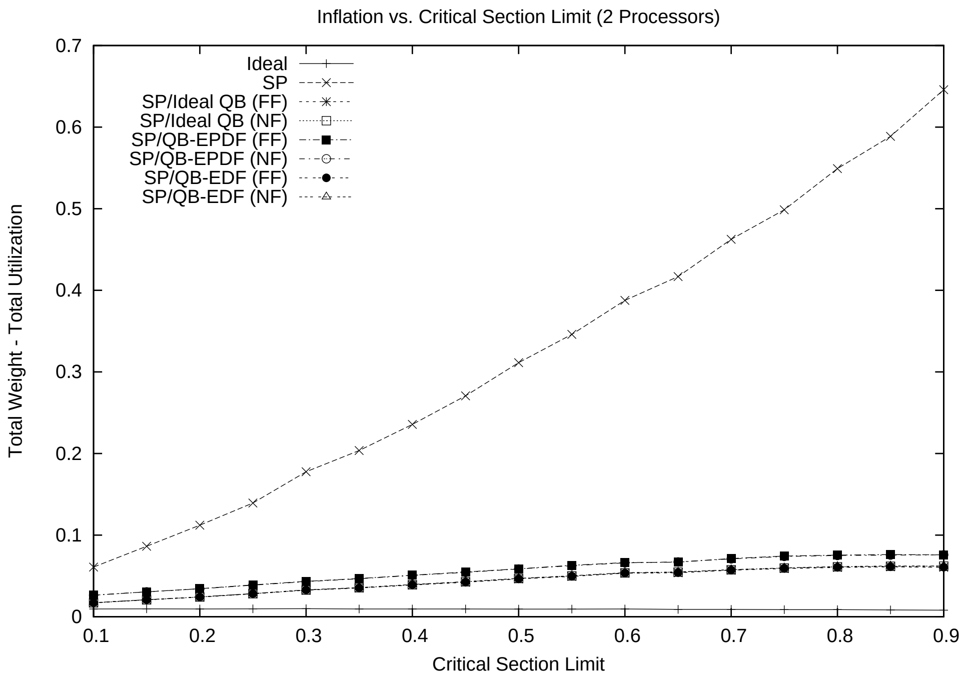


(a) Sample Means Only

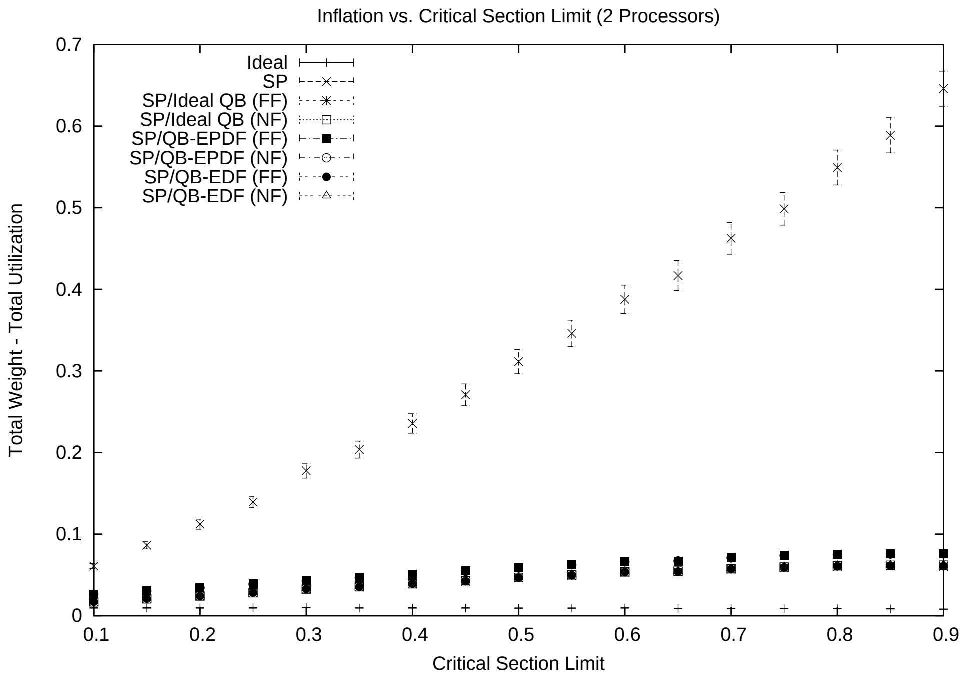


(b) 99% Confidence Interval

Figure 6.100: Plots show how inflation varies as C is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

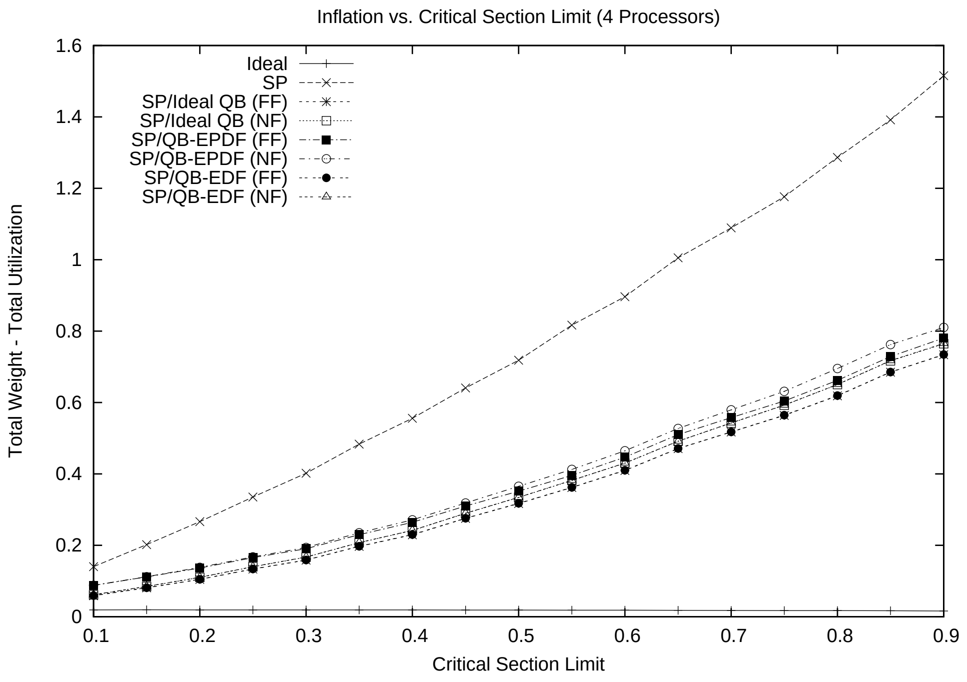


(a) Sample Means Only

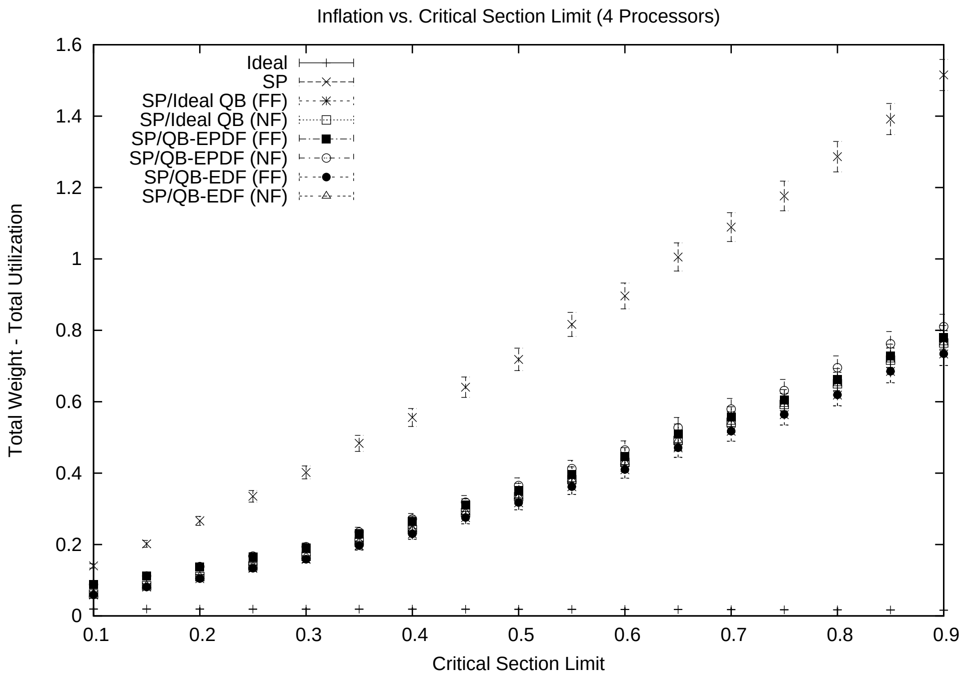


(b) 99% Confidence Interval

Figure 6.101: Plots show how inflation varies as C is increased when using locking synchronization on two processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

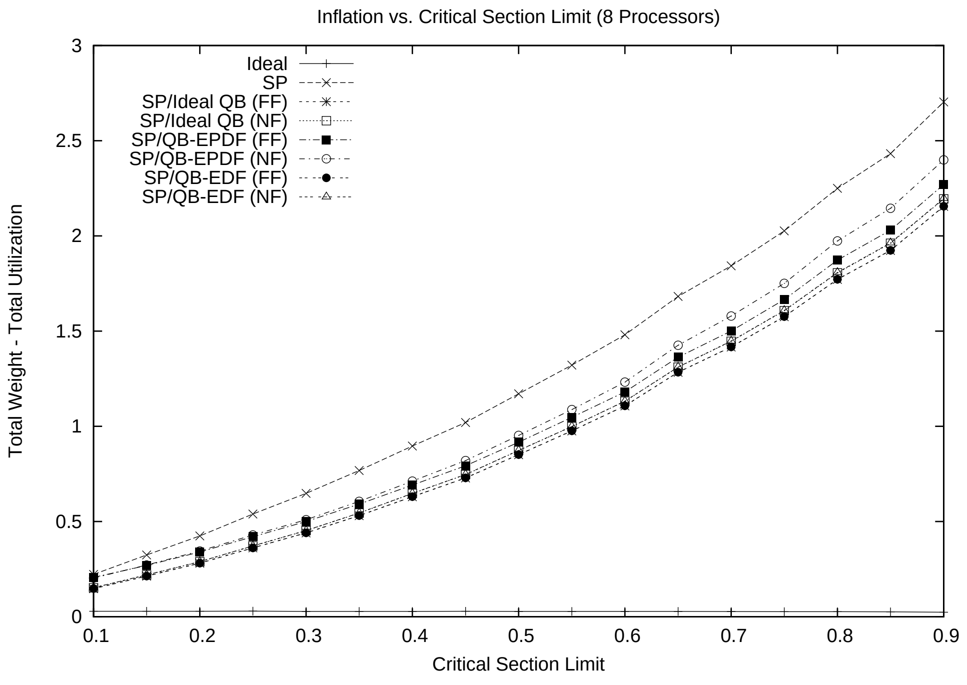


(a) Sample Means Only

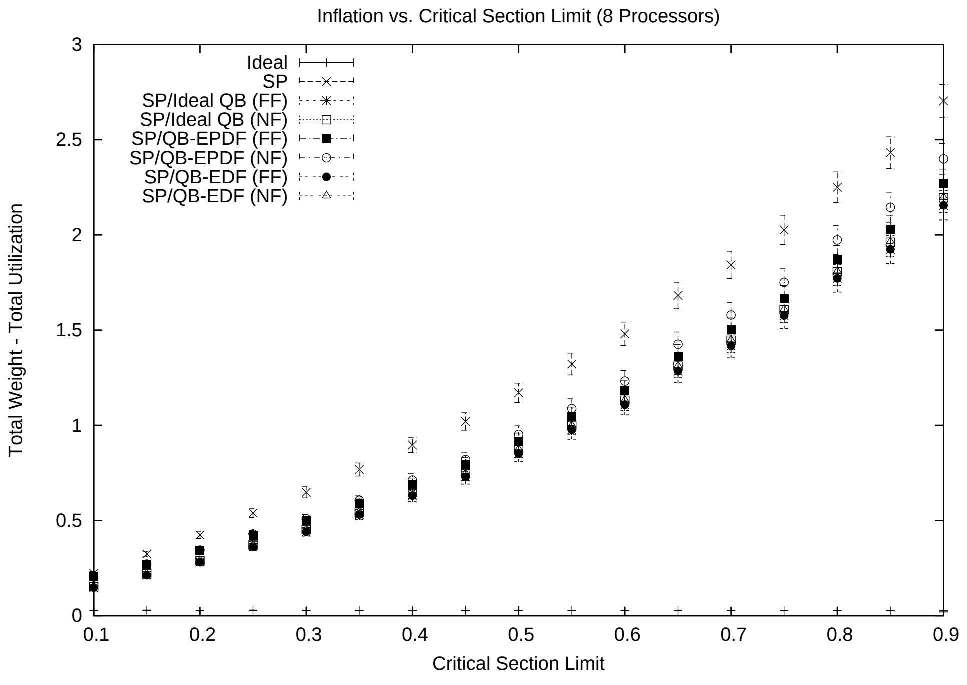


(b) 99% Confidence Interval

Figure 6.102: Plots show how inflation varies as C is increased when using locking synchronization on four processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

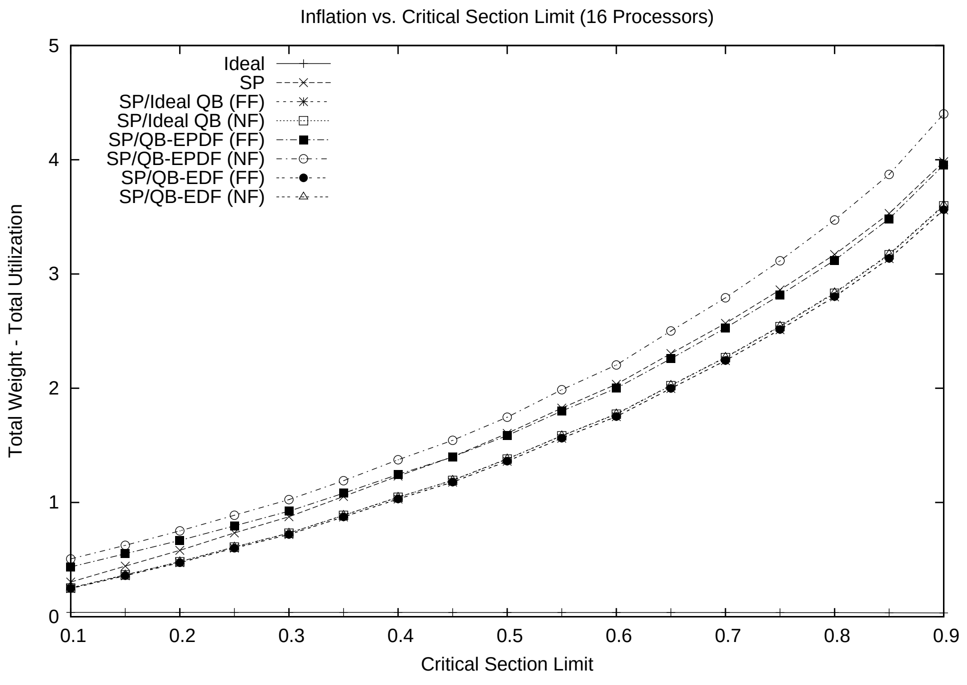


(a) Sample Means Only

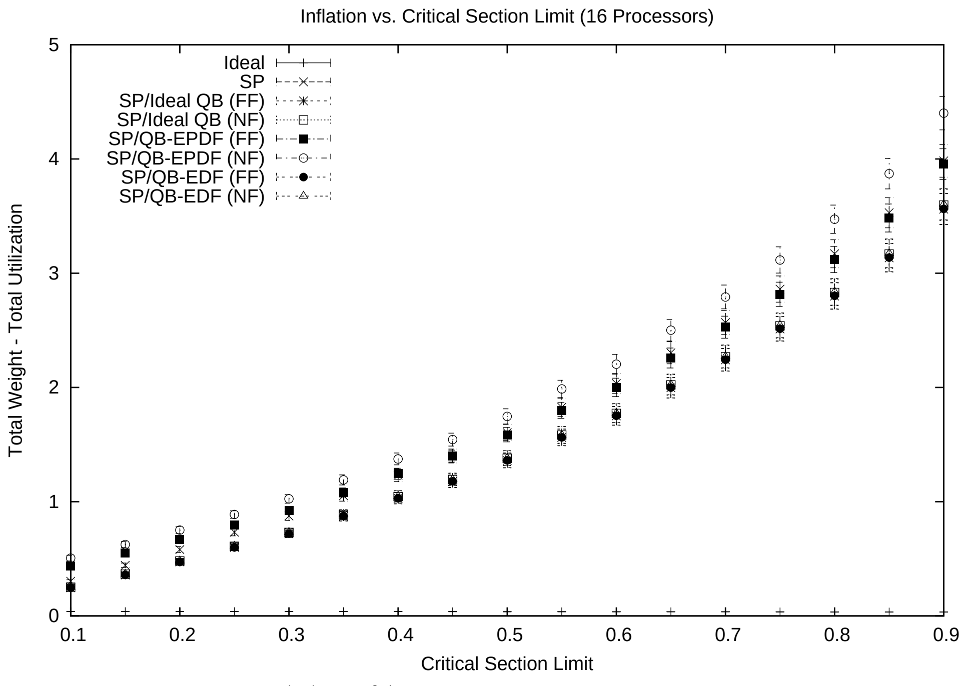


(b) 99% Confidence Interval

Figure 6.103: Plots show how inflation varies as C is increased when using locking synchronization on eight processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

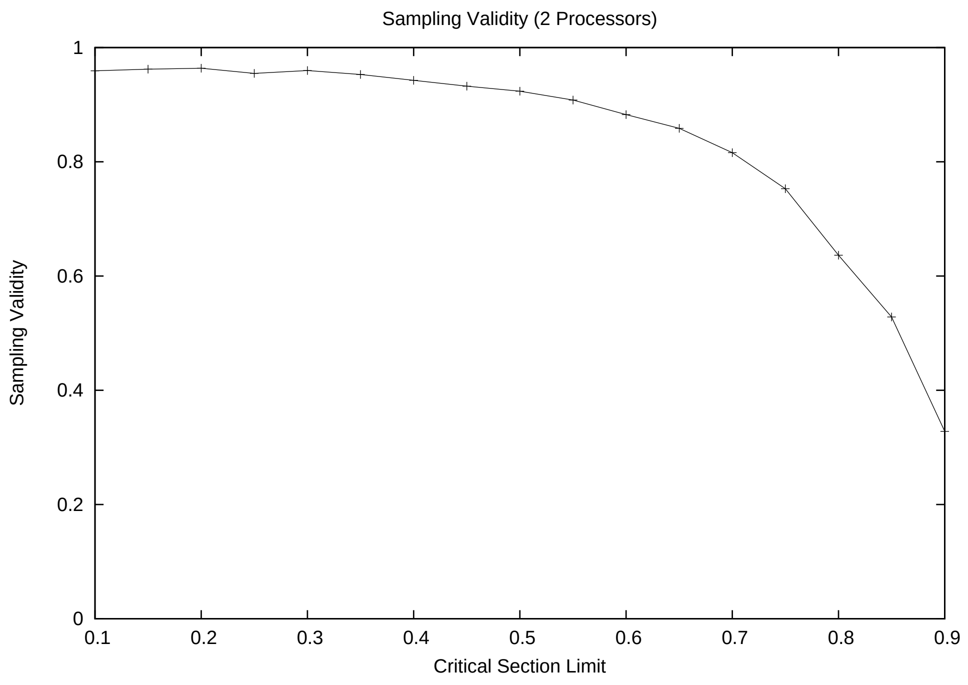


(a) Sample Means Only

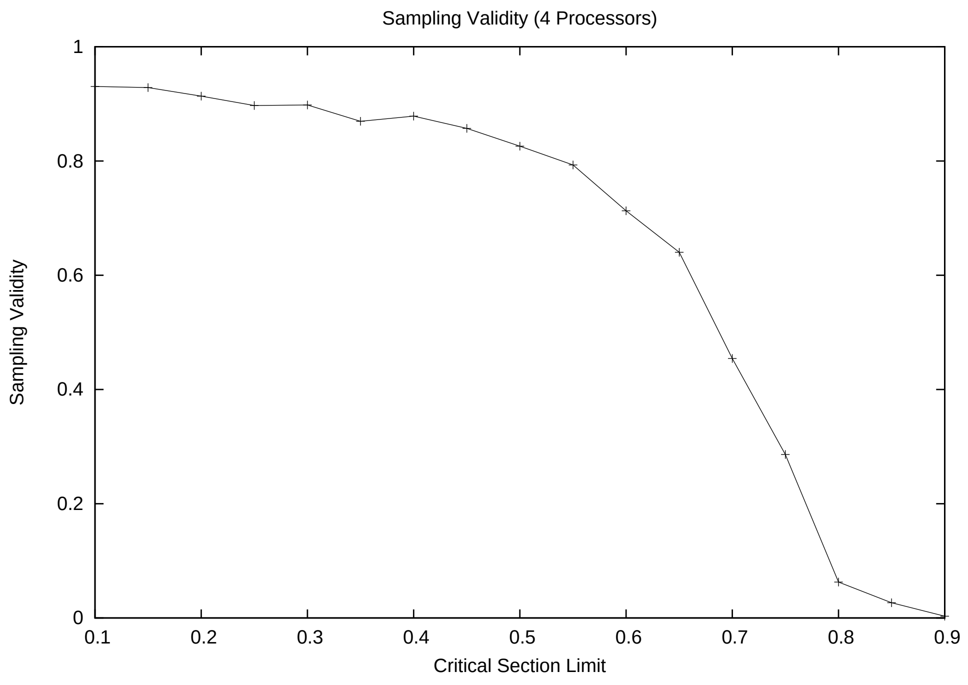


(b) 99% Confidence Interval

Figure 6.104: Plots show how inflation varies as C is increased when using locking synchronization on sixteen processors and when all locks satisfy (R1). The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

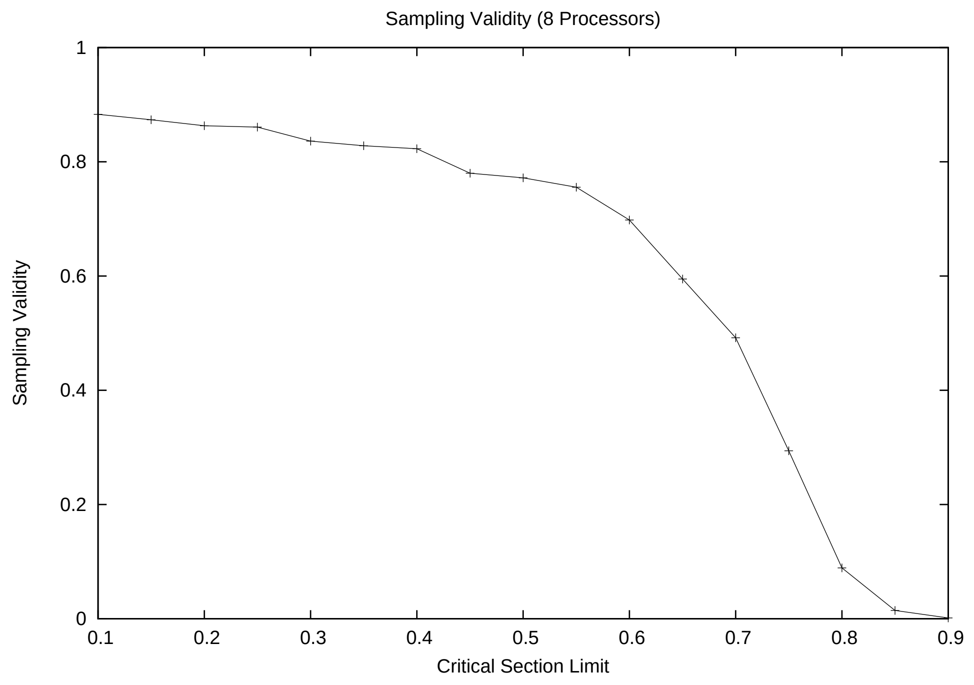


(a) $M = 2$

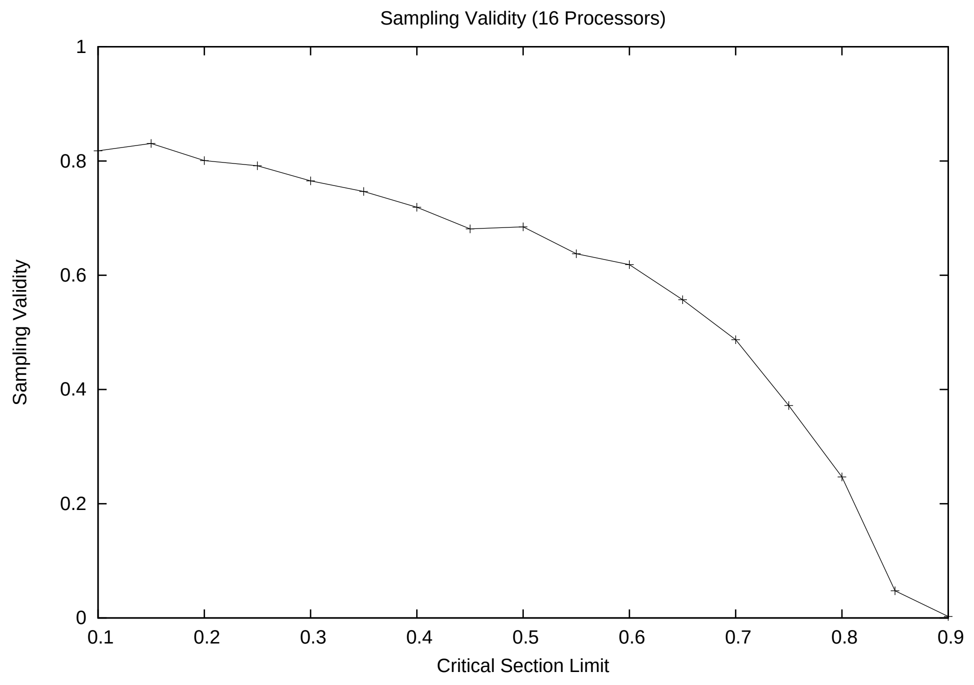


(b) $M = 4$

Figure 6.105: Plots show the sampling validity for the graphs showing inflation plotted against C when (R1) holds. The figure shows **(a)** the $M = 2$ and **(b)** the $M = 4$ cases.



(a) $M = 8$



(b) $M = 16$

Figure 6.106: Plots show the sampling validity for the graphs showing inflation plotted against C when (R1) holds. The figure shows **(a)** the $M = 8$ and **(b)** the $M = 16$ cases.

to that of the SP without supertasking for higher M. However, as in the previous study, this apparent convergence may be misleading since skew in the SP measurement due to low sampling validity could produce the same relationship. Again, this apparent convergence warrants further study.

6.6.4 Study 4: Locking within Supertasks

In this section, we present the results of the fourth and final locking study. This study focused on the special case in which a lock is shared only by component tasks of a common supertask. In such cases, one alternative is to use fully preemptive supertasking along with a uniprocessor locking protocol, such as the PCP [54]. In addition to the efficiency of uniprocessor techniques, this approach provides an advantage in that mapping overhead is avoided. To evaluate the performance of this approach relative to those proposed earlier in this chapter, we repeated the third study, but modified the sample space so that the PCP could be used.[19]

Sample space. For this study, the sample space was defined as follows:

- $|\tau|$ is in the range $4, \ldots, 12$;
- $\tau.u$ is in the range $0.05, \ldots, 0.5$;
- $|\Gamma|$ is in the range $1, \ldots, 10$;
- $T.p$ is in the range $50, \ldots, 2000$;
- C is in the range $0.05, \ldots, 0.95$.

Sampling. Samples were collected as follows:

[19]We used an exhaustive search when calculating the worst-case blocking under the PCP. Hence, the results of this study reflect the minimum inflation produced by the PCP.

- $|\tau|$ was set to each value in the range $4, \ldots, 12$;
- $\tau.u$ was set to each value in the set $\{\ 0.05x \mid 1 \le x \le 10\ \}$;
- $|\Gamma|$ was set to each value in the range $1, \ldots, 10$;
- C was set to each value in the set $\{\ 0.05x \mid 1 \le x \le 19\ \}$.

For each combination of the above parameter assignments, two valid *component* task sets was generated and evaluated.

Measurement. The following measurements were taken during the experimental runs.

SWSP: the total weight of all tasks when all locks use the SWSP and supertasks are not used; this measurement reflects mapping and locking overhead.

Zone Protocol (ZP): the total weight of all tasks when all locks use a zone protocol[20] and supertasks are not used; this measurement reflects mapping and locking overhead.

ZP/Ideal QB: the sum of the ideal weights of all tasks (a baseline measurement); this measurement reflects mapping and locking overhead.

ZP/QB-EPDF: the weight of the supertask when using QB-EPDF supertasking along with a zone protocol; this measurement reflects mapping, reweighting, and locking overhead.

ZP/QB-EDF: the weight of the supertask when using QB-EDF supertasking along with a zone protocol; this measurement reflects mapping, reweighting, and locking overhead.

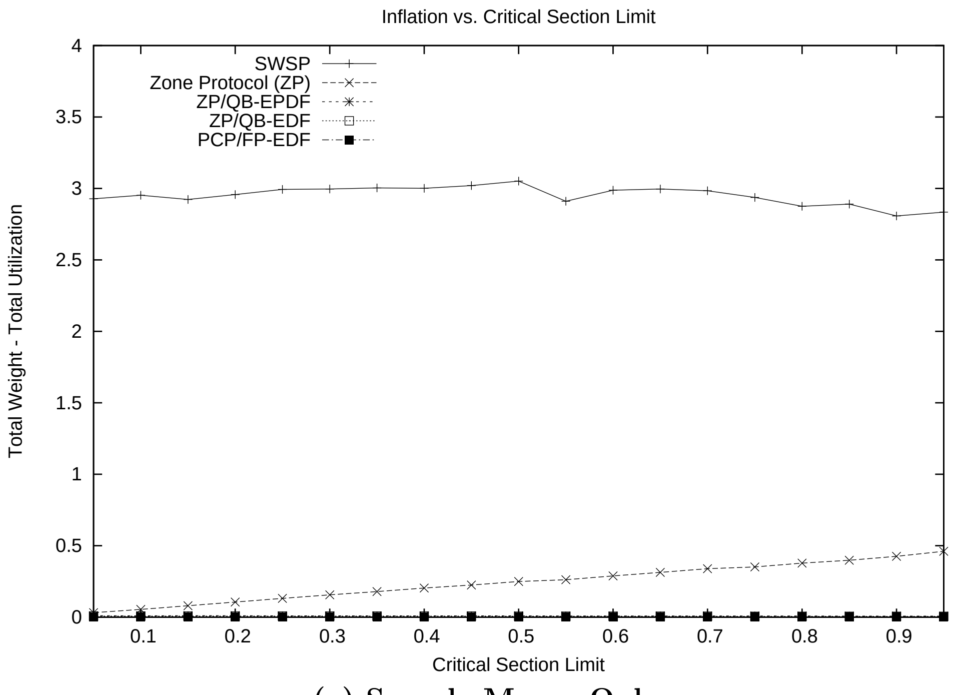
PCP/FP-EDF: the weight of the supertask when using FB-EDF supertasking along with the PCP; this measurement reflects reweighting and locking overhead.

[20]Active blocking cannot occur when all tasks reside in a common supertask. As a result, both the RP and SP produce identical results.

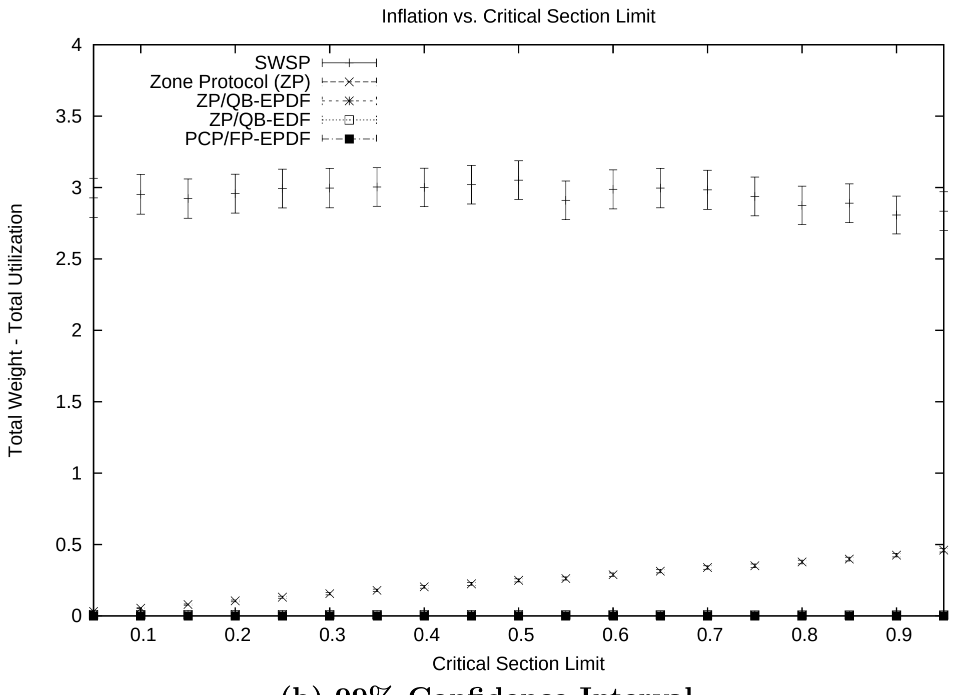
These measurements differ from those in previous studies in two ways. First, both the SP and the RP experience the same worst-case blocking in this special case. The ZP measurement reflects the inflation of both the SP and the RP. Second, only individual component task sets are tested in this study. Hence, there is no need to assign tasks to supertasks. To ensure termination when reweighting, $\mathcal{S}.w_\phi + 10^{-5}$ was passed as $w_{\min}$ when invoking `Reweight`. A smaller value of $w_{\min}$ was used for this study because the PCP/FP-EDF measurements were expected to be very small.

Results. Figures 6.107 and 6.108 show the results of this study. Figure 6.107 shows the results on the scale of the SWSP's inflation, while Figure 6.108 shows only the supertasking approaches. (The "ZP/Ideal QB" and "ZP/QB-EDF" curves are virtually identical in Figure 6.108.) Sampling validity is shown in Figure 6.109.

As shown, the PCP/FP-EDF approach far outperforms the other approaches. Indeed, the inflation under this approach is actually even lower than this study suggests. Due to our choice of parameters when invoking `Reweight`, inflation must be at least 10^{-5} for each sample. The PCP/FP-EDF samples consistently reported inflations around 10^{-5}, which suggests that the true inflation is likely lower. (Of course, inflation on this scale is negligible. Hence, there is little point in trying to lower it further.) These results clearly indicate that the PCP/FP-EDF approach is the best choice of the approaches that were tested. Again, Figure 6.109 and the fact that some means actually decrease with increasing C suggest that the samples of the other approaches are increasingly skewed toward zero after the $C = 0.3$ point. This skew is clearly visible in the ZP/QB-EPDF and ZP/QB-EDF curves in Figure 6.108. Finally, although the PCP/FP-EDF approach is clearly the favored approach, some samples (less than 0.1%) *did* favor the use of the zone-based protocols in conjunction with quantum-based

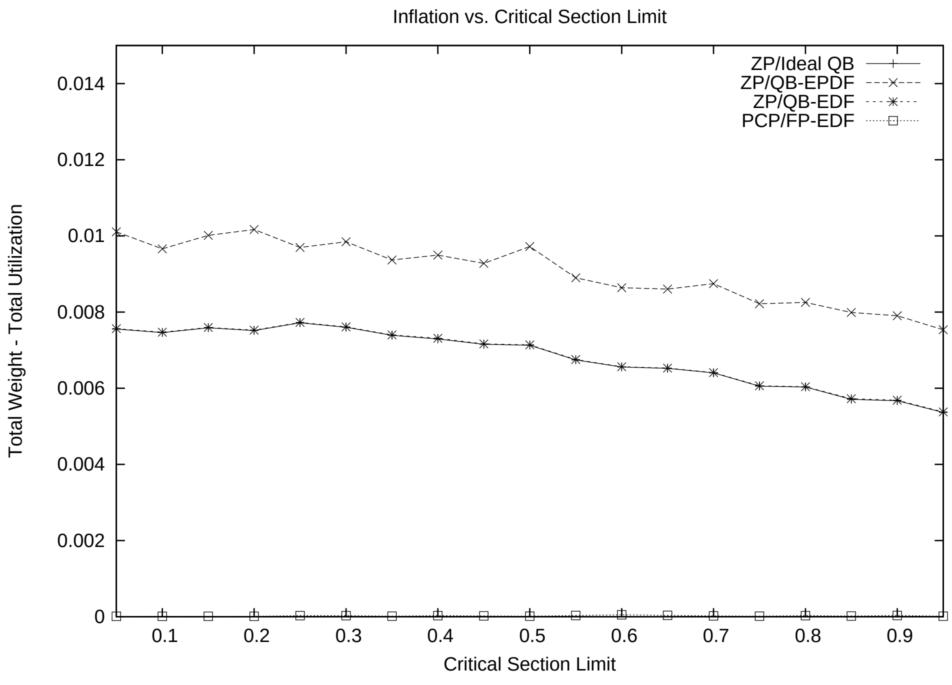


(a) Sample Means Only

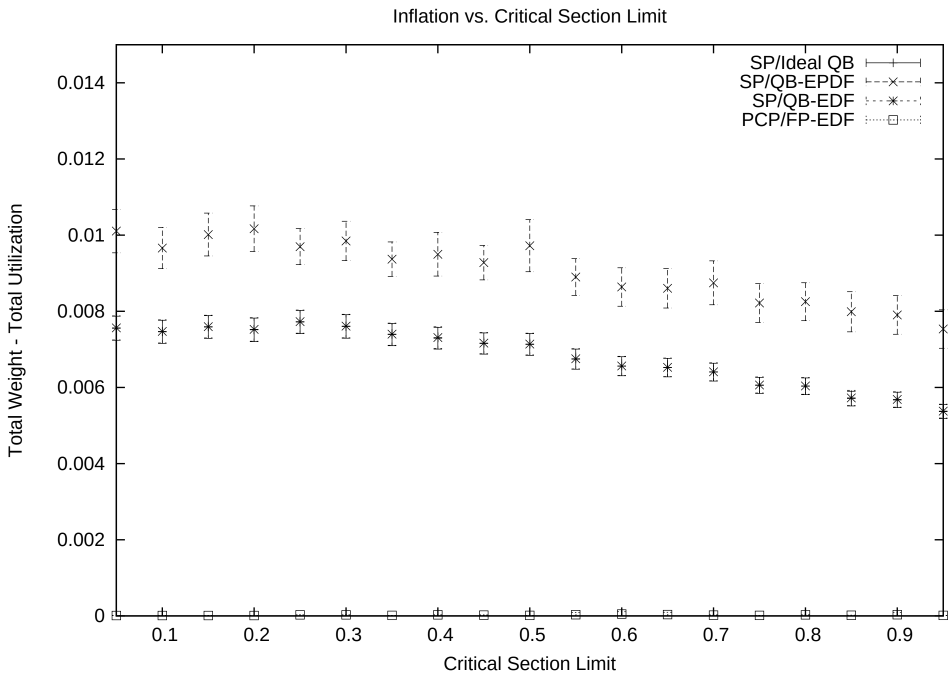


(b) 99% Confidence Interval

Figure 6.107: Plots show how inflation varies as C is increased when using locking synchronization within component task sets. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each mean.



(a) Sample Means Only



(b) 99% Confidence Interval

Figure 6.108: Plots show how inflation varies as C is increased when using locking synchronization within component task sets. The figure shows a close-up view of **(a)** the sample means and **(b)** the 99% confidence interval for each mean.

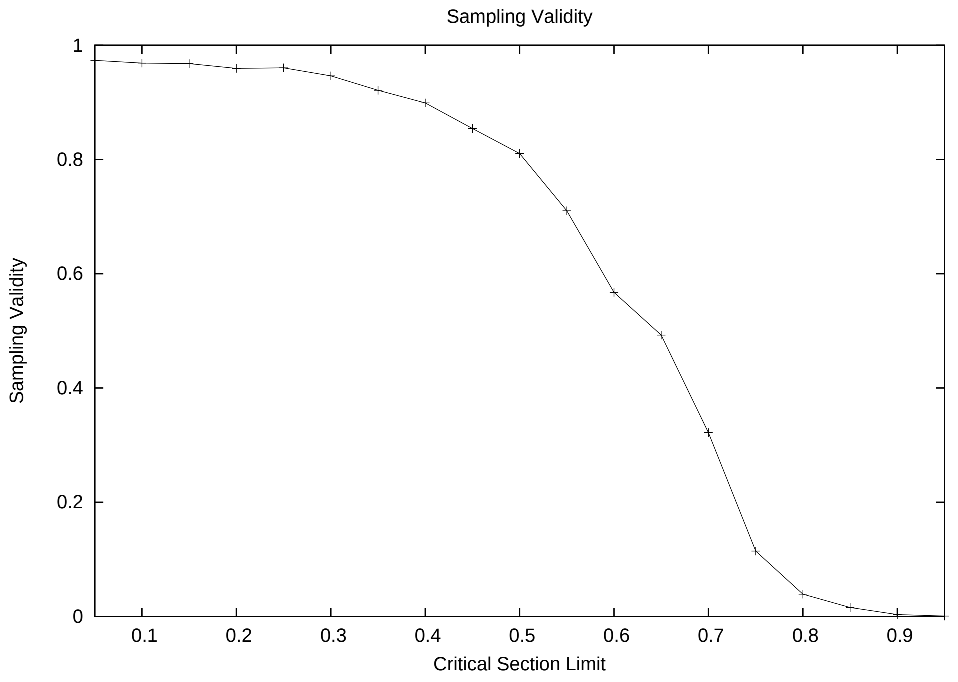


Figure 6.109: Plots show the sampling validity for the graphs showing inflation plotted against C when measuring locking overhead in component task sets.

supertasking. This suggests that the PCP/FP-EDF approach performed poorly in at least a few cases.

6.6.5 Conclusions

These studies showed the same supertasking trends as those presented in Chapter 5. Specifically, supertasking can provide a significant reduction in inflation, though QB-EPDF supertasking tends to provide benefit only to large task sets. As before, task sets were observed that favored the use of each of the NF and FF rules. Hence, neither rule is the best choice in all cases. Despite this, the FF rule was again favored by the majority of the tested sets.

These studies clearly suggest that using the PCP with FP-EDF supertasking tends to produce the least inflation in the majority of the cases in which this approach is applicable.

The SP appears to be the next best choice. However, for low task counts, the SP provides only marginal improvement over the RP when (R3) is satisfied. Since the RP introduces less per-slot overhead, it may be the preferred approach in such cases. Finally, the SWSP consistently produces much more inflation than the zone-based protocols. These studies suggest that the SWSP should be avoided when (R3) is satisfied. However, when (R3) did not hold, up to around 4% of the tested sets *did* favor the use of the SWSP instead of the SP. In addition, recall that the SWSP provides an additional advantage in that it supports long critical sections. The only other approach that provides this capability is the PCP, which is applicable only in special cases.

6.7 Summary

In this chapter, we discussed techniques for supporting locking synchronization in Pfair-scheduled systems. We began by considering the viability of inheritance-based protocols, which are the predominant approach on uniprocessors. Next, we presented two protocols based on the concept of blocking zones. These protocols avoid the preemption of critical sections by exploiting the inherent structure of quantum-based scheduling. Unfortunately, this approach can only be employed when all critical sections are short relative to the scheduling quantum. To support long critical sections, we presented a server-based approach, which is designed to avoid the problems that can arise from more complex locking protocols. In the last section, we compared the proposed approaches both to each other and to the use of the PCP within supertasks.

CHAPTER 7

Pfairness and Symmetric Multiprocessing[1]

In the previous chapters, we have focused on the problem of scheduling tasks using an abstract view of the hardware. Though obviously unrealistic, such a view is beneficial because it exposes inherent limitations in the solutions and leads to the development of techniques that can be applied in a wide range of systems. Despite this abstract view, several universal complicating factors[2] introduced by the hardware have been discussed, *e.g.*, interrupt handling, context switching, signal jitter, and signal postponement.

In this chapter, we embark on a more concrete discussion of the impact of hardware by considering the deployment of Pfair scheduling onto a symmetric multiprocessor (SMP) architecture. SMPs differ from other multiprocessor architectures in that each processor must have identical capabilities: if a task can perform some action on one processor, then it can perform that same action on any processor.

In practice, SMPs typically consist of only a small number of tightly coupled processors that share a centralized memory. Due to symmetry and tight coupling, SMPs appear to be an ideal platform for Pfair scheduling. However, to ensure predictability and reasonable

[1]This chapter is derived from work previously published in the following paper:
[35] P. Holman and J. Anderson. Implementing Pfairness on a symmetric multiprocessor. In *Proceedings of the 10th IEEE Real-time Technology and Applications Symposium*, pages 544–553, 2004.

[2]By "universal," we mean that these factors are expected to be present in any system.

performance, the challenges posed by the architecture must be addressed. These challenges are the focus of this chapter.

Compatibility. We consider an SMP architecture for three reasons. First, symmetric processing is inherent under Pfair scheduling, as each task is assumed to be able to execute on any processor.[3] Second, the frequency of context switches and migrations under Pfair scheduling suggests that tightly coupled processing is needed to keep these overheads reasonable. Third, the use of a central shared memory with coherent caches[4] provides an efficient and transparent mechanism for migrating data between processors. In addition, the worst-case migration cost under such a memory system is effectively no different than that of a preemption. This follows from the fact that the worst-case migration overhead occurs when a task is scheduled and assigned to a processor with a cold cache. When an executing task T is preempted, its cached data can be replaced in the cache by the data of tasks that execute after T on that processor. Hence, a task can experience a cold cache when it resumes after a preemption, even if it resumes on the same processor.

Incompatibility. Unfortunately, SMPs suffer from one major incompatibility with Pfair scheduling: synchronized quantum-based scheduling *promotes* bus contention. Due to the small processor count, processors in an SMP typically access the centralized memory using a shared high-speed bus. Per-processor caches are used to minimize memory accesses, and hence to keep bus contention reasonable. Under Pfair scheduling, contention arises from the fact that a preempted task may encounter a cold cache when it resumes. Since bus traffic

[3]Moir and Ramamurthy's work on "fixed" tasks is an exception [48].

[4]Real-time systems are often cacheless due to the need for reliable timing analysis. Though caches can improve performance, it is often difficult to accurately predict the degree of improvement *a priori*. One advantage of quantum-based scheduling, and hence Pfair scheduling, is that it ensures a minimum separation between preemptions. As a result, caching provides a more predictable benefit in such systems.

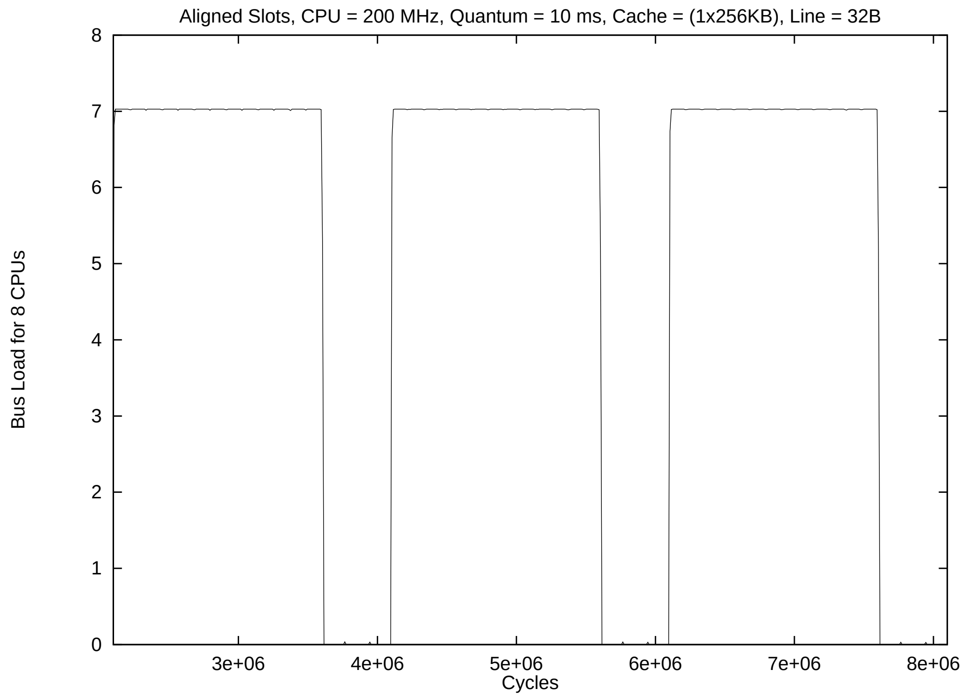
increases while reloading data into the cache, scheduling *all* processors simultaneously can result in very heavy contention at the start of each quantum. The worst-case duration of this contention grows with both the processor count and working-set sizes.[5]

Figure 7.1(a) illustrates this contention on eight processors. The number of pending bus requests across three slots are shown. In this experiment, each task was given an array that matched the cache's size and simply wrote to each cache line iteratively. (The setup of this experiment is explained in greater detail later in the chapter.) As shown, heavy contention occurs at the start of each quantum. Other results, presented later, suggest that such contention can significantly lengthen the execution times of tasks.

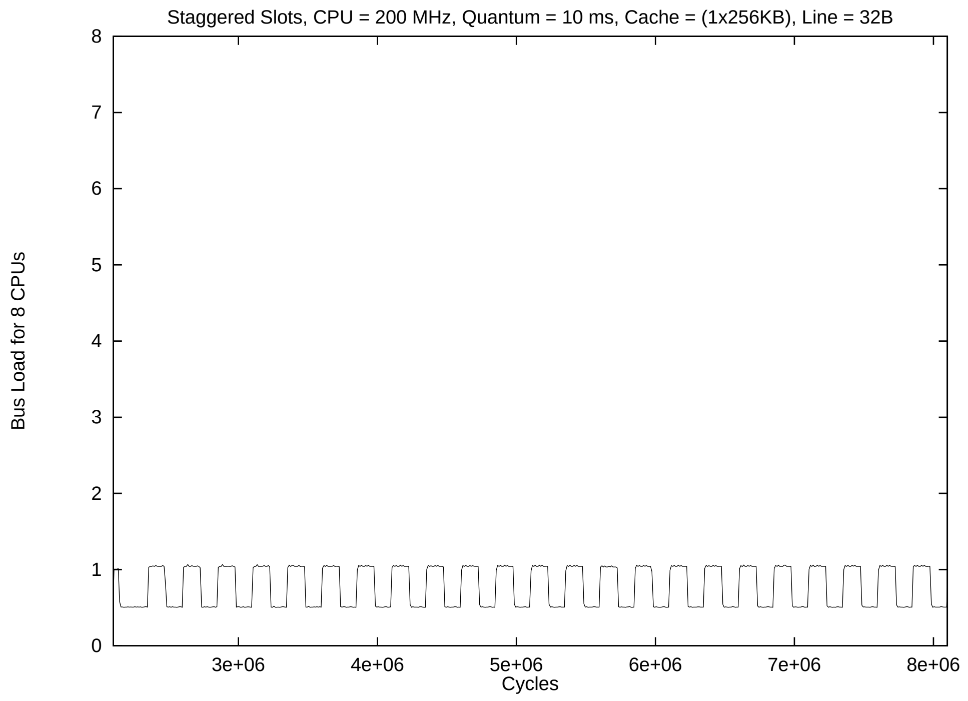
Focus. The primary focus of this chapter is a new *staggered* model for Pfair scheduling that more uniformly distributes these predictable bursts of bus traffic. Results of simulations and experimental measurements obtained from a prototype system suggest that this model can significantly reduce bus contention and hence improve performance. This model provides the additional benefit that scheduling overhead can be distributed across processors, thereby reducing the per-processor overhead. To support the use of staggering, we present an efficient distributed scheduling algorithm and a performance evaluation. Finally, we characterize the impact of the new model on previous Pfair results, including those presented in previous chapters, and explain how these results can be modified for use under the new model.

Models. Figure 7.2(b) shows the proposed staggered model alongside the traditional *aligned* model. Repeating the earlier experiment using the staggered model results in dramatically lower contention, as shown in Figure 7.1(b). Under the aligned model, all scheduling decisions are made by processor 1 (see Figure 7.2(a)). As a result, cycles are lost due to stalling the

[5]The term *working set* refers to the data required by a task to perform its function.



(a) Aligned Model



(b) Staggered Model

Figure 7.1: Each graph shows the bus load across three slots in an eight-processor system using blocking caches. Graphs show contention **(a)** under the aligned model, the **(b)** under the (proposed) staggered model.

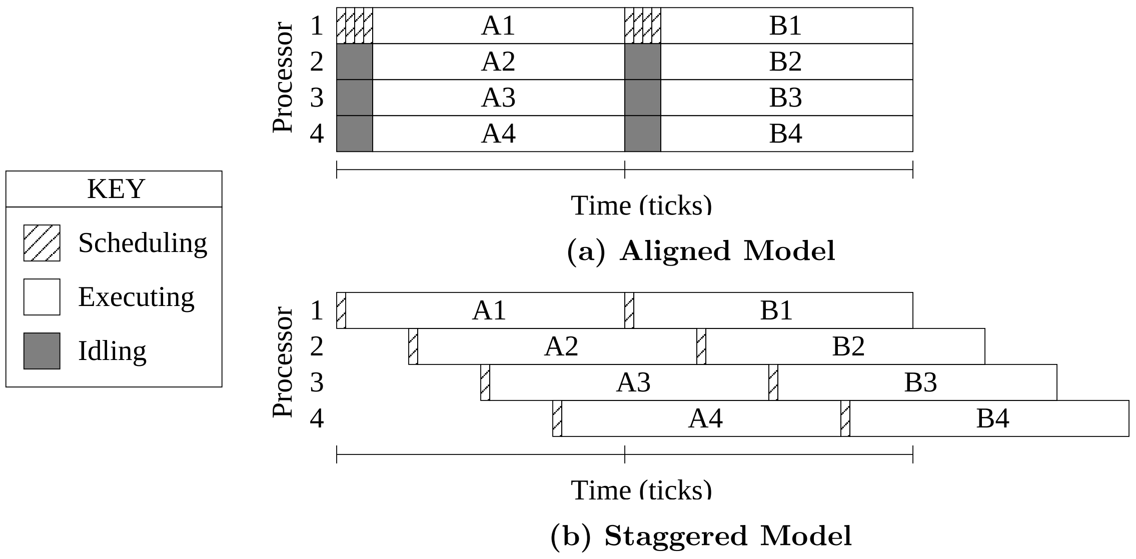


Figure 7.2: **(a)** Under the aligned model, processors are simultaneously scheduled and all decisions are made on a single processor. **(b)** Under the staggered model, scheduling points are uniformly distributed and each processor can make its own scheduling decision.

other processors. Under the staggered model, each processor can make its own scheduling decision (see Figure 7.2(b)). By avoiding processor stalls, both the worst-case and average-case scheduling overhead is reduced.

Before continuing, we formally define the placement of slot boundaries under each of the two models illustrated in Figure 7.2. Let $t(i,k)$ denote the time at which the i^{th} slot boundary occurs on processor k, where $0 \leq k < M$. The aligned model is then defined by the expression $t(i,k) = i$, while the staggered model is defined by $t(i,k) = i + \frac{k}{M}$.

Relevance. The problem of bus contention will obviously not be addressed solely by adopting this new model. If task behaviors are not restricted, then bus contention can also arise on-the-fly due to working-set changes that naturally occur within any task. Although we restrict attention in this chapter to preemption-related bursts of bus traffic, we briefly sketch a more complete strategy for addressing the broader problem of bus contention below.

One approach for managing the bus that has already received much attention is time-division multiplexing. Under this approach, each processor is guaranteed exclusive access to the bus during statically chosen intervals of time. For instance, processor k could be guaranteed exclusive access to the bus at the start of each slot by reserving the interval $\left[i + \frac{k}{M}, i + \frac{k+1}{M}\right)$ for all $i \geq 0$. However, this also implies that processor k cannot access the bus for the remainder of each slot because the bus is reserved for other processors during that time. Under such a division, the interval $\left[i + \frac{k}{M}, i + \frac{k+1}{M}\right)$ acts as a *loading* phase for the i^{th} slot of processor k, during which the scheduled task should load the data it requires into the local processor cache. The remainder of the slot, *i.e.*, the interval $\left[i + \frac{k+1}{M}, i + 1 + \frac{k}{M}\right)$, then acts as a *work* phase, during which the task is permitted to manipulate only the cached data.

The primary disadvantage of this approach is that performance depends heavily on whether tasks can be restructured to respect such a restriction without wasting processor time. If a task is within its slot's work phase but requires uncached data to make further progress, then the task will be stalled until the next loading phase. As a result, the remainder of the current quantum will be wasted. Determining the degree of waste that is likely to occur is a daunting task since it requires fairly extensive knowledge of how real task sets are structured. As in previous chapters, we necessarily leave work that requires such knowledge as a topic for further investigation.

Another complication that impacts this approach is that performance will depend on the cache structure and coherence policy. Since bus traffic should not be generated during a work phase, a *write back* or similar policy is clearly necessary for this approach to be successful. Furthermore, low associativity is undesirable since it may prevent two segments of memory that map to the same cache line from co-existing in the cache, even if some lines in the cache are unused. Such restrictions reduce the amount of data that can be used during the work phase, and hence will likely increase waste.

In practice, it may be impractical to restrict bus access to a single processor. Due to operating system activities, such as the scheduling, it may be necessary to allow more than one processor to access the bus. However, such activities differ from task activities because the algorithms used by the operating system are known prior to runtime. Due to this increased knowledge, we expect the pessimism in the analysis to remain reasonable as long as access to the bus is restricted to at most one *task* at a given time. However, further study is certainly needed to determine if this expectation is accurate and to investigate the related issues discussed above. In addition, the compatibility of Pfair scheduling with multiprocessors that use other types of interconnection networks is another important topic for further study.

Related work. Unfortunately, little (if any) prior work has considered techniques for improving performance in multiprocessor systems that require worst-case predictability. Consequently, available techniques are typically heuristic in nature and lack supporting analysis. Affinity-based scheduling algorithms are one common example. Such algorithms boost the priorities of currently executing tasks to make preemption less likely. By resisting preemption, these algorithms tend to improve cache performance, and hence reduce task execution times. Although the use of such techniques does improve average-case performance, the complexity of the algorithms makes worst-case analysis difficult. As a result, these techniques are not appropriate for hard real-time systems.[6] Though we are interested in average-case performance and soft real-time systems, our primary focus is on determining the inherent cost of guaranteeing predictable operation in multiprocessor systems. Techniques for optimizing Pfair scheduling for a general-purpose environment in which worst-case predictability is not

[6]Later, we consider a more limited use of processor affinity when assigning scheduled tasks to processors. This limited use differs from that described here in that task priorities are *not* dependent on processor affinity.

required were previously proposed by Chandra, Adler, and Shenoy [18].

7.1 Scheduling Algorithm

In this section, we present an efficient scheduling algorithm for the staggered model.

7.1.1 Safety and Assignment Affinity

Unfortunately, staggered scheduling is slightly more complicated than traditional Pfair scheduling. Under staggering, quanta from different slots can overlap, as the A3 and B1 quanta illustrated in Figure 7.2(b). Hence, the scheduler must ensure that tasks scheduled *back-to-back* (*i.e.*, in consecutive slots) do not execute within overlapping quanta.

Affinity-based processor assignment. Safety can be ensured by employing an affinity-based policy when assigning scheduled tasks to processors. A variety of policies can be implemented by maintaining the small amount of scheduling history summarized below.

Previous Processor (PP): the most recent processor on which a task executed;

Previous Slot (PS): the most recent slot in which a task executed;

Previous Task (PT): the most recent task to execute on a processor.

Each of the above fields can be maintained by the scheduling algorithm at the cost of only a trivial increase in scheduling overhead. All of the following affinity-based policies, except for the last, can be implemented using one or more of the above fields.

Back To Back (BTB): When scheduled in consecutive slots, a task must be granted the same processor. Requires the PP and PS fields.

Last Scheduled (LS): Each processor must be assigned to the task to which it was last assigned, if that task is selected. Requires the PT field.

Most Recent Processor (MRP): Each selected task is assigned to the processor to which it was most recently assigned; if multiple tasks should be assigned the same processor, the tie is broken in favor of the task that most recently executed. Requires the PP and PS fields.

Most Recent Available Processor (MRAP): An extension of the MRP policy that considers processors beyond the most recently assigned processor. Specifically, when a task loses a tie for its desired processor, the task tries to execute on the processor on which it executed prior to the unavailable processor. This continues until the task is assigned a processor or its scheduling history is exhausted. This policy can be efficiently implemented by maintaining a list of processors in affinity order for each task, *i.e.*, the assigned processor is moved to the head of the list each time the task is scheduled. By using a doubly linked list, this list update can be implemented with constant time complexity.

Unfortunately, assignment policies are more difficult to implement under staggering, as we explain below. However, at least the BTB and LS policies can be implemented with minimal effort. Since the BTB policy is sufficient to prevent the allocation of overlapping quanta, we focus on it for the remainder of the chapter.

In addition to ensuring safety under staggered scheduling, these policies are desirable because they can improve cache performance. When used with supertasks, the benefits of such a policy can be substantial, *i.e.*, the cache performance of a group of EDF-scheduled component tasks in a heavily weighted, fully preemptive supertask resembles that of using

EDF scheduling on a dedicated uniprocessor. Indeed, when using any of the policies described above, a unit-weight supertask effectively becomes a dedicated uniprocessor.

7.1.2 Concept

Before presenting the algorithm, we consider the problem of distributed scheduling with assignment affinity abstractly. By doing so, we intend to motivate the design of the algorithm presented later. Consider scheduling slot k on the first processor after executing task T on that processor in slot $k-1$. To be computationally efficient, the scheduler must require only $O(\log|\tau|)$ time on each processor. (PD^2 schedules all M processors in $O(M\log|\tau|)$ time [61].) In addition, the decision must respect the BTB policy described above to ensure safety of the executing task, *i.e.*, if T is selected to execute in slot k, then it must execute on the first processor. However, simply identifying all tasks that are selected for slot k is an $\Omega(M)$ operation.

The implication is that the tasks scheduled in slot k must be identified *before* invoking the scheduler at the start of slot k. This can be achieved by dividing scheduling into two steps: **(i)** up to M tasks (if that many are eligible) are selected to execute in slot k and stored in k's *scheduling set*, and **(ii)** each processor (later) selects a task to execute in its local slot k from those in k's scheduling set. To ensure that k's scheduling set is known before slot k begins on any processor, Step (i) can be performed *one slot early* (*i.e.*, by the scheduler invocations associated with slot $k-1$). Specifically, processor p's scheduler invocation in slot $k-1$ first selects a task to execute on processor p in slot $k-1$ from that slot's scheduling set, and then selects a task to add to the scheduling set of slot k (if an eligible task exists).

Necessity. Safety can actually be ensured by using a weaker policy than BTB. Specifically, the policy described below is both necessary and sufficient to ensure safety.

Directed Migration (DM): When scheduled back-to-back, a task that executed on processor p in the earlier slot must execute on a processor with index at least p in the later slot. Requires the PP and PS fields.

Under staggered scheduling, this policy provides an advantage over the BTB policy in that look-ahead scheduling, like that described above, is not necessary. However, because we are also interested in improving cache performance, we consider only the BTB policy here.

7.1.3 Basic Algorithm

We begin by presenting procedures (shown in Figure 7.3) needed to support Pfair or ER-fair scheduling of static task sets. Later, we present additional procedures to support the remaining extensions to Pfair scheduling.

Data structures. Scheduling sets are implemented by the *SchedNow*, *ReschedNow*, *SchedNext*, and *ReschedNext* heaps. When scheduling slot k, tasks scheduled in slot k (respectively, $k+1$) are stored in the *Now* (respectively, *Next*) heaps. The *Sched* (respectively, *Resched*) heaps store tasks that are not (respectively, are) scheduled back-to-back. The *Eligible* heap stores all remaining tasks that are eligible in slot $k+1$. All heaps are ordered according to task priorities. (The $\preceq$ and $\succeq$ relations, which define this ordering, are defined below.) In addition, the *Incoming*$[k]$ heap stores tasks that will not be eligible to execute until slot k. (We present these heaps as an unbounded array solely to simplify the presentation.) We assume that each task is initially stored in the appropriate *Incoming* heap.

Each task is represented by a record that contains (at least) the earliest slot in which the task may next execute (*elig*) and the task's current priority (*prio*). We assume that

```
typedef task:
    record
        $elig$: integer;
        $prio$: $ADT$

shared var
    $k$: integer initially $-1$;
    $SchedCount$: integer initially $M$;
    $Running$: array $1 \ldots M$ of task $\cup \{\perp\}$
                initially $\perp$;
    $SchedNow$: min-heap of task initially $\emptyset$;
    $ReschedNow$: min-heap of task initially $\emptyset$;
    $SchedNext$: min-heap of task initially $\emptyset$;
    $ReschedNext$: min-heap of task initially $\emptyset$;
    $Eligible$: max-heap of task initially $\emptyset$;
    $Incoming$: array $0 \ldots \infty$ of max-heap

private var
    $p$: $1 \ldots M$; $T$: task $\cup \{\perp\}$

procedure Initialize()
1:  $Eligible := Eligible \cup Incoming[0]$;
2:  while $|Eligible| > 0 \wedge |SchedNext| < M$ do
3:      $SchedNext :=$
            $SchedNext \cup \{$ExtractMax($Eligible$)$\}$
    do

procedure SelectTask($T$)
4:  if $T \in S_k$ then
5:      $ReschedNext := ReschedNext \cup \{T\}$
    else
6:      $SchedNext := SchedNext \cup \{T\}$
    fi

procedure Schedule($p$)
7:  if $SchedCount = M$ then
8:      $k := k + 1$;
9:      $Eligible := Eligible \cup Incoming[k + 1]$;
10:     Swap($SchedNow$, $SchedNext$);
11:     Swap($ReschedNow$, $ReschedNext$)
    fi;
12: if $Running[p] \in ReschedNow$ then
13:     $T := Running[p]$;
14:     $ReschedNow := ReschedNow \;/\; \{T\}$
15: else if $|SchedNow| > 0$ then
16:     $T :=$ ExtractMin($SchedNow$)
    else
17:     $T := \perp$
    fi;
18: $Running[p] := T$;
19: if $T \neq \perp$ then
20:     UpdatePriority($T$);
21:     if $T.elig \leq k + 1$ then
22:         $Eligible := Eligible \cup \{T\}$
        else
23:         $Incoming[T.elig] :=$
                $Incoming[T.elig] \cup \{T\}$
        fi
    fi;
24: if $|Eligible| > 0$ then
25:     SelectTask(ExtractMax($Eligible$))
    fi;
26: $SchedCount := (SchedCount \bmod M) + 1$
```

Figure 7.3: Basic staggered scheduling algorithm.

task priorities are implemented as an abstract data type that supports the $\prec$, $\preceq$, $\succ$, and $\succeq$ comparisons, where $\rho_1 \prec \rho_2$ (respectively, $\rho_1 \preceq \rho_2$) implies that priority ρ_1 is strictly higher than (respectively, at least as high as) priority ρ_2. We further assume that `UpdatePriority` encapsulates the algorithm for updating *prio* and *elig*.

The remaining variables include two counters (k and *SchedCount*) and the *Running* array. k is the index of the current slot. *SchedCount* indicates the number of scheduler invocations that have been performed for slot k. Finally, the *Running* array indicates which task is currently executing on each processor.

To simplify the presentation, some branches test for set inclusion ($\in$) and the branch at line 4 uses S_k to denote the set of tasks selected to execute in slot k, *i.e.*, the scheduling set of k. All of these tests can be implemented in $O(1)$ time complexity and $O(|\tau|)$ space complexity by associating a few additional variables with each task.

Detailed description. `Initialize` is invoked before all other procedures to create slot 0's scheduling set. First, tasks present at time 0 are merged into the *Eligible* heap at line 1. Lines 2–3 then make the scheduling decisions.

`SelectTask` schedules a task T in slot $k+1$. Line 4 determines whether T is scheduled back-to-back. T is then stored in the proper heap in lines 5–6.

`Schedule`(p) is invoked to schedule processor p. Lines 8–11 initialize a round of scheduler invocations by incrementing k, by merging newly eligible tasks into *Eligible*, and by initializing *SchedNow*, *ReschedNow*, *SchedNext*, and *ReschedNext*. Lines 12–18 select the task to execute on processor p and record the decision. The task is then updated and stored according to the priority of its successor subtask in lines 20–23. Lines 24–25 then select a task to execute in the slot $k+1$. Finally, progress is recorded by updating *SchedCount* at line 26.

Example. Figure 7.4 shows a sample schedule produced by the staggered algorithm. To illustrate the operation of the algorithm, a trace of task T3's heap-membership changes is presented in Figure 7.5.

7.1.4 Extensions

We now present the procedures needed to support tasks leaving and joining the system (shown in Figure 7.6). Systems consisting of dynamic task sets and those supporting ISfair and

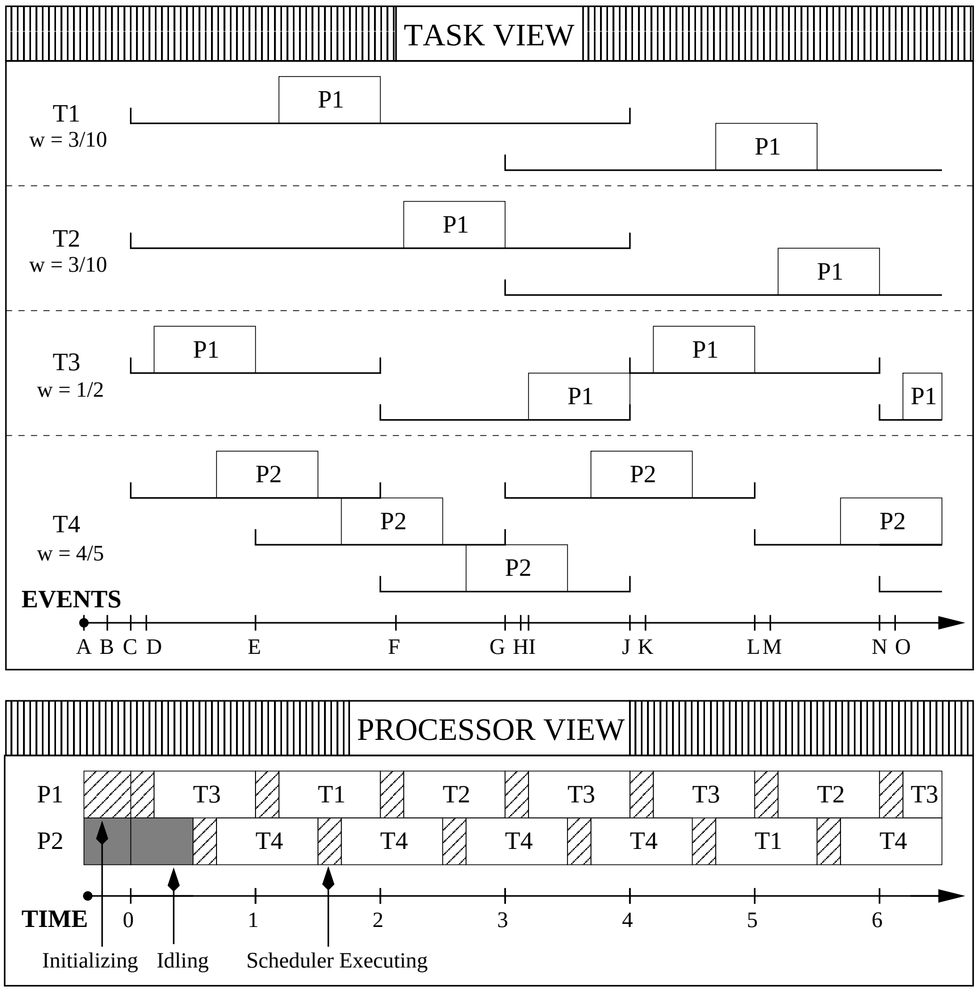


Figure 7.4: Four tasks (T1–T4) are executed on two processors (P1–P2) under the staggered model. The schedule is shown from two different views.

GISfair tasks require such support. In addition, systems that permit tasks to change weights at runtime will also require this support.[7]

Concerns. Two problems arise from tasks leaving and joining. First, these actions require modification of the scheduler's data structures. Hence, access to these structures must be synchronized.[8] Second, adding and removing tasks can lead to incorrect scheduling decisions.

[7]As explained earlier in Chapter 2, one simple technique for supporting weight changes is to require the task to leave with the old weight and re-join with the new weight.

[8]The need for explicit synchronization can potentially be avoided by postponing handling of all leave and

T3's HEAP TRANSITIONS		
EVENT	***DESTINATION***	***CAUSE OF TRANSITION***
A	Eligible	Initialize(): T3 is eligible to execute in slot 0.
B	SchedNext	Initialize(): T3 selected to execute in slot 0.
C	SchedNow	Schedule(P1): Slot 0 decisions committed.
D	Incoming[2]	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 2).
E	Eligible	Schedule(P1): T3 is eligible to execute in slot 2.
F	SchedNext	Schedule(P1): T3 is selected to execute in slot 3.
G	SchedNow	Schedule(P1): Slot 3 decisions committed.
H	Eligible	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 4).
I	ReschedNext	Schedule(P1): T3 is selected to execute in slot 4 and will execute back-to-back.
J	ReschedNow	Schedule(P1): Slot 4 decisions committed.
K	Incoming[6]	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 6).
L	Eligible	Schedule(P1): T3 is eligible to execute in slot 6.
M	SchedNext	Schedule(P1): T3 is selected to execute in slot 6.
N	SchedNow	Schedule(P1): Slot 6 decisions committed.
O	Incoming[8]	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 8).

Figure 7.5: Scheduling events affecting task **T3** are summarized in the table. Event labels correspond to those shown on the timeline in the upper inset of Figure 7.4.

To avoid this potential problem, the presented procedures are designed to ensure that one of the two conditions given below holds after the execution of every procedure.

(I1) $|SchedNext| + |ReschedNext| = SchedCount$ and, for all $T \in (SchedNext \cup ReschedNext)$ and $U \in Eligible$, $T.prio \preceq U.prio$.

(I2) $|SchedNext| + |ReschedNext| < SchedCount$ and $|Eligible| = 0$.

join requests until the next slot boundary. However, this approach has its own disadvantages in that postponed processing may lead to unnecessary idling and will also increase the per-slot overhead.

```
private var
    $H$: pointer to min-heap of task

procedure Deactivate($T$)
27: if $T \in SchedNext \cup ReschedNext$ then
28:     if $T \in ReschedNext$ then
29:         $ReschedNext := ReschedNext \,/\, \{T\}$
        else
30:         $SchedNext := SchedNext \,/\, \{T\}$
        fi;
31:     if $|Eligible| > 0$ then
32:         SelectTask(ExtractMax($Eligible$))
        fi
33: else if $T \in SchedNow \cup ReschedNow$ then
34:     if $T \in ReschedNow$ then
35:         $ReschedNow := ReschedNow \,/\, \{T\}$
        else
36:         $SchedNow := SchedNow \,/\, \{T\}$
        fi;
37:     UpdatePriority($T$)
38: else if $T \in Eligible$ then
39:     $Eligible := Eligible \,/\, \{T\}$
    else
40:     $Incoming[T.elig] :=$
            $Incoming[T.elig] \,/\, \{T\}$
    fi
```

```
procedure Activate($T$)
41: if $T.elig \leq k + 1$ then
42:     if $|SchedNext| + |ReschedNext|$
                $< SchedCount$ then
43:         SelectTask($T$)
        else
44:         if $|SchedNext| = 0 \vee (|ReschedNext| > 0$
                $\wedge$ Min($SchedNext$)$.prio \preceq$
                    Min($ReschedNext$)$.prio$) then
45:             $H := \&ReschedNext$
            else
46:             $H := \&SchedNext$
            fi;
47:         if Min($*H$)$.prio \prec T.prio$ then
48:             SelectTask($T$);
49:             $T :=$ ExtractMin($*H$)
            fi;
50:         $Eligible := Eligible \cup \{T\}$
        fi
    else
51:     $Incoming[T.elig] := Incoming[T.elig] \cup \{T\}$
    fi
```

Figure 7.6: Extensions to support task departures (left) and task arrivals (right).

Since *SchedCount* records the number of completed scheduler invocations in the current round of decisions, it should be the case that *SchedCount* tasks have been scheduled in the upcoming slot after each invocation. Informally, (I1) states that *SchedCount* tasks have been tentatively scheduled in the upcoming slot (first clause) and that each scheduled task has higher priority than each unscheduled task (second clause). In the event that too few eligible tasks exist, (I1) cannot be satisfied. This possibility is addressed by (I2). Informally, (I2) states that less than *SchedCount* tasks have been tentatively scheduled (first clause), which suggests an idle processor in the upcoming slot, and that no other tasks are eligible for that slot (second clause). We explain how our algorithm guarantees that either (I1) or (I2) always holds by exhaustively considering all possible scenarios below.

Detailed description. Invoking `Deactivate`(T) in slot k causes task T to be ignored when scheduling slots at and after $k + 1$. If T has been selected to execute in slot k but has not been granted a processor, then the decision to execute T is nullified; however, T is still charged as if it did execute. (This is analogous to having a task suspend immediately after it is granted the processor: the entire quantum is wasted.) When removing T, three cases must be considered to ensure that either (I1) or (I2) holds after execution: **(i)** T is scheduled in slot $k + 1$ (it is in either *SchedNext* or *ReschedNext*), **(ii)** T is scheduled in slot k but has not been granted a processor (it is in either *SchedNow* or *ReschedNow*), and **(iii)** T is not scheduled in slot $k+1$ (but may be currently executing in slot k). Lines 28–32 handle Case (i) by locating and removing T from either *SchedNext* or *ReschedNext* at lines 28–30 and then scheduling a replacement task at lines 31–32. Lines 33–37 handle Case (ii) by locating and removing T from either *SchedNow* or *ReschedNow* at lines 33–36 and then charging T for the unused quantum at line 37. Lines 38–40 handle Case (iii).

Invoking `Activate`(T) within slot k causes task T to be considered when scheduling slots at and after $k + 1$. Again, three cases must be considered to ensure that either (I1) or (I2) holds after execution: **(i)** T is not eligible to execute in slot $k + 1$, **(ii)** T is eligible to execute in slot $k + 1$ and a processor will idle in that slot, and **(iii)** T is eligible to execute in slot $k + 1$ but no processor will idle in that slot. Lines 51 and 42–43 handle Cases (i) and (ii), respectively. Lines 44–50 handle Case (iii). Specifically, lines 44–46 determine which of *SchedNext* and *ReschedNext* contains the lowest-priority task that is scheduled in slot $k + 1$. If this task's priority is lower than T's priority, then it is removed and replaced by T at lines 47–49. Whichever task is not scheduled in slot $k + 1$ is then stored in *Eligible* at line 50.

Usage. Observe that `Deactivate` neither halts executing tasks nor modifies *Running*.

The presented procedures are designed to be used as subroutines when implementing more complex services. Consequently, each procedure takes only those actions that are essential to achieving its goals. Indeed, it is not possible to present universal procedures for most services because their designs are based, at least in part, on policy decisions. For instance, a policy must be adopted to define how the system reacts to weight changes that cannot be immediately applied. Such policies are outside the scope of this discussion.

7.1.5 Time Complexity

Since task priorities can be updated and compared in constant time, the only significant complexity results from heap operations. Each procedure performs a constant number of heap operations and a constant number of calls to other procedures. Therefore, the time complexity of each procedure is $O(\log |\tau|)$ when using binomial heaps and PD^2 task priorities, and the aggregate time complexity of M scheduler invocations is $O(M \log |\tau|)$. Hence, the presented algorithm is computationally efficient.

Recall that under the aligned model, all scheduling decisions are made on a single processor, resulting in $O(M \log |\tau|)$ time complexity on that processor. The per-processor overhead under the staggered model is proportional to the time required by one invocation of `Schedule`, which has only $O(\log |\tau|)$ time complexity. This suggests that the staggered model can provide up to a factor-of-M improvement with respect to scheduling overhead. (The actual improvement will depend on the system architecture, as we later show.)

7.2 Impact on Analysis

The most problematic aspect of shifting from an aligned to a staggered model is the impact on prior results. Though supporting the new model is reasonably straightforward (as we demonstrated in the last section), we must also consider the impact of staggering on analytical results and mechanisms proposed for Pfair scheduling. Unfortunately, it would be impractical to consider each prior result in detail here. For brevity, we discuss this issue more broadly in this section by focusing on the side effects produced by staggering. We then illustrate the adaptation of prior results by explaining the impact of these side effects on the results presented earlier in this dissertation.

Side effects. A staggered slot extends up to $\Delta \overset{\text{def}}{=} \frac{M-1}{M}$ beyond the placement of the corresponding slot under the aligned model. Hence, slot k on a processor may overlap slots $k+1$ and $k-1$ on other processors, leading to the following side effects.

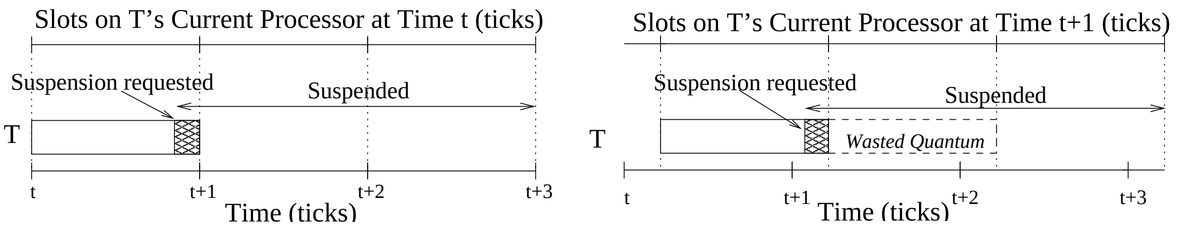
(E1) An event occurring at time t under the aligned model may be delayed until time $t + \Delta$ under staggering.

(E2) Each slot overlaps $M-1$ other slots when aligned, but $2(M-1)$ when staggered.

7.2.1 Independent Tasks

We begin by considering independent-task scheduling. Independent tasks are oblivious to the concurrent execution of other tasks and, hence, are unaffected by (E2). (E1), on the other hand, has two implications, which are discussed below.

First, deadlines guaranteed under the aligned model may be missed by up to Δ under staggering. We expect such misses, which are bounded by the slot length, to be acceptable in



(a) Aligned Model **(b) Staggered Model**

Figure 7.7: Illustration of how event timing differs between **(a)** the aligned model and **(b)** the staggered model. An event in slot t is no longer guaranteed to occur before time $t + 1$ under the staggered model. Due to this, a task may have already been scheduled in the next slot before a suspension request is issued, as shown in (b).

many cases. For instance, such a guarantee is inherent under proportional-share uniprocessor scheduling [66]. As Strict deadlines can be enforced simply by treating task deadlines as if they are Δ units earlier than they actually are. The resulting loss depends on the task's parameters: a task T requiring a quanta every b slots will need $T.w \approx \frac{a}{b}$ under the aligned model, but $T.w \approx \frac{a}{b-1}$ under staggering. (This overhead can be more precisely characterized for specific tasks by adapting the analysis presented earlier in Chapter 3.)

Second, events (such as suspension requests) occurring in slot k may occur *after* time $k+1$, at which point the scheduling decisions for slot $k+1$ are committed. As a result, a suspending task may occasionally cause a quantum to be wasted, as illustrated in Figure 7.7. (Recall that a quantum is wasted when a scheduled task leaves before being assigned a processor.)

Server tasks that suspend when no requests are pending will likely be most impacted by this property since worst-case analysis must pessimistically assume that a quantum is wasted each time the server suspends. Figure 7.8 depicts the worst-case scenario for a single request and response pair. Under the aligned model (see Figure 7.8(a)), task T sends a request to server S in slot 0. S then becomes active at time 1, services the request in slot 1, sends the response before time 2, and then suspends at time 2. Having received the request, T becomes

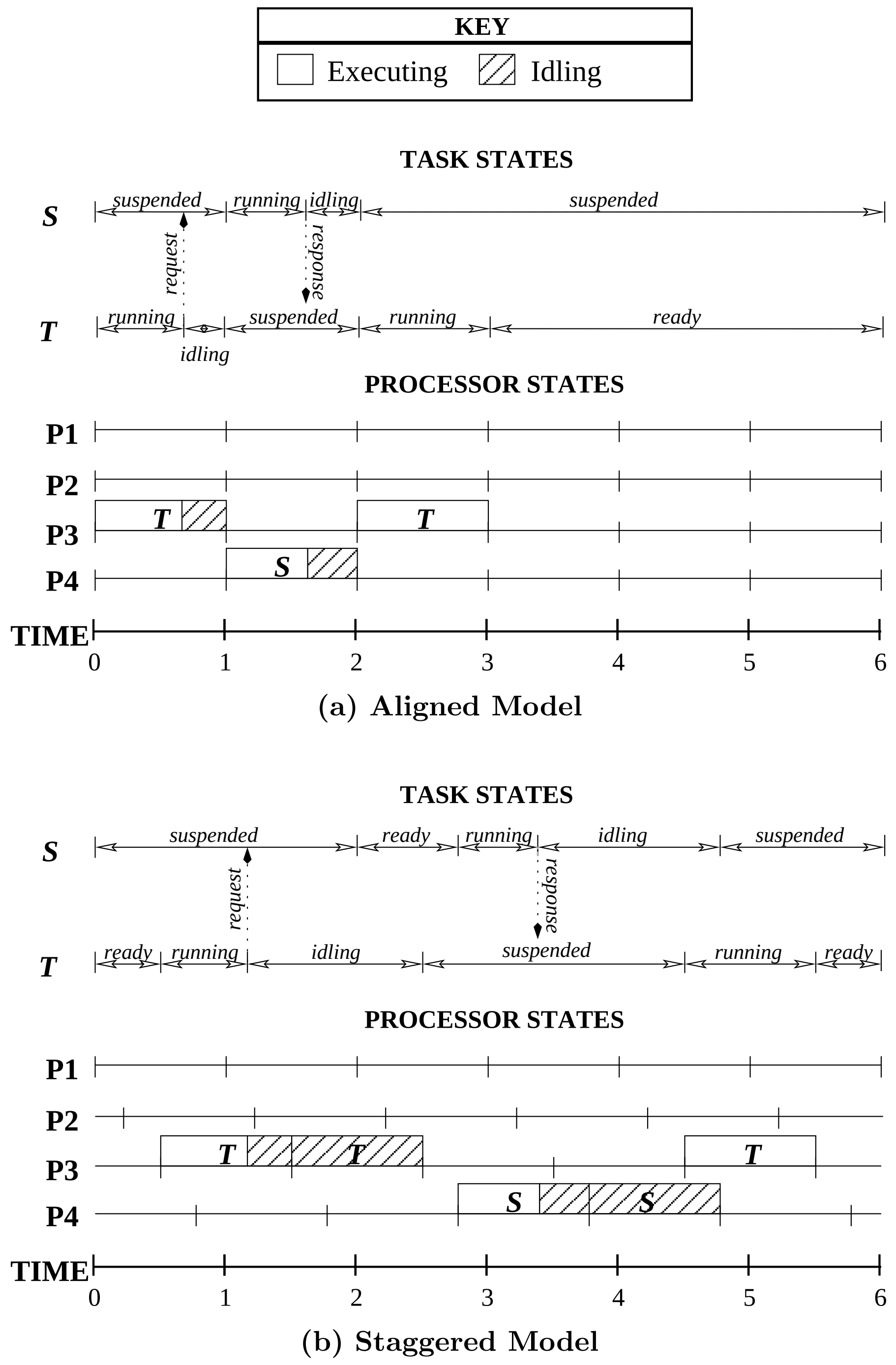


Figure 7.8: Illustration of how staggering adversely impacts client/server designs. The servicing of one request is shown **(a)** under the aligned model and **(b)** under the staggered model.

ready at time 2 and is immediately scheduled in slot 2.

On the other hand, under the staggered model (see Figure 7.8(b)), T is scheduled on processor 2 in slot 0. As a result, its request is sent after time 1, at which point the scheduling decisions for slot 1 are committed. Because of this, T is scheduled in slot 1 even though it cannot make progress, and S remains suspended until time 2. T's suspension is finally processed at time 2. Upon becoming active, S is scheduled in slot 2 and executes on processor 3. S's response is then sent after time 3. As a result, T remains suspended until time 4, while S is re-scheduled in slot 3 and idles. T finally resumes at time 4.

In the staggering scenario, both the requesting task and the server waste a quantum (in slots 1 and 3, respectively). In addition, notice that the delay between the request and response is longer under staggering. Though unlikely, this worst-case scenario must be assumed to occur (under worst-case analysis) each time a request is issued unless other restrictions are placed on the runtime behavior of tasks.

7.2.2 Supertasking

Recall (from Chapter 4) that a supertask weight is chosen by comparing the least amount of processor time guaranteed to a supertask (*i.e.*, the supply) to the maximum requirement of all component tasks (*i.e.*, the demand). Only the first of these two quantities requires adjustment due to staggering. In addition, this quantity is unaffected by (E2).

The primary impact of (E1) is that the amount of processor time guaranteed to the supertask is slightly lower under staggering. In this case, the delay suggested by (E1) can be accounted for by reducing the estimate of each supertask's supply by Δ. Figure 7.9 shows how (E2) can impact the supply in this way. However, notice that this impact only occurs when the supertask is scheduled in the last slot of the interval, as depicted in Figure 7.9. Hence, staggering only affects estimates of supply that are based on such intervals. By taking

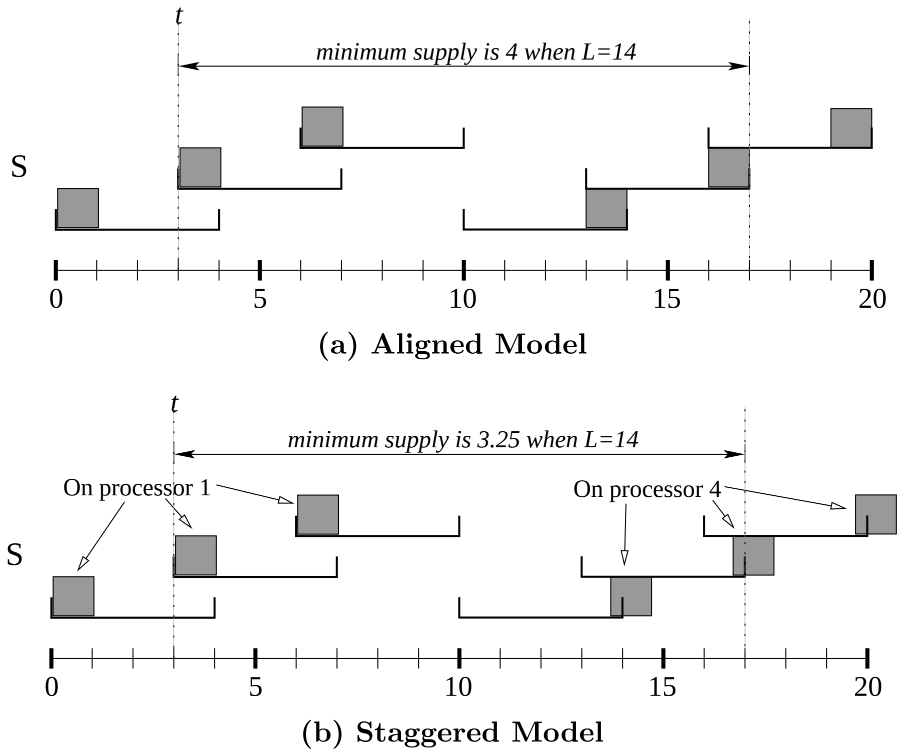


Figure 7.9: The interval that defines the supply of a supertask (*i.e.*, minimum guaranteed allocation) when $M = 4$ (and hence $\Delta = \frac{3}{4} = 0.75$) and $L = 14$ is shown **(a)** under the aligned model and **(b)** under the staggered model. Due to staggering, the supply under the staggered model is Δ units lower than that under the aligned model.

this fact into account when deriving the supply function, tighter bounds can be obtained.

7.2.3 Lock-free Synchronization

Recall (from Chapter 5) that lock-free operations are optimistically attempted and retried until successful. Under the aligned model, it is sufficient to assume that the worst-case mix of $M - 1$ interfering tasks when determining the worst-case number of retries. By (E2), similar reasoning can still be applied under staggering; however, $2(M - 1)$ tasks must be considered, which potentially doubles the overhead. It is possible to derive tighter bounds by considering that only a fraction of each of those $2(M - 1)$ overlapping slots actually overlaps the slot in question. However, the resulting analysis would be much more complex (and time-consuming) than that presented in Chapter 5.

7.2.4 Zone-based Lock Synchronization

Both the SP and RP (from Chapter 6) delay lock requests to prevent the preemption of critical sections. As in the lock-free case, this analysis assumes that each task T can be preempted at most once before an operation completes when the lock in question satisfies (R3). Hence, the situation is quite similar to that discussed above: bounds on T's worst-case blocking overhead are computed by considering the interference of $M - 1$ tasks under the aligned model and $2(M - 1)$ tasks under the staggered model. Unlike the lock-free case discussed above, there is little benefit provided by the property that only a fraction of each overlapping slot actually overlaps. This follows from the fact that each task that executes in parallel with the requesting task can interfere with (*i.e.*, block) the lock request only once. (Under lock-free analysis, consideration of fractional quanta would be beneficial primarily because it would enable the use of tighter bounds on the number of interferences that occur within the quantum.) The results based on (R1) can be similarly adapted.

7.2.5 Server-based Lock Synchronization

Since analysis of the SWSP (from Chapter 6) assumes that every competing task has its critical-section request executed before that of the requesting task T, the worst-case time required to process T's request is unaffected by (E1) and (E2). However, since both the requesting task and lock server may suspend under this protocol, they can suffer from the suspension-related problems mentioned earlier in Section 7.2.1. For the server, this problem translates into slightly inflated worst-case response times.

7.2.6 Evaluating the Trade-off

Staggering represents a trade-off between off-line schedulability and on-line performance. Based on the above discussion, staggering introduces additional overhead in four areas: mapping, reweighting, suspensions, and synchronization. We consider each below.

First, consider mapping overhead. Based on the observations made in Section 7.2.1, the increase in mapping overhead for a periodic or sporadic task T is approximately $\frac{a}{b-1} - \frac{a}{b}$, where $a \approx \frac{T.e}{Q}$ and $b \approx T.p$. (These quantities are approximate because tasks may need to request slightly more processor time than they can actually use to ensure that deadlines are met under quantum-based scheduling, as implied by Lemmas 3.4–3.7.) This estimate simplifies to $\frac{T.u}{Q(T.p-1)}$. It follows that mapping overhead is most impacted when $T.p$ is small, *i.e.*, comparable to the slot size. However, quantum-based scheduling cannot efficiently support tasks for which period sizes are comparable to the slot size because many forms of overhead (including mapping and reweighting overhead) are prohibitively high in such cases. For such cases, alternative scheduling approaches must be considered.[9] Hence, such small periods should occur rarely, if at all, in practice.

Second, consider reweighting overhead. To estimate the impact of staggering on reweighting overhead, we repeated the first study from Chapter 4 and restricted attention to the FP-EDF scenario. The results of this study are given in Appendix C. (The QB-EDF and QB-EPDF scenarios suffer from both mapping and reweighting overhead, while the FP-EDF scenario suffers from only reweighting overhead.) These results suggest that staggering increases the reweighting overhead by a factor of approximately 2.5. Since reweighting overhead

[9]Fully preemptive supertasking may be a viable alternative in such cases since mapping is unnecessary.

is very low under both the FP-EDF and QB-EDF[10] approaches (see Chapter 4), it will likely remain reasonably low under staggered scheduling. On the other hand, the reweighting overhead may become prohibitively high under the QB-EPDF approach. However, since the QB-EDF approach will likely be used instead of the QB-EPDF approach, the total reweighting overhead is not expected to increase significantly due to staggering.

Third, consider the overhead introduced by suspensions. As explained earlier in Chapter 1, a task that yields before the next scheduling point under quantum-based scheduling is still charged for the unused processor time. Hence, frequent suspensions lead to both poor task performance and low processor utilization. To achieve good performance, tasks exhibiting such behaviors should *not* be included in the global schedule, if it can be avoided. Indeed, this observation was a driving motivation behind our consideration of fully preemptive supertasking, which allows such tasks to be scheduled *as a group* rather than individually. The benefit of this approach is that the global scheduler must handle only the supertask, which never suspends. In addition, component tasks can be scheduled using a uniprocessor algorithm that is more flexible with respect to suspensions. Given the availability of mechanisms that avoid this problem, suspensions are expected to occur infrequently in the global schedule and, hence, should contribute little to the actual cost of staggering.

Finally, consider synchronization overhead. To optimize performance, synchronization at the global level should be limited to lock-free and zone-based locking synchronization, if possible. As illustrated in earlier chapters, these approaches tend to produce low overhead in most cases. Since staggering at most doubles this overhead, the cost of synchronization should remain reasonable under staggering.

[10]QB-EDF primarily suffers from mapping overhead.

Based on the above reasoning, staggering is expected to produce only a small increase in overhead in most systems. Indeed, it seems that staggering introduces significant overhead primarily in situations in which Pfair scheduling is likely to perform poorly anyway. More importantly, experimental results presented in the next section suggest that substantial performance gains can be achieved through staggering. These gains translate into decreased execution times, which are beneficial in both the average and worst cases. We expect the losses considered in this section to be exceeded by the gains provided by these decreased execution requirements.

7.3 Experimental Results

In this section, we present the results of an experimental evaluation of staggering and of the proposed scheduling algorithm.

7.3.1 Simulations

In this subsection, we present a simulation-based comparison of the Pfair models. The advantage of simulation over direct measurement is that the processor count can be varied and the cache characteristics can be set arbitrarily. The latter trait was used to test performance on a simple blocking cache.

Experimental setup. These experiments used the Limes [37] simulator. Limes simulates execution close to the hardware level. As such, it provides no process management. However, Limes permits hardware-level aspects of the system to be more easily controlled than more complex simulators. Due to the limitations of Limes, preemption effects were only simulated. Specifically, caches were initially filled with dirty lines. Each task then performed writes to a local array until a "preemption" was detected. Tasks reacted to preemptions by exchanging their arrays for other uncached local ones. Hence, the cold-cache effect of preemptions was simulated using working-set changes.

We considered processor counts in the range $1, \ldots, 16$ and reported behavior across three slots. The system consisted of 200 MHz i86 processors using a 10 millisecond quantum. Caching consisted of blocking, direct-mapped caches with 256 KB capacity and 32-byte lines. The MESI coherency protocol[11] [52] was used. Since our goal was to measure only preemption-related contention, both actual and false data sharing was avoided.[12]

Bus contention was measured by counting pending bus requests at each cycle. Since only one request could be serviced in each cycle, the presence of i requests implied that $i-1$ tasks wasted that cycle waiting for bus access. Cycles required to service a task's bus requests were *not* counted as wasted cycles.

Relevance. These simulations focused on simultaneously scheduled tasks that process large data sets. Consideration of this scenario is motivated by the fact that many real-time multiprocessor systems consist of significant numbers of such tasks. Signal-processing and virtual-reality systems are two examples. Hence, we believe that these scenarios do represent situations that can arise in practice.

Worst-case contention under the aligned model. The first experiment estimated the worst-case bus contention under the aligned model (when working sets are fully cacheable). (This experiment does *not* characterize the worst case for staggering.) The worst case occurs when each task writes to each cache line in its working set at the start of each quantum.

[11]Also known as the Illinois protocol. MESI is an acronym based on the four possible states of a cache line: modified, exclusive, shared, and invalid.

[12]False sharing occurs when two or more tasks access a common cache line, but do not share any data.

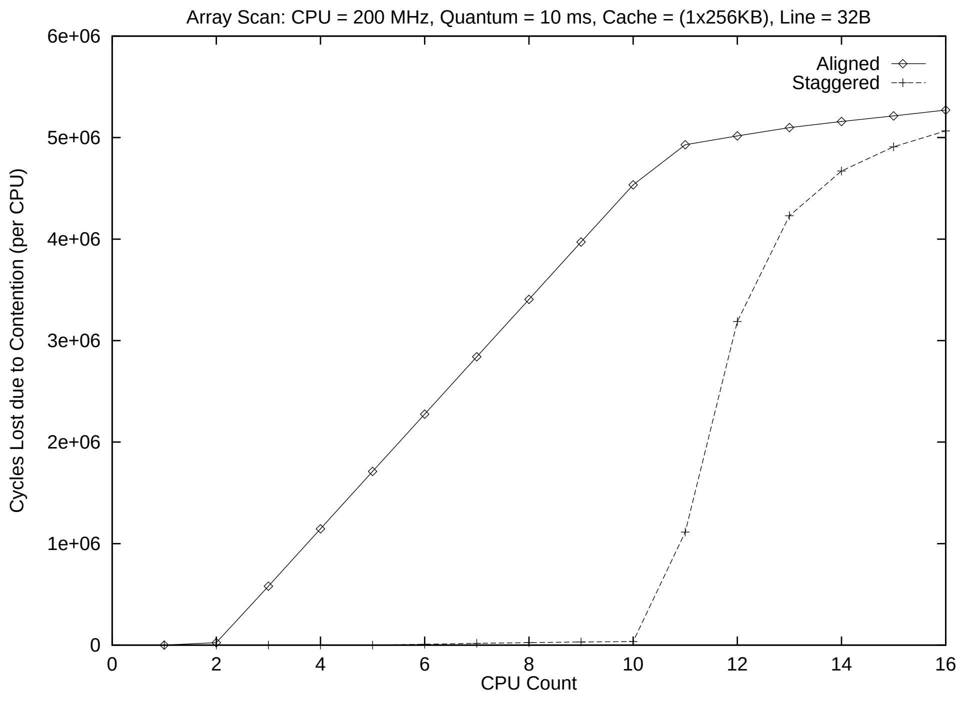


Figure 7.10: Cycles lost to bus contention under each of the aligned and staggered models. Each working set completely fills the cache and lines are systematically written.

Figure 7.10 shows the average number of cycles lost per task.

Notice that both curves converge as the processor count increases; this indicates an overload of the bus. When tasks fail to completely load their working sets within a single quantum, the resulting traffic pattern is approximately uniform across every quantum. As a result, staggering provides no benefit. Hence, increasing the volume of bus traffic must eventually cause performance under both models to converge.

Random-access contention. Our second experiment measured performance under a random-access pattern when working-set sizes vary. Each task randomly selected cells to access from a fraction α of the full array. This behavior results in a burst of bus traffic at the start of each slot, followed by a gradual decline as the probability of referencing an uncached line decreases. The value of α was chosen from $\{\ 0.2k \mid 1 \leq k \leq 5\ \}$.

Results are shown in Figure 7.11. As shown, staggering produces significantly less loss in all cases. This experiment was repeated several times, producing virtually identical results. (These simulations were unfortunately too long to produce confidence intervals.)

7.3.2 Prototype Measurements

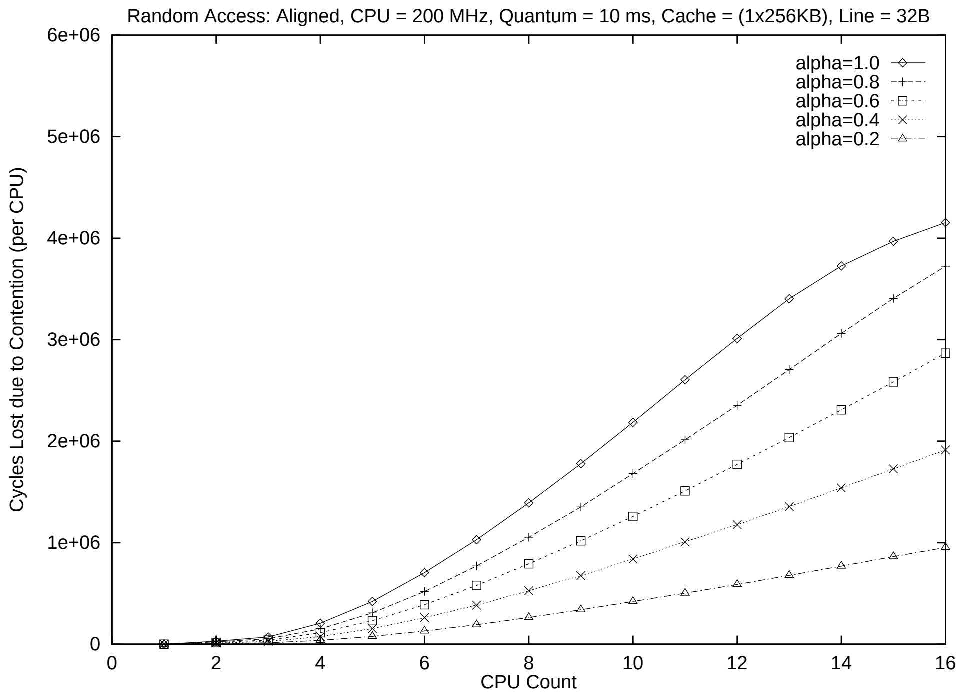
In this subsection, we present a comparison of the staggered and aligned models using a simple prototype of a Pfair system.

Experimental setup. The prototype microkernel executes as a thread package within QNX Neutrino 6.2.[13] The system consisted of four 200 MHz Pentium Pro processors, each of which had a 4-way, 512 KB L2 cache. These processors provide several latency-hiding features, including out-of-order execution, branch prediction, non-blocking caches, and support for multiple pending bus operations. Hence, this experiment will demonstrate whether staggering can improve upon simply applying common hardware-based techniques. Staggering should provide a much greater benefit to systems with fewer latency-hiding features.

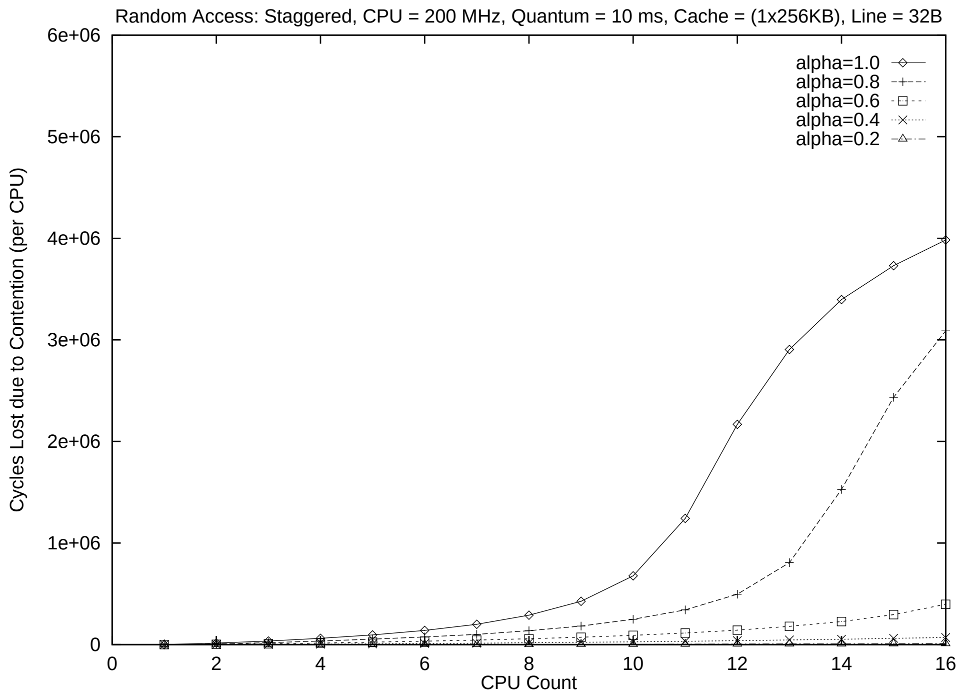
Both experiments described in the previous section were conducted on the prototype. Due to the hardware complexity, performance was measured at the user level by calculating the average number of cycles per write operation in each quantum.

Results. Figure 7.12 (respectively, Figure 7.13) shows the average number of cycles per write under the linear-access (respectively, random-access) reference pattern. 99% confidence intervals were computed, but are omitted due to scale. (Marked intervals show the observed sample range.) As shown, staggering provides an increasing improvement until the array size

[13]The prototype takes control of the system when running. The underlying kernel activates only to process timer interrupts and generate the signals that drive the prototype kernel. Neutrino was selected specifically to support this design.



(a) Aligned Model



(b) Staggered Model

Figure 7.11: Cycles lost to bus contention under each of the **(a)** aligned and **(b)** staggered models. Random writes are performed and working-set sizes are varied.

reaches approximately 150 KB, at which point overload occurs.

Figure 7.14 shows the ratios of corresponding sample means from the previous graphs. As shown, up to 7 (respectively, 2.5) times more writes were performed under the staggered model with the linear-access (respectively, random-access) pattern. Recall that this comparison is on a platform *with* latency-hiding features: improvement should be more dramatic without such features.

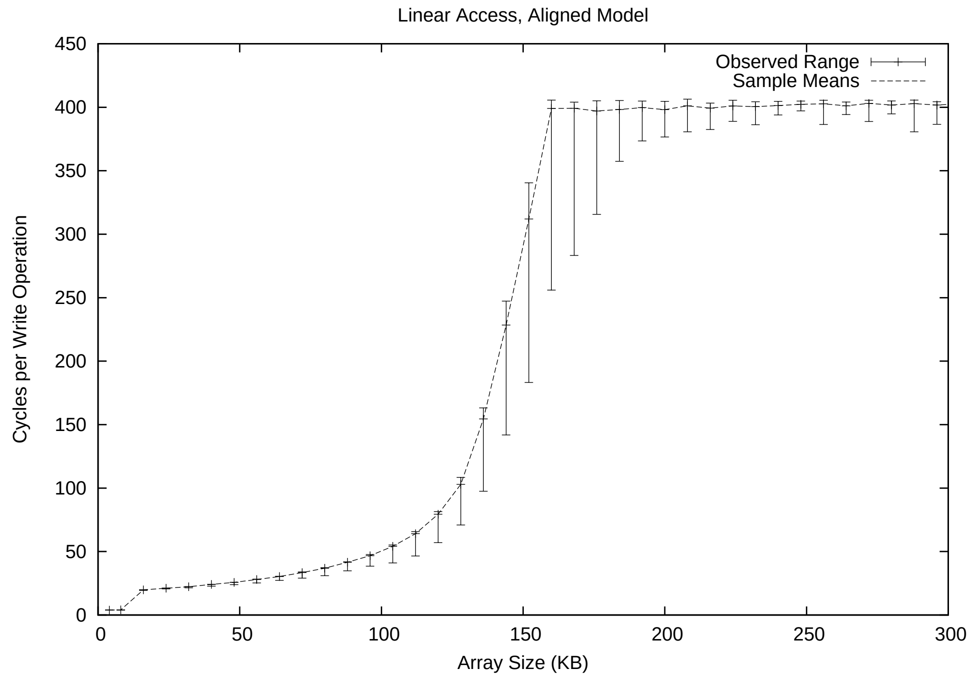
7.3.3 Scheduling Overhead

In the final series of experiments, the per-slot scheduling overhead of our staggered algorithm was compared to that of the master/slave PD^2 algorithm from [5].

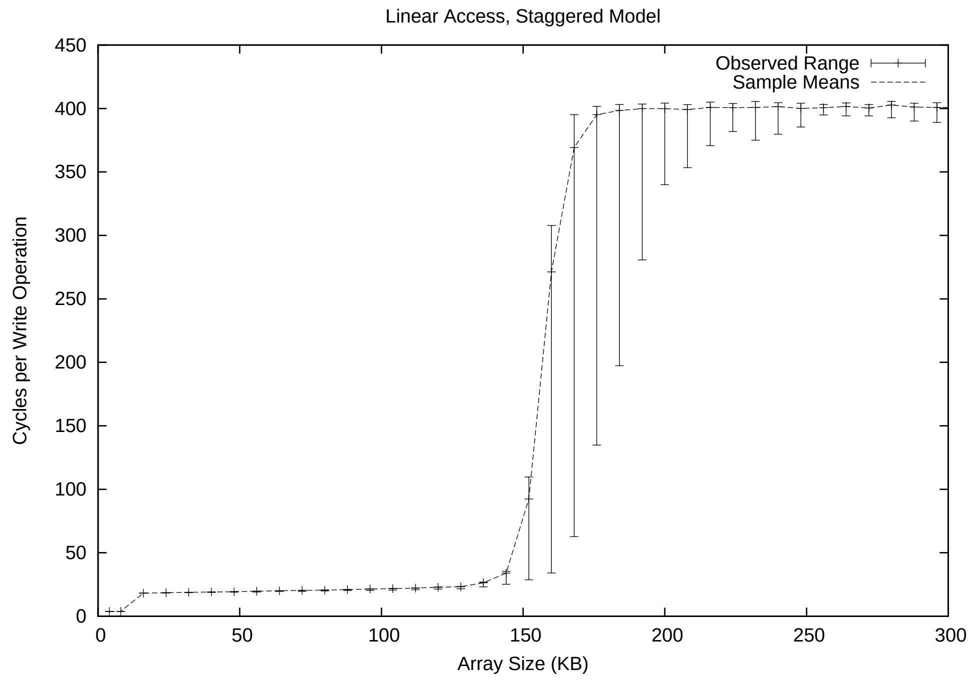
Experimental setup. Each experiment described below tested 1,000 randomly generated sets of independent tasks for each pairing of $|\tau| \in \{\ 10n \mid 1 \leq n \leq 50\ \}$ and $M \in \{2, 4, 8, 16\}$. From the execution-time measurements, the ratio of the average per-slot overhead of the master/slave algorithm to that of the staggered algorithm was computed. Again, 99% confidence intervals were computed, but are omitted due to scale.

Warm cache. In the first experiment, we considered performance when scheduler invocations are performed iteratively on a uniprocessor (a 700-MHz Dell PC running Red Hat Linux 2.4). After a warm-up delay, all memory references hit in cache. This experiment approximates the best-case performance of each algorithm. Architectures with highly effective latency-hiding features should provide comparable performance on average. Figure 7.15(a) shows the results from this experiment. As shown, the staggered algorithm approaches, and often matches, the factor-of-M improvement suggested by its time complexity.

Cold cache. Due to idealized cache behavior, the previous experiment provides no insight

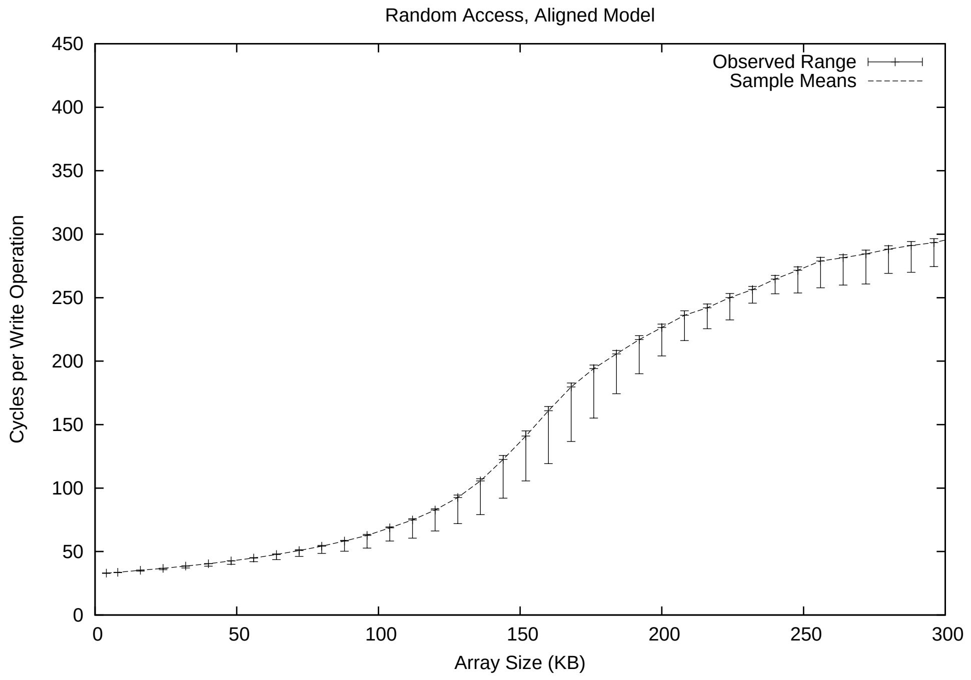


(a) Aligned Model

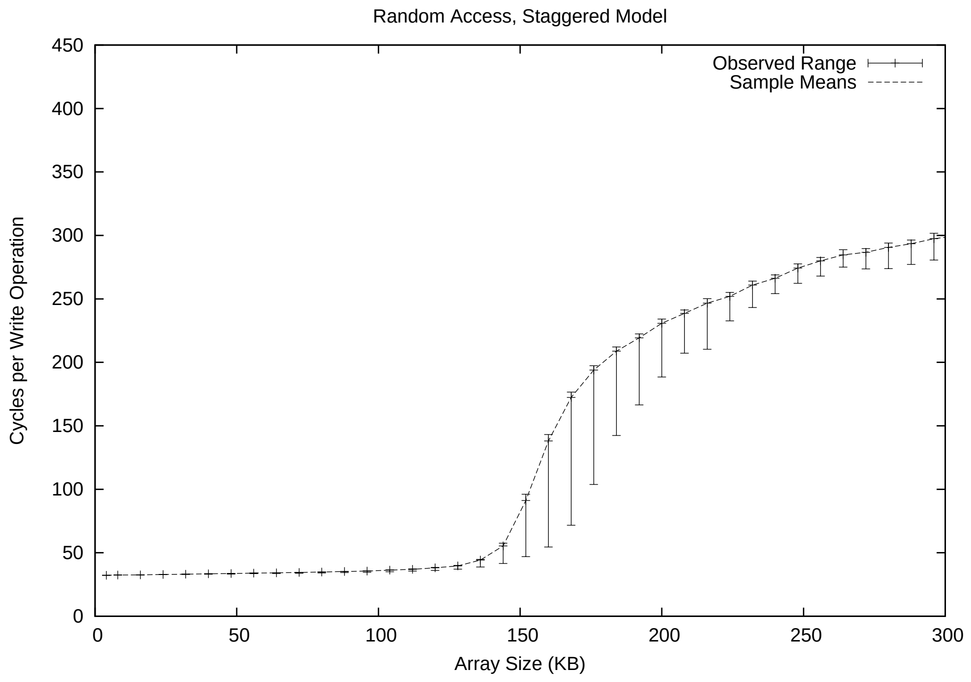


(b) Staggered Model

Figure 7.12: Results of prototype linear-access experiment for the **(a)** aligned and **(b)** staggered models.

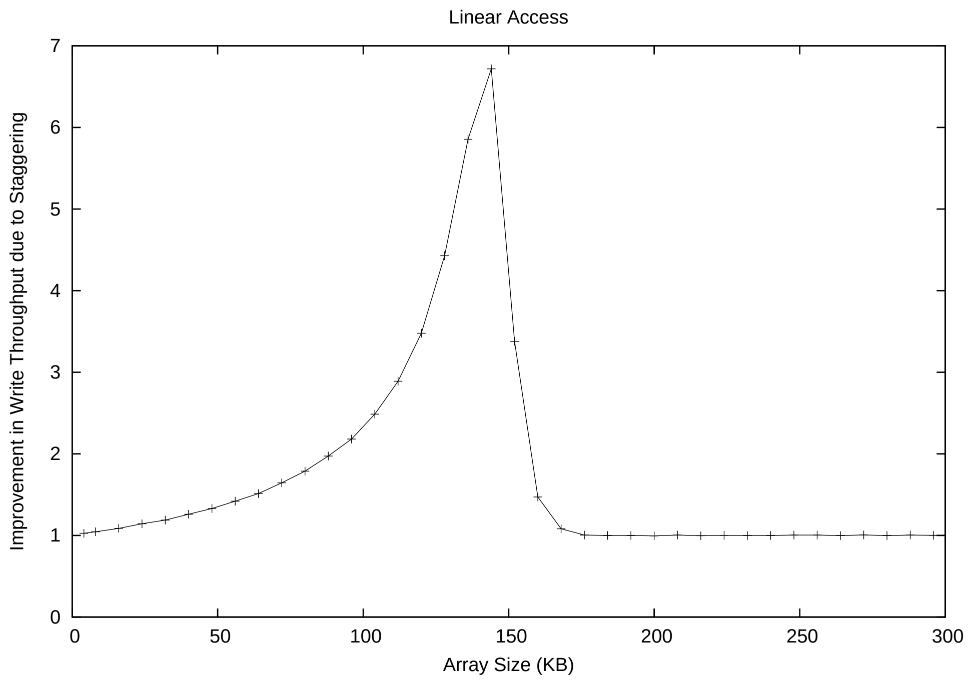


(a) Aligned Model

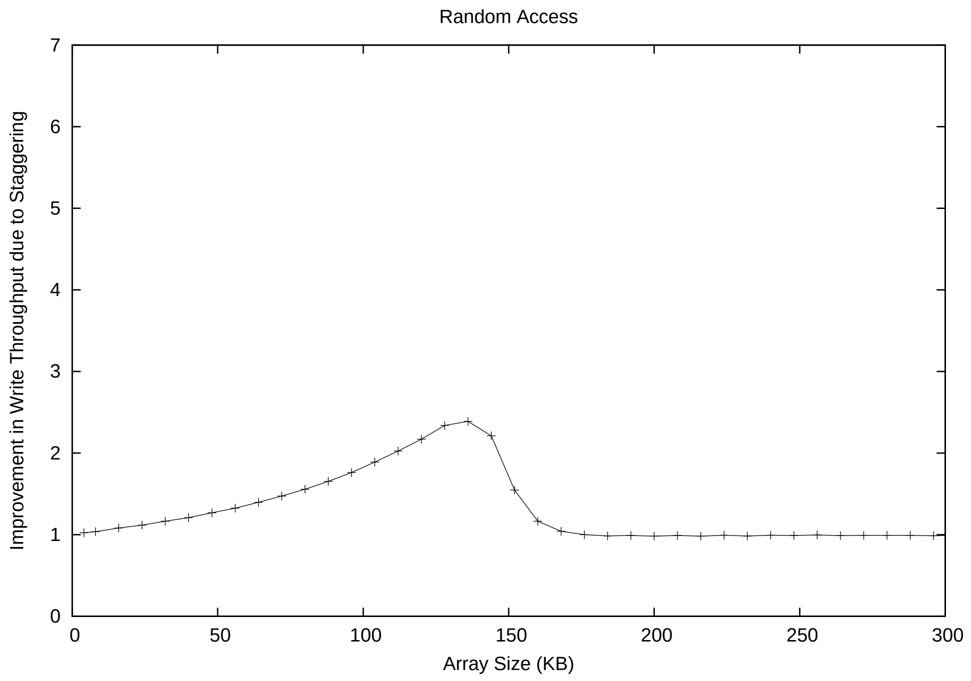


(b) Staggered Model

Figure 7.13: Results of prototype random-access experiment for the **(a)** aligned and **(b)** staggered models.

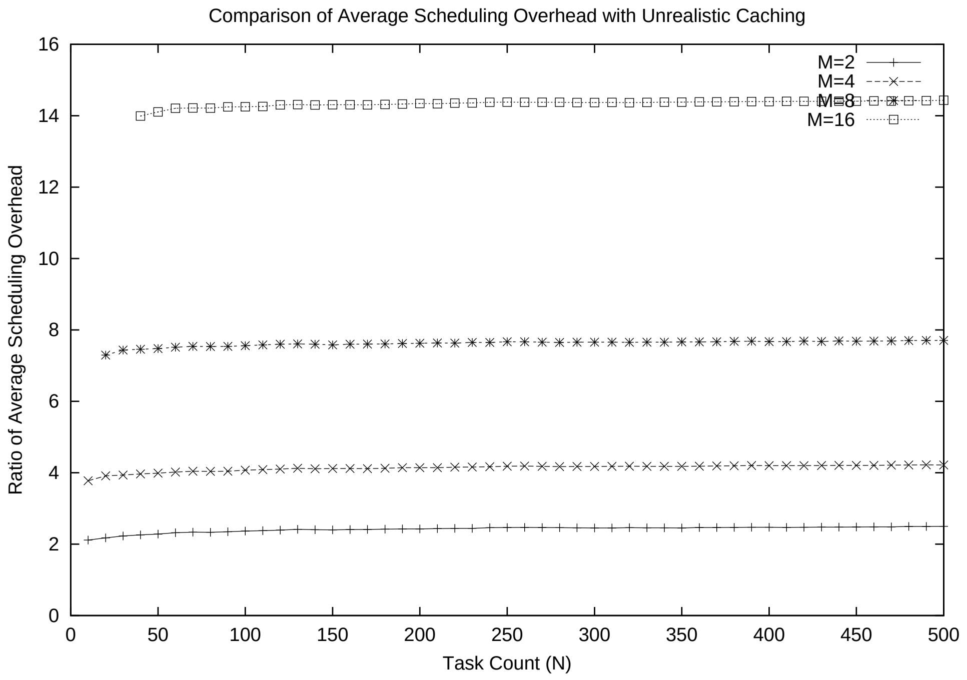


(a) Linear Access

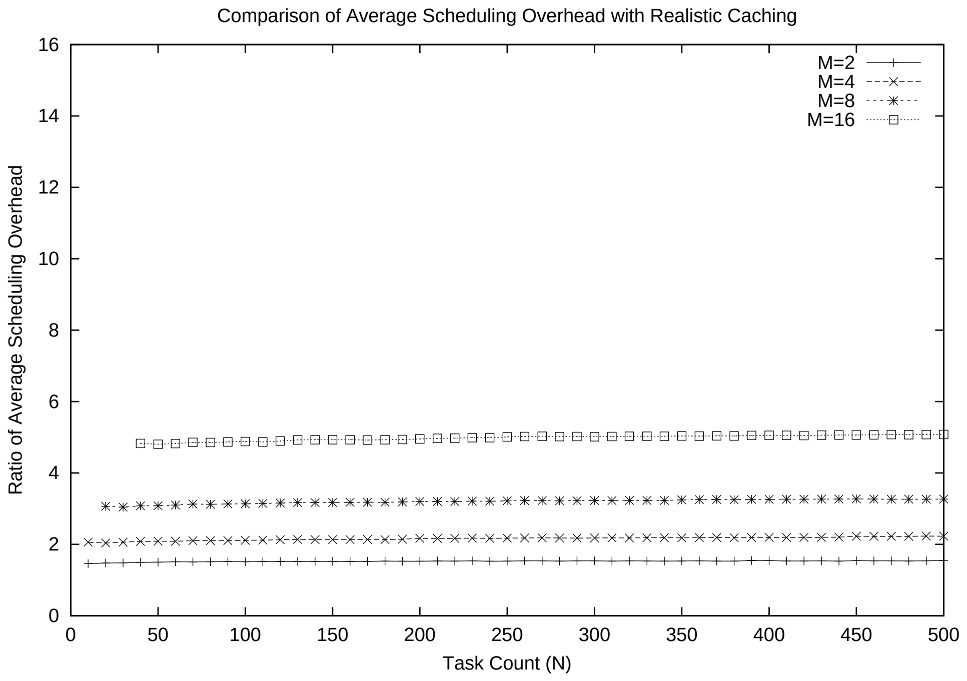


(b) Random Access

Figure 7.14: Ratios of write throughput under staggering to that under the aligned model for the **(a)** linear- and **(b)** random-access prototype experiments.



(a) Ideal Latency Hiding



(b) Realistic Caching

Figure 7.15: Ratio of the average per-slot scheduling overhead of master/slave scheduling to that of staggered scheduling. Measurements reflect performance **(a)** when all memory latency is hidden, and **(b)** when realistic caching is assumed.

into how the algorithms perform on simpler architectures or in worst-case situations. To provide more realistic estimates, we performed a second comparison on an SGI Reality Monster with 32 300-MHz R10000 processors in which multiple copies of the scheduler were distributed on the processors. Control was then transfered between these copies at appropriate points so that the scheduler would encounter a (somewhat) cold cache.[14] Under master/slave (respectively, staggered) scheduling, these transfers occurred after each scheduler (respectively, `Schedule`) invocation. Figure 7.15(b) shows the results from this experiment. Caching effects close the performance gap substantially compared to the previous experiment. Despite this, staggering still provides a significant improvement. Indeed, since each invocation of `Schedule` makes fewer memory references than the master/slave algorithm, staggered scheduling should *never* produce more overhead, regardless of the platform. However, the magnitude of the improvement may vary significantly, as these experiments suggest.

7.4 Summary

In this chapter, we discussed architectural concerns that arise when deploying Pfair scheduling on an SMP. Although SMPs are well-suited to Pfair scheduling in many ways, experimental results presented herein suggest that preemption-related bus contention can significantly degrade performance. To address this problem, we proposed and demonstrated the effectiveness of a staggered model under which preemption-related bus traffic is more evenly distributed over time. We also developed and experimentally evaluated an efficient scheduling algorithm to support this model that also produces less scheduling overhead than current Pfair algorithms. To facilitate the use of this new model, we explained how prior results, including those

[14]There is no way to accurately predict the cache states that a scheduler will encounter in a real system. We chose this approach because it seemed reasonable and straightforward to implement.

presented in earlier chapter, can be modified to support the proposed model. We concluded the chapter by presenting an experimental evaluation of both the staggered model and the proposed scheduling algorithm.

CHAPTER 8

Conclusions and Future Work

Prior work on Pfair scheduling has suggested many advantages over alternative approaches [2, 5, 6, 7, 10, 12, 18, 22, 48, 58, 61, 62, 63, 64, 65]. The most striking by far is that Pfair scheduling is capable of optimally scheduling recurrent and rate-based tasks on a multiprocessor. The strong notion of scheduling fairness provides an additional benefit in that scheduling is highly predictable. Indeed, we expect this to be the case even under overload conditions. For real-time systems, this predictability translates into temporal-isolation guarantees that are essential when multiplexing independently developed applications on a multiprocessor.

Unfortunately, prior work has limited attention to independent tasks and idealized hardware. Consequently, many open questions remained as to the practicality of Pfairness. In this dissertation, our goal was to address questions concerning resource sharing and common system overheads. In this chapter, we summarize the results presented in the previous chapters. Following this summary, we discuss directions for future work.

8.1 Summary

Chapter 3 outlined the design of a Pfair system. First, we considered the problem of implementing quantum-based scheduling and its implications to the assumptions underlying

Pfair analysis. Included in this discussion was the impact of overhead introduced by context switching, scheduling, interrupt handling, signal delay, and data sharing within the operating system. Second, we derived basic formulas that characterize relaxed Pfair scheduling, including formulas that map the requirements of independent periodic and sporadic tasks onto weights. Overhead resulting from the use of such mappings was included in this discussion. Third, we discussed the use of server tasks under Pfair scheduling and proved basic properties of such tasks. To support servicing of one-shot tasks with deadlines, we also presented a simple on-line acceptance test.

Chapter 4 extended prior supertasking work by Moir and Ramamurthy [48]. Specifically, we presented a flexible analytical framework that supports the use of supertasking under a wide variety of scenarios. We focused on two forms of supertasking: quantum-based and fully preemptive. Quantum-based supertasking can be used to selectively prevent groups of tasks (*i.e.*, the component tasks) from executing in parallel while respecting quantum-based scheduling. Such restrictions can be used not only to limit the number of migrations, as suggested by Moir and Ramamurthy, but also to reduce the worst-case resource contention. (This latter use was discussed in Chapters 5 and 6.) Fully preemptive supertasking, on the other hand, provides a scheduling environment that resembles that found in traditional uniprocessor systems. Tasks that are ill-suited to quantum-based scheduling can be more efficiently handled by scheduling them as a group within such an environment. In addition, such environments permit the use of uniprocessor techniques and protocols designed for fully preemptive scheduling, such as the PCP. (For example, we considered the use of the PCP under these conditions in the last experiment of Chapter 6.) To enable the use of both forms of supertasking, we presented a flexible reweighting algorithm that can be used both to assign weights to supertasks and also to determine whether a given weight is safe. This work included a discussion of reweighting overhead, which results from the need to use inflated supertask weights in order to ensure the timeliness of component tasks.

In Chapter 5, we considered the use of lock-free algorithms under Pfair scheduling. Such algorithms are well-suited to quantum-based scheduling because they avoid blocking and hence suspensions. Instead, these algorithms retry failed operations until successful. Although this approach is typically considered impractical on multiprocessors, we showed how Pfair's inherent structure facilitates the derivation of a bound on the number of retries performed by an operation. We also showed how quantum-based supertasking can be used to reduce lock-free overhead. Under special conditions, this approach can even enable the use of more efficient uniprocessor, wait-free object implementations.

Locking synchronization was the focus of Chapter 6. First, we considered several forms of inheritance that can easily be supported under Pfair scheduling. (Since inheritance is the predominant approach taken on uniprocessors, this was the obvious starting point.) Unfortunately, we noted several limitations of such protocols; most problematic among these was the interdependence between task weights and worst-case blocking durations. Next, we presented two zone-based protocols (*i.e.*, the Skip and Rollback Protocols). These protocols exploit Pfair's quantum-based structure to prevent critical sections from being preempted. This approach was motivated by the observation that preemption of a lock-holding task effectively extends the duration of the associated critical section. Empirical data presented in that chapter suggests that these protocols perform quite well. Unfortunately, the zone-based approach is only applicable when critical-section durations are shorter than Q. To support longer critical sections, we investigated the use of a simple server protocol. Although this approach provides only mediocre performance, it is easy to use and avoids many of the complications that can arise from more complex protocols. In addition, this approach was found to occasionally outperform the zone-based approaches in random task-set experiments.

In Chapter 7, we turned our attention to the issue of deployment. Based on the characteristics of Pfair scheduling, SMP-based architectures seem to be ideal candidates for the use of Pfair scheduling. Unfortunately, we noted an incompatibility between the quantum-based model of Pfair scheduling and the use of a shared memory bus. Specifically, tasks tend to produce high volumes of bus traffic while reloading data into the local processor cache at the start of each quantum. Due to the alignment of slots under Pfair scheduling, heavy contention arises during this reload period. To more evenly distribute preemption-related bus traffic, we proposed a new staggered model for Pfair scheduling under which slots do not align across processors. This new model provides an additional benefit in that it permits scheduling overhead to be more evenly distributed across processors. To enable the use of this new model, we proposed an efficient distributed scheduling algorithm and explained how results derived from the aligned model can be adapted to the staggered model. Finally, we presented empirical data to show the improvement that can be obtained by using the proposed model.

8.2 Future Work

We now describe a few of the several challenges remaining in this research area. Indeed, there are too many remaining challenges to list them all here. For this reason, we restrict attention to challenges that are closely related to the work presented in this dissertation.

8.2.1 Optimization Problems

Two optimization problems are apparent from the results presented earlier. First, the slot size impacts several overheads differently. Extremely small values are impractical due to scheduling overhead. Similarly, extremely large values are likely to produce excessive loss due to partially used quanta. Selecting an optimal slot size given the constraints of the system is undoubtedly a non-trivial problem. Second, supertasking provides many potential benefits, but at the cost of reweighting overhead. To determine the optimal system configuration, these benefits and the reweighting overhead must be quantified and related. This daunting task is further complicated by the fact that some benefits are difficult to quantify, such as the algorithmic benefits obtained by using lock-free synchronization with supertasking.

8.2.2 Variable-size Quanta

Under proportional-share scheduling [66], partially used quanta are truncated to avoid idling the processor. This action provides two benefits: tasks are not penalized for yielding within a quantum and a new quantum can be immediately started. Such behavior is an interesting alternative to the strict form of quantum-based scheduling considered in this dissertation.

Unfortunately, direct incorporation of this behavior under Pfair scheduling sacrifices optimality, as illustrated by Figure 8.1. This figure shows a system in which tasks are still charged for unused processor time,[1] but a new quantum is allowed to begin immediately after the executing task yields. In the first slot, both scheduled tasks yield early, causing the processors to be immediately granted to the remaining two tasks. As a result, the unit-weight task misses its second deadline. This scenario can easily be generalized to an arbitrary number of

[1]Not charging tasks for unused processor time is a more complex issue since Pfair scheduling does not support requests for fractional quanta. Adding this capability is another interesting direction for future work.

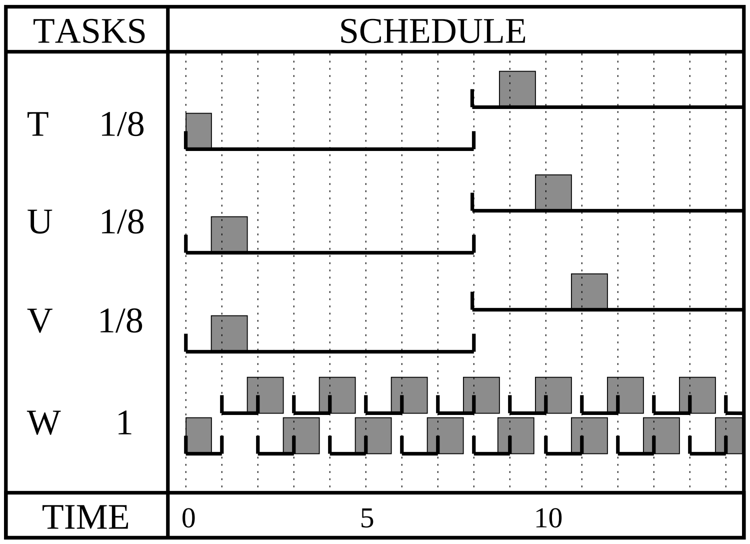


Figure 8.1: Counterexample that shows how the use of variable-length quanta leads to deadline misses. Four tasks are scheduled on two processors. In the first slot, both tasks T and W finish early. The next quanta begin immediately and are granted to U and V. As a result, W misses its next deadline.

processors by using one task with unit weight and $2M - 1$ tasks with weight $\frac{1}{k}$. Notice that the total weight of such a task set approaches unity as $k \to \infty$.

Indeed, it may not be possible to support variable-size quanta in an optimal scheduler. Proportional-share scheduling only guarantees a deadline-miss threshold of one quantum. Interestingly, the counterexample shown above does not preclude the existence of a similar guarantee for variants of Pfair scheduling that support the use of variable-size quanta.

8.2.3 Overload Guarantees

Because Pfairness ensures proportional allocation, the share of the system guaranteed to each task should always be predictable, even under overload conditions. However, overload has the effect of changing the interpretation of task weights. Under normal conditions, weights represent a fixed fraction of a single processor to which the associated task is entitled. Under overload conditions, weights effectively become relative measures of the task's share. For instance, a task with weight w in an overloaded system of total weight W should still receive approximately a fraction $\frac{w}{W} \cdot M$ of a processor. Hence, a general formula for the fraction of a processor to which a task is entitled is given by $w \cdot \min\left(1, \frac{M}{W}\right)$. (A system is overloaded when $\frac{M}{W} < 1$, fully loaded when $\frac{M}{W} = 1$, and underloaded when $\frac{M}{W} > 1$.)

Providing bounds on each task's guaranteed share during overload is of interest in at least two situations. First, transient processor failures may result in temporary periods of overload. For systems in which such failures are a concern, the impact of an overload must be determined in order to verify that fault-tolerance requirements will always be met. This concern highlights another interesting problem. It is possible that Pfair scheduling is self-stabilizing, *i.e.*, a system that is normally underloaded may, given sufficient time, return to a state in which no deadlines are being missed following a transient period of overload.

Second, when scheduling soft-real-time tasks, weights can be assigned according to the desired average-case performance. When the total weight of the task set exceeds the number of processors, the system must either reduce weights or continue to operate under overload. The former option is undesirable since updating weights and priorities at runtime can introduce substantial overhead [2, 63]. However, it is presently unclear whether using a single weight with each task is desirable when working with soft-real-time tasks. The work of Anderson, Block, and Srinivasan [2] on QRfair scheduling suggests that two weights, which represent the minimum and maximum share needed by each task, may provide more predictable performance without introducing significantly more overhead.[2] When using only one weight per task, understanding the impact of overload would be invaluable in soft real-time systems.

[2]To the best of our knowledge, the use of multiple weights per task is a largely unexplored approach. Hence, this would also be an interesting topic for future work in both the hard and soft real-time areas.

8.2.4 Supertasking

There are two obvious directions for future work in this area. First, all prior work on supertasking has focused on the use of either an Idle-and-Wait or a Drop-and-Wait server policy. One benefit of these policies is that analysis is straightforward. However, it is possible that a more elaborate server policy may achieve a significant reduction in reweighting overhead. Hence, consideration of other server policies is one obvious direction of future work.

Second, the experimental studies in Chapters 5 and 6 suggest that a substantial reduction in synchronization overhead can be obtained by using supertasks to restrict the parallel execution of tasks that share resources. However, these studies consider only the use of static supertasks, *i.e.*, supertasks in which the component task set is fixed at runtime. An alternative approach that may reduce synchronization overhead even further is to associate each lock or lock-free object with a specific supertask and then require tasks to migrate[3] into that supertask before accessing the shared object. In this way, synchronization overhead would be minimized. However, the migration of a task may need to be delayed in order to prevent timing violations within the supertask. Hence, this approach trades synchronization overhead for overhead resulting from migration delay.

To support intersupertask migration, rules are needed to determine when a task may safely leave and join a supertask. In addition, it may also be desirable to change the weight of each supertask on-the-fly to prevent it from consuming an unnecessarily large fraction of the processor time. Hence, rules for safely changing the supertask's weight may also be needed.

Indeed, this last observation raises an additional issue: How can a supertask be reweighted at runtime? There are two obvious solutions. The first would be to utilize the reweighting

[3]Here, migration refers to intersupertask migration, and not to interprocessor migration.

algorithm from Chapter 4. The input parameters provided by the algorithm can be used to control the duration of reweighting. The disadvantage of this approach is that the selected weight may be very pessimistic. Alternatively, each such supertask could be restricted to take on one of a discrete set of weights. For instance, a supertask's weight could be constrained to be in the set $\{ 0.2, 0.4, 0.6, 0.8, 1.0 \}$. In this case, rules for handling each allowed weight to be derived *a priori*, thereby avoiding the need for on-the-fly reweighting.

8.2.5 Synchronization

Several challenges remain in the area of synchronization. For instance, efficient I/O support is needed. I/O-based critical sections differ from software-based critical sections in two ways. First, these critical sections are typically much longer, particularly when mechanical devices like disk drives are involved. Second, these critical sections do not require processor occupancy. Indeed, processor time should *not* be granted to lock-holding tasks performing I/O-based critical sections. Since this last characteristic also describes server-based synchronization, techniques designed to improve I/O handling can likely be applied in that context as well.

The techniques presented in this dissertation address only sharing between persistent real-time tasks. In real systems, resources may be shared by hard-real-time, soft-real-time, non-real-time, and one-shot tasks. Since the runtime behavior of many of these tasks will not be known *a priori* or even be predictable, off-line analysis like that presented for locking protocols may not be possible. For instance, Real-time Java (RTJ) is required to support communication between real-time and non-real-time tasks [14], despite the fact that the behavior of non-real-time tasks is not predictable. To resolve this problem, RTJ requires the use of wait-free queues as communication channels between such tasks. This choice sacrifices generality and design flexibility in order to guarantee timeliness.

Finally, this dissertation focuses on non-nested critical sections. Nested critical sections are undesirable because they greatly complicate analysis, particularly on multiprocessors [54]. However, they may be unavoidable in some cases, and hence should be supported.

8.2.6 Interrupt Handling

Although a few simplistic techniques like that given in Chapter 3 have been proposed to account for interrupt overhead, it is unclear whether these approaches are efficient or even reasonable. For instance, trying the model interrupts as periodic or sporadic events may not be appropriate for some interrupts. As with tasks, it is unlikely that a single model can effectively describe all interrupts. To enable proper analysis, more extensive studies of real systems are needed to provide models for the different components of interrupt handling and baseline measurements for the model parameters. Such information is a necessary first step toward providing adequate handling for interrupts.

8.2.7 Other Services

Systems consist of more than scheduling and synchronization services. Applications require network access, file systems, interactive shells, *etc.* Support for such services is key to the creation of a full-featured Pfair system. Although many approaches to providing these services have been proposed, few (if any) are directly applicable under Pfair scheduling.

8.2.8 Implementation

Conventional event-driven schedulers can usually be tested by replacing the schedulers in conventional operating systems, such as Linux or FreeBSD. However, due to its time-driven nature, forcing a Pfair scheduler into a conventional system will almost certainly produce both poor performance and measurements that are not reflective of a from-scratch implementation. Accurate assessment is essential to making an unbiased comparison to partitioning and other alternative approaches. Hence, mechanisms that exploit strengths and compensate for weaknesses are an important factor in accurately evaluating the worth of each tested approach. Unfortunately, many such mechanisms are yet to be developed for Pfair scheduling, as is likely apparent from the previous sections.

Although a full-featured Pfair system cannot be implemented at present, a Pfair microkernel can be. Indeed, we developed a microkernel with limited functionality as a thread package for the experiments presented in Chapter 7. For a comparison against partitioning, a from-scratch Pfair prototype that executes directly on the hardware would be preferable. Such a prototype is also essential to measuring the performance and implementation overhead of the various extensions that have been and will be proposed for Pfair scheduling.

APPENDIX A

Correctness Proof

In this appendix, we present a proof of correctness for the multiprocessor lock-free queue implementation shown in Figure A.1. A slightly more complicated version of this algorithm appeared previously in the case study presented in Chapter 5.

A.1 Design Assumptions

The version of the algorithm considered here is based on the following (unrealistic) assumptions, which are intended to simplify the proof:

(A1) Each tag field has an unbounded range.

(A2) Each queue node is used only once.

In addition, rather than re-using the local variable x, this version of the algorithm uses extra variables y and z. Avoiding the re-use of variables and nodes simplifies the proofs without changing the nature of the algorithm. Finally, the value field of a queue node has been renamed from *value* to *elem* to avoid confusion with the *value* field of a tagged variable.

Due to (A2), the enqueue and dequeue algorithms shown in Figure A.1 accept and return, respectively, the element type used by the queue, rather queue nodes. (The version presented earlier in Chapter 5 handles nodes instead.) In addition, the special symbol $\perp$ has been introduced at line 15 to signal an empty queue.

```
template tagged(T):
    record
        tag: integer;
        value: T

private var
    x, y, z, h, t: tagged(pointer to node);
    done: boolean; out, in: element;
    n: pointer to node

procedure Enqueue_MW(in)
1:  n := MakeNode();
2:  *n:= (in, (0, nil));
    do
3:      done := false;
4:      t := Tail;
5:      x := *(t.value).next;
6:      if t = Tail then
7:          done := CAS(&(*(t.value).next),
                        (x.tag,nil), (x.tag+1,n));
8:          y := *(t.value).next;
9:          CAS(&Tail, t, (t.tag+1,y.value))
        fi
10: while ¬done
```

```
typedef node:
    record
        elem: element;
        next: tagged(pointer to node)

shared var
    Head, Tail: tagged(pointer to node)

procedure Dequeue_MR() returns element
    do
11:     done := false;
12:     h := Head;
13:     if h.value = Tail.value then
14:         if h = Head then
15:             return ⊥
            fi
        else
16:         z := *(h.value).next;
17:         if z.value ≠ nil then
18:             out := *(z.value).elem;
19:             done := CAS(&Head, h,
                            (h.tag+1,z.value))
            fi
        fi
20: while ¬done;
21: return out
```

Figure A.1: Simplified code listing for multiprocessor (lock-free) shared queue.

A.2 Notation

In this section, we summarize notation that will be used when discussing the algorithm and its correctness. An enqueuing (respectively, dequeuing) task will be denoted by either e or e' (respectively, d or d'). The notation $T.x$ will be used to denote the local variable x of task T. The condition $T@k$ holds if and only if task T will execute line k next. Similarly, $T@S$ holds if and only if $T@k$ holds for some $k \in S$. The notation $n.T$ will be used to specify the execution of line n by task T.

A.3 Operational Assumptions

The queue is assumed to be initialized as follows:

(I1) The dummy node D is initialized so that $D.next = (0, nil)$.[1]

(I2) $Head = Tail = (0, \&D)$.

(I3) $e@1$ holds for each enqueuer e.

(I4) $d@11$ holds for each dequeuer d.

For simplicity, we assume that each enqueuer (respectively, dequeuer) performs enqueue (respectively, dequeue) operations in an unbounded loop. Hence, after executing line 10, each enqueuer is assumed to transition back to line 1. Similarly, after executing line 20, each dequeuer is assumed to transition back to line 11.

A.4 Basic Properties

The following properties are derived directly from the program code (as explained below) and will be used in the proof arguments that follow.

(P1) The node that is modified in line 2 is not shared (when line 2 is executed).

(P2) For any tagged variable v and copy[2] of that variable v', $v.tag \geq v'.tag$ holds.

Property P1 follows trivially from (A2). Specifically, since nodes are not re-used, each node is effectively private to the enqueuing task until the node is added to the data structure at line 7. The initialization of the node at line 2 occurs before this point.

[1]The initial value of $D.value$ is unimportant since it is never used.

[2]For this to be the case, the most recent write to v' must have assigned a value to v' that was returned by a previous read of v.

Property P2 follows trivially from the program code. Each update of a tagged variable increments its tag field. (Indeed, when initialized to zero, the tag effectively serves as a counter for the number of updates performed.) Since (A1) ensures that the range of each tag field is unbounded, all previous reads of a tagged variable must have returned tag values that are at most the current tag value. Property P2 follows.

A.5 Input Correctness

In this section, we derive properties of the algorithm to prove that enqueue operations are performed correctly. Enqueue operations are operationally more complex than dequeue operations because they require the modification of two shared variables. Consequently, the queue must temporarily be placed into an intermediate state during the interval of time spanning the first modification and the second.

A.5.1 Queue States and Safety Criteria

Figure A.2 shows the normal and intermediate (or *extended*) queue states. An enqueue operation first appends the new node onto the end of the queue, which places the queue in the extended state. This is accomplished by line 7. The *Tail* pointer is then updated to point to the appended node, thereby returning the queue to the normal state. This step is achieved by line 9. To ensure proper operation, we must ensure that line 9 can succeed only when the queue is in the extended state. Similarly, we must also ensure that line 7 can succeed only when the queue is in the normal state. This criteria is expressed formally by conditions (S1) and (S2), respectively, shown below.

(S1) $e@9 \;\wedge\; \mathit{Tail} = e.t \;\Rightarrow\; e.y = *(e.t.\mathit{value}).\mathit{next} \;\wedge\; e.y.\mathit{value} \neq \mathit{nil}$

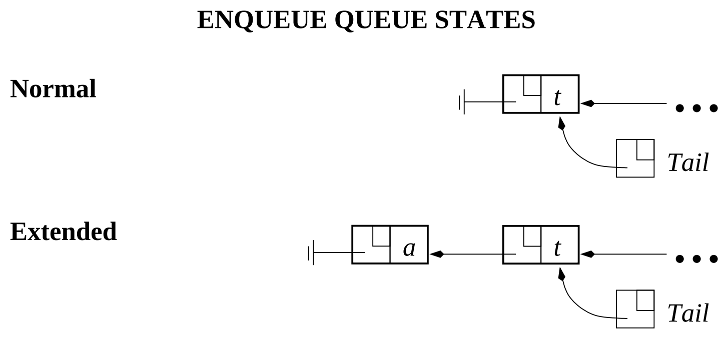


Figure A.2: Illustrations of the queue states that occur during an enqueue operation.

(S2) $e@7 \ \wedge \ *(e.t.value).next = (e.x.tag, nil) \ \Rightarrow \ e.x = *(e.t.value).next \ \wedge \ e.t = Tail$

Informally, (S1) states that if the `CAS` call at line 9 succeeds, then *Tail* is advanced to the node immediately after the previous tail node (first clause of the consequent) and such a successor node must exist (second clause of the consequent). The latter implication ensures that the queue is in the extended state whenever line 9 is successfully executed. Similarly, (S2) states that a successful execution of line 7 can only occur if $e.x$ is an up-to-date copy of $*(e.t.value).next$ (first clause of the consequent) and $e.t$ is an up-to-date copy of *Tail* (second clause of the consequent). By the second clause, $*(e.t.value).next$ is equivalent to $*(Tail.value).next$. The fact that this pointer is *nil* (by the second clause of the antecedent) implies that the queue is in the normal state whenever line 7 is successfully executed.

When proving invariants that do not depend on *Head*, dequeue operations can be ignored. This follows trivially from the fact that such operations only change the value of *Head*. Correctness of dequeue operations is the subject of the next section.

A.5.2 Proof of (S1)

In this section, we prove a series of invariants that lead up to the proof of (S1). Specifically, (S1) corresponds to A.4, which appears at the end of this section.

Invariant A.1 $e@\{5\ldots9\} \wedge\ e'@\{9\}\ \wedge\ e'.t = \mathit{Tail}$

$\Rightarrow\ e.t \neq (e'.t.tag + 1, e'.y.value)$

Proof. Property P2 implies that $e.t.tag \leq \mathit{Tail}.tag$. When the antecedent holds, it follows that $e.t.tag \leq e'.t.tag$ holds also. In this case, $e.t \neq (e'.t.tag + 1, e'.y.value)$ must also hold since $e.t.tag \leq e'.t.tag$ implies that $e.t.tag < e'.t.tag + 1$. □

Invariant A.2 $e@\{6\ldots7\}\ \wedge\ e.t = \mathit{Tail}\ \Rightarrow\ *(\mathit{Tail}.value).next.value \neq nil\ \vee\ e.x = *(\mathit{Tail}.value).next$

Proof. By (I3), $e@1$ holds initially, and thus A.2 is true. A.2 may be falsified only by $5.e$ (which establishes $e@\{6\ldots7\}$), $2.e'$ (which may change the value of $*(\mathit{Tail}.value).next$), $7.e'$ (which may change the value of $*(\mathit{Tail}.value).next$), and $9.e'$ (which may establish $e.t = \mathit{Tail}$). Here, e' refers to an enqueuer other than e.

Consider $5.e$. This line establishes the antecedent if and only if $e.t = \mathit{Tail}$ holds. When this is the case, the execution of $5.e$ establishes $e.x = *(\mathit{Tail}.value).next$ also. Thus, A.2 is maintained.

Consider $2.e'$. Property P1 implies that the node being initialized is effectively private when this line is executed. Hence, $\mathit{Tail}.value$ cannot be a pointer to that node, and thus A.2 remains true.

Consider $7.e'$. To modify the value of $*(\mathit{Tail}.value).next$, this line must also establish $*(\mathit{Tail}.value).next.value \neq nil$. Thus, A.2 remains true.

Consider $9.e'$. By A.1, the execution of this line can only falsify $e.t = \mathit{Tail}$. Therefore, A.2 remains true. $\square$

Invariant A.3 $e@\{8\ldots9\} \ \wedge\ e.t = \mathit{Tail} \ \Rightarrow\ *(\mathit{Tail.value}).\mathit{next.value} \neq \mathit{nil}$

Proof. By (I3), $e@1$ holds initially, and thus A.2 is true. A.2 may be falsified only by $7.e$ (which establishes $e@\{8\ldots9\}$), $2.e'$ (which may change the value of $*(\mathit{Tail.value}).\mathit{next}$), $7.e'$ (which may change the value of $*(\mathit{Tail.value}).\mathit{next}$), and $9.e'$ (which may establish $e.t = \mathit{Tail}$). Here, e' refers to an enqueuer other than e.

Consider $7.e$. This line establishes the antecedent if and only if $e.t = \mathit{Tail}$ holds. By A.2, $e.x = *(\mathit{Tail.value}).\mathit{next} \ \vee\ *(\mathit{Tail.value}).\mathit{next.value} \neq \mathit{nil}$ must hold prior to the execution of $7.e$ in this case. Hence, A.3 can be falsified only if $e.x = *(\mathit{Tail.value}).\mathit{next}$ holds. Assume (toward a contradiction) that this is the case. Since $e.t = \mathit{Tail}$ holds, $e.x = *(e.t.\mathit{value}).\mathit{next}$ must also hold in this case. If $*(e.t.\mathit{value}).\mathit{next.value} = \mathit{nil}$, then the `CAS` call will succeed and establish the consequent. Therefore, A.3 cannot be falsified when the `CAS` call succeeds. On the other hand, if $*(e.t.\mathit{value}).\mathit{next.value} \neq \mathit{nil}$, then the `CAS` call will fail. However, $*(\mathit{Tail.value}).\mathit{next.value} \neq \mathit{nil}$ must hold in this case both before and after the call (since $e.t = \mathit{Tail}$ holds). Therefore, A.3 cannot be falsified when the `CAS` call fails either. This contradicts our earlier assumption that $*(\mathit{Tail.value}).\mathit{next.value} \neq \mathit{nil}$ does not hold after the execution of $7.e$. Thus, A.3 is true.

By the same arguments given earlier in the proof of A.2, A.3 cannot be falsified by $2.e'$, $7.e'$, or $9.e'$. $\square$

Invariant A.4 $e@9 \ \wedge\ \mathit{Tail} = e.t \ \Rightarrow\ e.y = *(e.t.\mathit{value}).\mathit{next} \ \wedge\ e.y.\mathit{value} \neq \mathit{nil}$

Proof. By (I3), $e@1$ holds initially, and thus A.4 is true. A.4 may be falsified only by $8.e$ (which establishes $e@9$), $2.e'$ (which may change the value of $*(\mathit{Tail.value}).\mathit{next}$), $7.e'$ (which may change the value of $*(\mathit{Tail.value}).\mathit{next}$), and $9.e'$ (which may establish $e.t = \mathit{Tail}$). Here, e' refers to an enqueuer other than e.

Consider $8.e$. The execution of $8.e$ establishes $e.y = *(e.t.\mathit{value}).\mathit{next}$. To establish the antecedent, $e.t = \mathit{Tail}$ must also hold prior to the execution. By A.3, $*(\mathit{Tail.value}).\mathit{next.value} \neq \mathit{nil}$ holds prior to the execution of $8.e$. Therefore, $8.e$ also establishes $e.y.\mathit{value} \neq \mathit{nil}$ when the antecedent is established. Thus, A.4 is maintained.

Consider $7.e'$. Since this line cannot establish the antecedent, it can falsify A.4 only by falsifying the consequent when both the antecedent and consequent held prior to its execution. Assume (toward a contradiction) that this is the case. Since $7.e'$ cannot falsify the second clause of the consequent, it follows that $e.y \neq *(e.t.\mathit{value}).\mathit{next}$ was established by $7.e'$, and hence that the `CAS` call must have succeeded. However, since both the antecedent and consequent held prior to the execution of $7.e'$, $*(e.t.\mathit{value}).\mathit{next.value} \neq \mathit{nil}$ held also, which implies that the `CAS` call must have failed. Hence, we have reached a contradiction, which implies that $7.e'$ maintains A.4.

By the same arguments given earlier in the proof of A.2, A.4 cannot be falsified by $2.e'$ or $9.e'$. □

A.5.3 Proof of (S2)

In this section, we prove that (S2) is invariant.

Invariant A.5 $e@7 \wedge *(e.t.\mathit{value}).\mathit{next} = (e.x.\mathit{tag}, \mathit{nil}) \Rightarrow e.x = *(e.t.\mathit{value}).\mathit{next} \wedge e.t = \mathit{Tail}$

Proof. By (I3), $e@1$ holds initially, and thus A.5 is true. A.5 may be falsified only by $6.e$ (which may establish $e@7$), $2.e'$ (which may change the value of $*(e.t.value).next$), $7.e'$ (which may change the value of $*(e.t.value).next$), and $9.e'$ (which may establish $e.t \neq Tail$). Here, e' refers to an enqueuer other than e.

Consider $6.e$. For $6.e$ to establish the antecedent, both $e.t = Tail$ and $*(e.t.value).next = (e.x.tag, nil)$ must hold. Hence, $e.t = Tail$ holds when the antecedent is established, which by A.2, implies that $*(Tail.value).next.value \neq nil \ \vee \ e.x = *(Tail.value).next$ also holds. Since $*(e.t.value).next = (e.x.tag, nil)$ implies that $*(Tail.value).next.value = nil$ holds (since $e.t = Tail$ holds), it follows that $e.x = *(Tail.value).next$ holds. Hence, when the antecedent is established, $e.x = *(e.t.value).next$ also holds. Thus, A.5 is maintained.

Consider $7.e'$. If the value of $*(e.t.value).next$ is modified, then the *value* field must change to a non-*nil* value. Hence, the execution of this line cannot modify $*(e.t.value).next$ without also falsifying $*(e.t.value).next = (e.x.tag, nil)$. Thus, A.5 is maintained.

Consider $9.e'$. Since $9.e'$ cannot establish the antecedent, A.5 can be falsified only if both the antecedent and consequent hold prior to the execution of $9.e'$, and its execution establishes $e.t \neq Tail$. Assume (toward a contradiction) that this is the case. By A.4, the `CAS` call succeeds only if $e.y = *(e.t.value).next \ \wedge \ e.y.value \neq nil$ holds. Since $e.t = Tail$ holds prior to the execution of $9.e'$, it follows that $*(Tail.value).next.value \neq nil$ holds also. However, if $e.t = Tail$ and $*(e.t.value).next = (e.x.tag, nil)$ both hold prior to execution of $9.e'$, then so does $*(Tail.value).next.value = nil$. Hence, we have reached a contradiction, and thus A.5 is maintained by $9.e'$.

By the same argument given earlier in the proof of A.2, A.5 cannot be falsified by $2.e'$. □

A.6 Output Correctness

In this section, we argue the correctness of dequeue operations. Since these operations require the modification of only one variable, they are much simpler than enqueue operations. Because of this, we argue correctness more informally in this section by considering necessary conditions for a return at each of lines 15 and 19. Specifically, we argue that a return can occur from each line only when the conditions required to ensure correctness are met. The primary difficulty in this context lies in ensuring that an empty queue is properly detected and handled. Line 15 handles this case.

Dequeue operations use the *Head* pointer both to detect an empty queue and to synchronize concurrent dequeuers. Specifically, *Head* is set to point to a dummy head node, *i.e.*, a head node that does not contain a queue element. At any given instant, the elements considered to be in the queue are those in the chain of nodes ranging from the node *after* the dummy head node to the node referenced by *Tail*. (See Figure 5.4.) When this chain (and hence the queue) is empty, both *Head* and *Tail* reference the dummy node. Otherwise, the head element of the queue is stored in the node after the dummy node.

An element is claimed by the first dequeuer to advance the *Head* pointer to the node containing that element. This action has the effect of converting the element's node into the dummy node of the queue. (Hence, the identity of the dummy node changes as elements are dequeued.) By advancing *Head* using a `CAS` call (at line 19), we ensure that only one dequeuer can claim each element. The correctness of dequeue operations is discussed in detail in the sections that follow.

A.6.1 Detecting an Empty Queue

First, we argue the correctness of operations that return at line 15. These operations indicate that the queue was empty, and hence require that $Head.value = Tail.value$ held at some point during the operation. Hence, it must be shown that a return at line 15 implies that $Head.value = Tail.value$ held at some point between the execution of lines 11 and 15.

Suppose that line 15 is reached. By the comparison at line 14, *Head* did not change between lines 12 and 14. Consequently, the result of the comparison $h.value = Tail.value$ at line 13 was equivalent to that of $Head.value = Tail.value$. The correctness of the return at line 15 follows.

A.6.2 Dequeuing an Element

We now argue the correctness of operations that return at line 21. Specifically, we show that each operation returns the head element of the queue and correctly updates the *Head* pointer, which ensures that no other dequeue operation can return the same element.

Suppose an operation returns at line 21. By line 20, $done = true$, which implies that the `CAS` call in line 19 succeeded. Since $h = Head$ must have held at line 19, it must have held at all other points between the execution of lines 12 and 19. It follows from lines 16 and 18 that z and *out* store the address of the node after the dummy node and the element value stored in that node, respectively.[3] Recall that the element value stored in that node is the head element. It follows that the execution of line 19 advanced *Head* to point to the node after the (old) dummy node. In addition, the value returned at line 21 was the head element in the queue when line 19 was executed. The correctness of returns at line 21 follows.

[3]A node that is already in the queue could only be removed by advancing the *Head* pointer. Hence, the fact that $h = Head$ still holds at line 19 implies that no other dequeuer removed the node referenced by z between the execution of lines 16 and 19.

A.7 Liveness

In this section, we argue that liveness is guaranteed by the algorithm. To satisfy the requirements of lock-freedom, it must be shown that the group of all enqueuers and dequeuers must be capable of making progress at all times, though individual tasks may still be starved.

A.7.1 Enqueue Operations

Given the properties used to establish the correctness of enqueue operations, arguing liveness is straightforward. Specifically, we argue that at least one change[4] in the queue's state must occur during each enqueue loop iteration by some task. Since each pair of consecutive state changes represents the addition of a new element into the queue, it follows that some enqueue operations must be completing (and hence, making progress).

First, suppose a loop iteration begins when the queue is in the normal state. In this case, the queue must transition to an extended state at or before line 7 is executed, unless *Tail* is changed first. However, by (S2), no execution of line 9 can succeed while the queue is in the normal state. Since *Tail* is only updated at line 9, it follows that line 7 can only be avoided by an earlier transition to the extended state. In either case, the transition must occur before the end of the loop iteration.

Now, suppose a loop iteration begins when the queue is in the extended state. In this case, the queue must return to the normal state at or before line 9. This line can only be avoided by the condition at line 6. However, if *Tail* changes between lines 4 and 6, then some task must have successfully executed line 9, in which case the queue has already returned to the normal state. It follows that the queue must return to the normal state before the end of

[4]Specifically, a state change refers to the queue toggling between the normal and extended states.

the loop iteration.

A.7.2 Dequeue Operations

The liveness of dequeue operations is easily proved. Specifically, a dequeue attempt fails only if *Head* changes after line 12, resulting in a failed comparison at lines 14 or 19.[5] However, such a change implies that some dequeuer successfully claimed an element by executing line 19 (and hence, made progress).

A.8 Linearization

In [30], Herlihy and Wing proposed *linearizability* as a correctness condition for concurrent object implementations. For an object to be linearizable, it must be possible to define a *linearization point* for each operation. This point is required to correspond to the execution of a specific line that occurred during the operation.[6] In addition, the result of each operation must be identical to the result that would be achieved by applying each operation atomically in the order implied by the operations' linearization points. Hence, linearization points effectively define a total ordering of operations.

A.8.1 Enqueue Operations

Since the queue is considered empty whenever $Head.value = Tail.value$ holds, an enqueued value first becomes "visible" to other tasks when the queue transitions from the extended state

[5]When nodes are not recycled, the comparison at line 17 cannot fail. Specifically, the dummy node's pointer is *nil* only when the queue is empty. In that case, the branch consisting of lines 14–15 is taken and line 17 is never reached. When nodes are recycled, a failure at line 17 implies that the node has already been removed from the queue, and hence that another dequeue operation completed.

[6]The linearization point can be the execution of a line by a task other than the one performing the operation, provided that the line's execution occurred during the operation in question.

to the normal state, *i.e.*, when the *Tail* pointer is updated at line 9. Hence, the linearization point of an enqueue operation of task e is the execution of line 9 that updates *Tail* to point to $e.n$. This point may correspond to the execution of line 9 by an enqueuer other than e. Once e succeeds in appending $e.n$ onto the queue at line 7, it must attempt to update *Tail* at line 9 to ensure that the linearization point occurs during the operation. Specifically, e can only fail to update *Tail* at line 9 if some other enqueuer updates *Tail* first. Hence, the linearization point must occur while $e@\{8 \ldots 9\}$ holds.

A.8.2 Dequeue Operations

For a dequeue operation, two cases must be considered. The first case corresponds to operations that return at line 15. As explained above, an operation can return at this line only if $Head.value = Tail.value$ held when line 13 was executed. Hence, the queue was empty at this point and the execution of line 13 is the linearization point of these operations.

The second case corresponds to operations that return at line 21. Again, by the correctness arguments given earlier, a return at line 21 can only occur if the `CAS` call at line 19 succeeds. This call, in turn, succeeds only if the value of the dequeuer's local variables reflect the state of the queue when line 19 is executed. Hence, the execution of line 19 serves as the linearization point when operations return at line 21.

APPENDIX B

Pairwise Comparison

This appendix provides additional information obtained from the synchronization studies discussed earlier in Chapters 5 and 6. In addition to the previously presented results, three of these studies included a pairwise comparison of approaches. Specifically, for each pair of approaches A and B, the fraction of the samples for which A outperforms B, B outperforms A, and both A and B produce the same performance were calculated. (In this context, A outperforms B if it produces less inflation.) We now present the results of these pairwise comparisons.

B.1 Lock-free Studies

The first study that included a pairwise comparison was the lock-free study presented earlier in Section 5.5.1. (See Section 5.5.1 for a detailed description of the study.) Figures B.1, B.2, B.3, and B.4 summarize the pairwise comparison results for the $M = 2$, 4, 8, and 16 cases.

The proper interpretation of table data will be explained using Figure B.1 as an example. Each table shown in Figure B.1 gives a 5-by-5 matrix of percentages. Notice that row and column indexes (from 1 to 5) are given to the left and above these percentages. To simplify this discussion, let I and E denote the upper (inset (a)) and lower (inset (b)) matrices of percentages, respectively. In addition, let $I_{r,c}$ (respectively, $E_{r,c}$) denote the value in row r and column c of matrix I (respectively, matrix E). For instance, $I_{3,2}$ = 20.948% while

$M = 2$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	35.258%	31.135%	3.140%	2.949%
QB-EPDF (FF)	**2**	64.742%	*N/A*	2.877%	0.000%	2.266%
QB-EPDF (NF)	**3**	68.865%	20.948%	*N/A*	0.708%	0.000%
QB-EDF (FF)	**4**	96.858%	48.395%	46.443%	*N/A*	7.549%
QB-EDF (NF)	**5**	97.049%	47.797%	43.674%	4.092%	*N/A*

(a) Inequality ($<$)

$M = 2$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	0.000%	0.000%	0.002%	0.002%
QB-EPDF (FF)	**2**	0.000%	*N/A*	76.175%	51.605%	49.937%
QB-EPDF (NF)	**3**	0.000%	76.175%	*N/A*	52.849%	56.326%
QB-EDF (FF)	**4**	0.002%	51.605%	52.849%	*N/A*	88.358%
QB-EDF (NF)	**5**	0.002%	49.937%	56.326%	88.358%	*N/A*

(b) Equality ($=$)

Figure B.1: Pairwise comparison of the number of tested samples favoring each approach in the $M = 2$ case of the first lock-free synchronization study. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 4$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	35.258%	31.135%	3.140%	2.949%
QB-EPDF (FF)	**2**	64.742%	*N/A*	2.877%	0.000%	2.266%
QB-EPDF (NF)	**3**	68.865%	20.948%	*N/A*	0.708%	0.000%
QB-EDF (FF)	**4**	96.858%	48.395%	46.443%	*N/A*	7.549%
QB-EDF (NF)	**5**	97.049%	47.797%	43.674%	4.092%	*N/A*

(a) Inequality ($<$)

$M = 4$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%
QB-EPDF (FF)	**2**	0.000%	*N/A*	37.783%	46.145%	26.800%
QB-EPDF (NF)	**3**	0.000%	37.783%	*N/A*	28.912%	50.892%
QB-EDF (FF)	**4**	0.000%	46.145%	28.912%	*N/A*	47.286%
QB-EDF (NF)	**5**	0.000%	26.800%	50.892%	47.286%	*N/A*

(b) Equality ($=$)

Figure B.2: Pairwise comparison of the number of tested samples favoring each approach in the $M = 4$ case of the first lock-free synchronization study. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 8$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	17.102%	17.008%	0.022%	0.045%
QB-EPDF (FF)	**2**	82.898%	*N/A*	52.983%	0.000%	37.489%
QB-EPDF (NF)	**3**	82.992%	37.415%	*N/A*	6.082%	0.000%
QB-EDF (FF)	**4**	99.978%	63.394%	85.569%	*N/A*	72.600%
QB-EDF (NF)	**5**	99.955%	54.862%	59.155%	15.114%	*N/A*

(a) Inequality (<)

$M = 8$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%
QB-EPDF (FF)	**2**	0.000%	*N/A*	9.602%	36.606%	7.649%
QB-EPDF (NF)	**3**	0.000%	9.602%	*N/A*	8.349%	40.845%
QB-EDF (FF)	**4**	0.000%	36.606%	8.349%	*N/A*	12.286%
QB-EDF (NF)	**5**	0.000%	7.649%	40.845%	12.286%	*N/A*

(b) Equality (=)

Figure B.3: Pairwise comparison of the number of tested samples favoring each approach in the $M = 8$ case of the first lock-free synchronization study. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 16$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	22.040%	24.942%	0.248%	0.557%
QB-EPDF (FF)	**2**	77.960%	*N/A*	73.742%	0.000%	37.482%
QB-EPDF (NF)	**3**	75.058%	25.394%	*N/A*	3.725%	0.000%
QB-EDF (FF)	**4**	99.752%	81.852%	95.525%	*N/A*	85.934%
QB-EDF (NF)	**5**	99.443%	61.892%	80.283%	12.943%	*N/A*

(a) Inequality (<)

$M = 16$	#	**1**	**2**	**3**	**4**	**5**
No Supertasks	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%
QB-EPDF (FF)	**2**	0.000%	*N/A*	0.865%	18.148%	0.626%
QB-EPDF (NF)	**3**	0.000%	0.865%	*N/A*	0.751%	19.717%
QB-EDF (FF)	**4**	0.000%	18.148%	0.751%	*N/A*	1.123%
QB-EDF (NF)	**5**	0.000%	0.626%	19.717%	1.123%	*N/A*

(b) Equality (=)

Figure B.4: Pairwise comparison of the number of tested samples favoring each approach in the $M = 16$ case of the first lock-free synchronization study. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$E_{3,2} = 76.175\%$. Finally, the labels shown in the first column of each table (to the left of the row indexes) identify which approach is associated with each index. For instance, the "No Supertasks" approach is assigned index 1 in Figure B.1.

The I matrix is based on a $<$ comparison. Specifically, the percentage $I_{r,c}$ reflects the fraction of the tested samples for which the approach with index r produced strictly less inflation than the approach with index c. For instance, consider comparing the "No Supertasks" and "QB-EPDF (NF)" approaches, which are associated with 1 and 3, respectively. By $I_{1,3}$, the "No Supertasks" approach produced less inflation than the "QB-EPDF (NF)" approach in 31.135% of the samples. Similarly, by $I_{3,1}$, the "QB-EPDF (NF)" approach produced less inflation than the "No Supertasks" approach in 68.865% of the samples.

One possibility is not directly presented in the I matrix: both approaches could have produced the same inflation. Actually, the fraction of the samples associated with this third possibility is easily calculated using the formula $100\% - I_{r,c} - I_{c,r}$. For convenience, we have provided these values in the E matrix. Specifically, the percentage $E_{r,c}$ reflects the fraction of the tested samples for which the approach with index r produced the same inflation as the approach with index c. Since this matrix is based on equivalence, it is symmetric about its diagonal, *i.e.*, $E_{r,c} = E_{c,r}$.

Since comparing an approach to itself is uninteresting, the diagonal in each matrix was simply filled by *N/A*. (By the definitions given above, $I_{r,r} = 0\%$ and $E_{r,r} = 100\%$ for all r.)

B.2 Locking Studies

The first two locking studies, discussed earlier in Sections 6.6.1 and 6.6.2, also included pairwise comparisons. Figures B.5, B.6, B.7, and B.8 summarize the comparison results for the $M = 2$, 4, 8, and 16 cases of the first study. This study focuses on the performance of the locking approaches when locks satisfy the (R3) property. The second study then considered the case in which (R3) does not hold. Figures B.9, B.10, B.11, and B.12 summarize the comparison results for the $M = 2$, 4, 8, and 16 cases of the second study.

$M = 2$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.062%	0.060%	0.000%	0.000%
SP	**2**	100.000%	*N/A*	39.140%	36.045%	11.071%	12.292%
SP/QB-EPDF (FF)	**3**	99.938%	60.860%	*N/A*	3.528%	0.000%	2.618%
SP/QB-EPDF (NF)	**4**	99.940%	63.955%	20.000%	*N/A*	0.248%	0.000%
SP/QB-EDF (FF)	**5**	100.000%	88.929%	48.160%	46.282%	*N/A*	8.778%
SP/QB-EDF (NF)	**6**	100.000%	87.708%	46.897%	43.508%	3.145%	*N/A*
RP	**7**	100.000%	0.000%	38.351%	35.086%	9.371%	10.323%
RP/QB-EPDF (FF)	**8**	99.938%	58.845%	0.694%	1.483%	0.012%	0.666%
RP/QB-EPDF (NF)	**9**	99.940%	61.682%	18.543%	0.280%	0.060%	0.000%
RP/QB-EDF (FF)	**10**	100.000%	84.122%	45.143%	42.177%	0.057%	2.722%
RP/QB-EDF (NF)	**11**	100.000%	83.051%	44.574%	40.771%	2.105%	0.006%

$M = 2$	#	**7**	**8**	**9**	**10**	**11**	
SWSP	**1**	0.000%	0.062%	0.060%	0.000%	0.000%	
SP	**2**	14.572%	41.155%	38.318%	15.878%	16.949%	
SP/QB-EPDF (FF)	**3**	61.649%	19.006%	14.074%	5.895%	6.882%	
SP/QB-EPDF (NF)	**4**	64.914%	30.217%	16.788%	5.472%	6.866%	
SP/QB-EDF (FF)	**5**	90.629%	51.028%	49.386%	21.234%	23.878%	
SP/QB-EDF (NF)	**6**	89.677%	49.962%	47.637%	18.614%	18.912%	
RP	**7**	*N/A*	40.229%	37.138%	13.638%	14.471%	
RP/QB-EPDF (FF)	**8**	59.771%	*N/A*	2.426%	0.000%	1.611%	
RP/QB-EPDF (NF)	**9**	62.862%	20.589%	*N/A*	0.231%	0.000%	
RP/QB-EDF (FF)	**10**	86.362%	48.043%	45.205%	*N/A*	5.795%	
RP/QB-EDF (NF)	**11**	85.529%	47.308%	43.454%	3.537%	*N/A*	

(a) Inequality (<)

$M = 2$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	76.472%	51.840%	50.485%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	76.472%	*N/A*	53.471%	56.492%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	51.840%	53.471%	*N/A*	88.077%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	50.485%	56.492%	88.077%	*N/A*
RP	**7**	0.000%	85.428%	0.000%	0.000%	0.000%	0.000%
RP/QB-EPDF (FF)	**8**	0.000%	0.000%	80.300%	68.300%	48.960%	49.372%
RP/QB-EPDF (NF)	**9**	0.000%	0.000%	67.383%	82.932%	50.554%	52.363%
RP/QB-EDF (FF)	**10**	0.000%	0.000%	48.962%	52.351%	78.709%	78.665%
RP/QB-EDF (NF)	**11**	0.000%	0.000%	48.545%	52.363%	74.017%	81.082%

$M = 2$	#	**7**	**8**	**9**	**10**	**11**	
SWSP	**1**	0.000%	0.000%	0.000%	0.000%	0.000%	
SP	**2**	85.428%	0.000%	0.000%	0.000%	0.000%	
SP/QB-EPDF (FF)	**3**	0.000%	80.300%	67.383%	48.962%	48.545%	
SP/QB-EPDF (NF)	**4**	0.000%	68.300%	82.932%	52.351%	52.363%	
SP/QB-EDF (FF)	**5**	0.000%	48.960%	50.554%	78.709%	74.017%	
SP/QB-EDF (NF)	**6**	0.000%	49.372%	52.363%	78.665%	81.082%	
RP	**7**	*N/A*	0.000%	0.000%	0.000%	0.000%	
RP/QB-EPDF (FF)	**8**	0.000%	*N/A*	76.985%	51.957%	51.082%	
RP/QB-EPDF (NF)	**9**	0.000%	76.985%	*N/A*	54.565%	56.546%	
RP/QB-EDF (FF)	**10**	0.000%	51.957%	54.565%	*N/A*	90.668%	
RP/QB-EDF (NF)	**11**	0.000%	51.082%	56.546%	90.668%	*N/A*	

(b) Equality (=)

Figure B.5: Pairwise comparison of the number of tested samples favoring each approach in the $M = 2$ case and (R3) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 4$	#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
SWSP	**1**	*N/A*	0.000%	0.008%	0.005%	0.000%	0.000%	0.000%	0.008%	0.005%	0.000%	0.000%
SP	**2**	100.000%	*N/A*	28.340%	25.192%	1.308%	1.394%	85.688%	29.454%	26.389%	3.031%	3.668%
SP/QB-EPDF (FF)	**3**	99.992%	71.660%	*N/A*	23.275%	0.000%	17.663%	73.497%	57.237%	47.217%	31.335%	36.145%
SP/QB-EPDF (NF)	**4**	99.995%	74.808%	33.675%	*N/A*	1.863%	0.000%	76.714%	60.269%	56.331%	27.108%	33.943%
SP/QB-EDF (FF)	**5**	100.000%	98.692%	54.023%	63.943%	*N/A*	35.634%	99.831%	77.209%	79.095%	61.569%	68.291%
SP/QB-EDF (NF)	**6**	100.000%	98.606%	50.854%	49.477%	8.038%	*N/A*	99.820%	71.309%	75.095%	50.329%	60.718%
RP	**7**	100.000%	0.000%	26.503%	23.286%	0.169%	0.180%	*N/A*	27.388%	24.258%	0.440%	0.517%
RP/QB-EPDF (FF)	**8**	99.992%	70.546%	1.848%	10.198%	0.089%	5.022%	72.612%	*N/A*	20.186%	0.000%	14.800%
RP/QB-EPDF (NF)	**9**	99.995%	73.611%	27.365%	0.625%	0.157%	0.000%	75.742%	34.152%	*N/A*	1.911%	0.000%
RP/QB-EDF (FF)	**10**	100.000%	96.969%	45.957%	46.745%	0.205%	10.406%	99.560%	53.906%	60.771%	*N/A*	31.755%
RP/QB-EDF (NF)	**11**	100.000%	96.332%	44.125%	41.157%	2.023%	0.012%	99.483%	51.149%	49.309%	7.755%	*N/A*

(a) Inequality (<)

$M = 4$	#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%	14.312%	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	43.049%	45.977%	31.483%	0.000%	40.915%	25.418%	22.708%	19.731%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	43.049%	*N/A*	34.194%	50.523%	0.000%	29.532%	43.045%	26.148%	24.900%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	45.977%	34.194%	*N/A*	56.328%	0.000%	22.702%	20.748%	38.226%	29.686%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	31.483%	50.523%	56.328%	*N/A*	0.000%	23.669%	24.905%	39.265%	39.269%
RP	**7**	0.000%	14.312%	0.000%	0.000%	0.000%	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
RP/QB-EPDF (FF)	**8**	0.000%	0.000%	40.915%	29.532%	22.702%	23.669%	0.000%	*N/A*	45.662%	46.094%	34.051%
RP/QB-EPDF (NF)	**9**	0.000%	0.000%	25.418%	43.045%	20.748%	24.905%	0.000%	45.662%	*N/A*	37.318%	50.691%
RP/QB-EDF (FF)	**10**	0.000%	0.000%	22.708%	26.148%	38.226%	39.265%	0.000%	46.094%	37.318%	*N/A*	60.489%
RP/QB-EDF (NF)	**11**	0.000%	0.000%	19.731%	24.900%	29.686%	39.269%	0.000%	34.051%	50.691%	60.489%	*N/A*

(b) Equality (=)

Figure B.6: Pairwise comparison of the number of tested samples favoring each approach in the $M = 4$ case and (R3) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 8$	#	1	2	3	4	5	6
SWSP	**1**	*N/A*	0.000%	0.002%	0.011%	0.000%	0.000%
SP	**2**	100.000%	*N/A*	27.600%	26.469%	1.962%	2.608%
SP/QB-EPDF (FF)	**3**	99.998%	72.400%	*N/A*	46.537%	0.000%	31.860%
SP/QB-EPDF (NF)	**4**	99.989%	73.531%	38.849%	*N/A*	3.600%	0.000%
SP/QB-EDF (FF)	**5**	100.000%	98.038%	63.475%	82.634%	*N/A*	65.686%
SP/QB-EDF (NF)	**6**	100.000%	97.392%	55.888%	58.862%	12.232%	*N/A*
RP	**7**	100.000%	0.000%	22.128%	21.397%	0.089%	0.163%
RP/QB-EPDF (FF)	**8**	99.998%	69.974%	3.802%	22.938%	0.083%	4.971%
RP/QB-EPDF (NF)	**9**	99.989%	70.760%	27.495%	1.374%	0.058%	0.002%
RP/QB-EDF (FF)	**10**	100.000%	91.548%	46.371%	48.877%	0.142%	10.632%
RP/QB-EDF (NF)	**11**	100.000%	89.694%	43.538%	42.323%	0.731%	0.020%
$M = 8$	#	**7**	**8**	**9**	**10**	**11**	
SWSP	**1**	0.000%	0.002%	0.011%	0.000%	0.000%	
SP	**2**	99.886%	30.026%	29.240%	8.452%	10.305%	
SP/QB-EPDF (FF)	**3**	77.872%	83.235%	67.631%	48.142%	52.329%	
SP/QB-EPDF (NF)	**4**	78.603%	69.926%	84.028%	43.734%	51.557%	
SP/QB-EDF (FF)	**5**	99.911%	94.443%	95.586%	91.246%	93.745%	
SP/QB-EDF (NF)	**6**	99.837%	88.685%	93.883%	78.966%	91.048%	
RP	**7**	*N/A*	23.523%	22.883%	0.378%	0.671%	
RP/QB-EPDF (FF)	**8**	76.477%	*N/A*	45.055%	0.000%	30.618%	
RP/QB-EPDF (NF)	**9**	77.117%	38.583%	*N/A*	3.478%	0.000%	
RP/QB-EDF (FF)	**10**	99.622%	63.357%	80.768%	*N/A*	65.246%	
RP/QB-EDF (NF)	**11**	99.329%	55.382%	58.960%	11.308%	*N/A*	

(a) Inequality (<)

$M = 8$	#	1	2	3	4	5	6
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	14.614%	36.525%	12.252%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	14.614%	*N/A*	13.766%	41.138%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	36.525%	13.766%	*N/A*	22.082%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	12.252%	41.138%	22.082%	*N/A*
RP	**7**	0.000%	0.114%	0.000%	0.000%	0.000%	0.000%
RP/QB-EPDF (FF)	**8**	0.000%	0.000%	12.963%	7.135%	5.474%	6.345%
RP/QB-EPDF (NF)	**9**	0.000%	0.000%	4.874%	14.598%	4.355%	6.115%
RP/QB-EDF (FF)	**10**	0.000%	0.000%	5.488%	7.389%	8.612%	10.402%
RP/QB-EDF (NF)	**11**	0.000%	0.002%	4.132%	6.120%	5.525%	8.932%
$M = 8$	#	**7**	**8**	**9**	**10**	**11**	
SWSP	**1**	0.000%	0.000%	0.000%	0.000%	0.000%	
SP	**2**	0.114%	0.000%	0.000%	0.000%	0.002%	
SP/QB-EPDF (FF)	**3**	0.000%	12.963%	4.874%	5.488%	4.132%	
SP/QB-EPDF (NF)	**4**	0.000%	7.135%	14.598%	7.389%	6.120%	
SP/QB-EDF (FF)	**5**	0.000%	5.474%	4.355%	8.612%	5.525%	
SP/QB-EDF (NF)	**6**	0.000%	6.345%	6.115%	10.402%	8.932%	
RP	**7**	*N/A*	0.000%	0.000%	0.000%	0.000%	
RP/QB-EPDF (FF)	**8**	0.000%	*N/A*	16.362%	36.643%	14.000%	
RP/QB-EPDF (NF)	**9**	0.000%	16.362%	*N/A*	15.754%	41.040%	
RP/QB-EDF (FF)	**10**	0.000%	36.643%	15.754%	*N/A*	23.446%	
RP/QB-EDF (NF)	**11**	0.000%	14.000%	41.040%	23.446%	*N/A*	

(b) Equality (=)

Figure B.7: Pairwise comparison of the number of tested samples favoring each approach in the $M = 8$ case and (R3) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 16$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.032%	0.162%	0.000%	0.000%
SP	**2**	100.000%	*N/A*	40.048%	40.332%	6.035%	7.962%
SP/QB-EPDF (FF)	**3**	99.968%	59.952%	*N/A*	64.732%	0.000%	27.785%
SP/QB-EPDF (NF)	**4**	99.838%	59.668%	33.155%	*N/A*	3.702%	0.000%
SP/QB-EDF (FF)	**5**	100.000%	93.963%	80.945%	94.052%	*N/A*	79.723%
SP/QB-EDF (NF)	**6**	100.000%	92.038%	70.402%	77.908%	16.054%	*N/A*
RP	**7**	100.000%	0.000%	28.866%	30.394%	0.569%	1.115%
RP/QB-EPDF (FF)	**8**	99.966%	52.180%	6.686%	38.134%	0.031%	2.534%
RP/QB-EPDF (NF)	**9**	99.826%	51.205%	17.954%	2.214%	0.008%	0.008%
RP/QB-EDF (FF)	**10**	100.000%	71.083%	54.100%	56.451%	0.042%	7.231%
RP/QB-EDF (NF)	**11**	100.000%	66.969%	50.449%	50.478%	0.095%	0.011%
$M = 16$	#	**7**	**8**	**9**	**10**	**11**	
SWSP	**1**	0.000%	0.034%	0.174%	0.000%	0.000%	
SP	**2**	99.998%	47.820%	48.795%	28.917%	33.031%	
SP/QB-EPDF (FF)	**3**	71.134%	86.080%	81.815%	45.632%	49.434%	
SP/QB-EPDF (NF)	**4**	69.606%	61.208%	89.092%	42.818%	49.185%	
SP/QB-EDF (FF)	**5**	99.431%	99.703%	99.869%	99.438%	99.722%	
SP/QB-EDF (NF)	**6**	98.885%	96.937%	99.655%	91.697%	99.468%	
RP	**7**	*N/A*	31.126%	32.702%	2.312%	3.549%	
RP/QB-EPDF (FF)	**8**	68.874%	*N/A*	66.111%	0.000%	30.037%	
RP/QB-EPDF (NF)	**9**	67.298%	31.551%	*N/A*	3.232%	0.000%	
RP/QB-EDF (FF)	**10**	97.688%	80.832%	94.409%	*N/A*	84.063%	
RP/QB-EDF (NF)	**11**	96.451%	67.932%	78.046%	12.018%	*N/A*	

(a) Inequality ($<$)

$M = 16$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.002%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	2.112%	19.055%	1.814%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	2.112%	*N/A*	2.246%	22.092%
SP/QB-EDF (FF)	**5**	0.000%	0.002%	19.055%	2.246%	*N/A*	4.223%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	1.814%	22.092%	4.223%	*N/A*
RP	**7**	0.000%	0.002%	0.000%	0.000%	0.000%	0.000%
RP/QB-EPDF (FF)	**8**	0.000%	0.000%	7.234%	0.658%	0.266%	0.529%
RP/QB-EPDF (NF)	**9**	0.000%	0.000%	0.231%	8.694%	0.123%	0.337%
RP/QB-EDF (FF)	**10**	0.000%	0.000%	0.268%	0.731%	0.520%	1.072%
RP/QB-EDF (NF)	**11**	0.000%	0.000%	0.117%	0.337%	0.183%	0.522%
$M = 16$	#	**7**	**8**	**9**	**10**	**11**	
SWSP	**1**	0.000%	0.000%	0.000%	0.000%	0.000%	
SP	**2**	0.002%	0.000%	0.000%	0.000%	0.000%	
SP/QB-EPDF (FF)	**3**	0.000%	7.234%	0.231%	0.268%	0.117%	
SP/QB-EPDF (NF)	**4**	0.000%	0.658%	8.694%	0.731%	0.337%	
SP/QB-EDF (FF)	**5**	0.000%	0.266%	0.123%	0.520%	0.183%	
SP/QB-EDF (NF)	**6**	0.000%	0.529%	0.337%	1.072%	0.522%	
RP	**7**	*N/A*	0.000%	0.000%	0.000%	0.000%	
RP/QB-EPDF (FF)	**8**	0.000%	*N/A*	2.338%	19.168%	2.031%	
RP/QB-EPDF (NF)	**9**	0.000%	2.338%	*N/A*	2.358%	21.954%	
RP/QB-EDF (FF)	**10**	0.000%	19.168%	2.358%	*N/A*	3.918%	
RP/QB-EDF (NF)	**11**	0.000%	2.031%	21.954%	3.918%	*N/A*	

(b) Equality ($=$)

Figure B.8: Pairwise comparison of the number of tested samples favoring each approach in the $M = 16$ case and (R3) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 2$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	1.904%	0.022%	0.015%	0.000%	0.000%
SP	**2**	98.096%	*N/A*	10.254%	9.555%	0.015%	0.011%
SP/QB-EPDF (FF)	**3**	99.978%	89.746%	*N/A*	3.518%	0.000%	2.290%
SP/QB-EPDF (NF)	**4**	99.985%	90.445%	7.787%	*N/A*	0.074%	0.000%
SP/QB-EDF (FF)	**5**	100.000%	99.985%	40.022%	40.033%	*N/A*	10.636%
SP/QB-EDF (NF)	**6**	100.000%	99.989%	37.816%	39.566%	0.849%	*N/A*

(a) Inequality (<)

$M = 2$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	88.695%	59.978%	59.893%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	88.695%	*N/A*	59.893%	60.434%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	59.978%	59.893%	*N/A*	88.515%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	59.893%	60.434%	88.515%	*N/A*

(b) Equality (=)

Figure B.9: Pairwise comparison of the number of tested samples favoring each approach in the $M = 2$ case and (R1) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 4$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	2.871%	0.467%	0.548%	0.423%	0.515%
SP	**2**	97.129%	*N/A*	8.809%	8.140%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	99.533%	91.191%	*N/A*	49.673%	0.000%	41.426%
SP/QB-EPDF (NF)	**4**	99.452%	91.860%	24.062%	*N/A*	4.074%	0.000%
SP/QB-EDF (FF)	**5**	99.577%	100.000%	63.415%	81.746%	*N/A*	65.548%
SP/QB-EDF (NF)	**6**	99.485%	100.000%	44.423%	58.930%	7.875%	*N/A*

(a) Inequality (<)

$M = 4$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	26.265%	36.585%	14.151%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	26.265%	*N/A*	14.180%	41.070%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	36.585%	14.180%	*N/A*	26.577%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	14.151%	41.070%	26.577%	*N/A*

(b) Equality (=)

Figure B.10: Pairwise comparison of the number of tested samples favoring each approach in the $M = 4$ case and (R1) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 8$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	3.544%	2.456%	2.706%	2.283%	2.430%
SP	**2**	96.456%	*N/A*	16.125%	17.540%	0.000%	0.015%
SP/QB-EPDF (FF)	**3**	97.544%	83.875%	*N/A*	63.945%	0.000%	42.518%
SP/QB-EPDF (NF)	**4**	97.294%	82.460%	33.757%	*N/A*	5.401%	0.000%
SP/QB-EDF (FF)	**5**	97.717%	100.000%	81.015%	93.824%	*N/A*	84.092%
SP/QB-EDF (NF)	**6**	97.570%	99.985%	56.724%	80.805%	13.504%	*N/A*

(a) Inequality (<)

$M = 8$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	2.298%	18.985%	0.757%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	2.298%	*N/A*	0.776%	19.195%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	18.985%	0.776%	*N/A*	2.404%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	0.757%	19.195%	2.404%	*N/A*

(b) Equality (=)

Figure B.11: Pairwise comparison of the number of tested samples favoring each approach in the $M = 8$ case and (R1) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

$M = 16$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	3.923%	3.739%	4.415%	3.426%	3.478%
SP	**2**	96.077%	*N/A*	37.062%	47.460%	0.287%	1.213%
SP/QB-EPDF (FF)	**3**	96.261%	62.938%	*N/A*	74.974%	0.000%	19.743%
SP/QB-EPDF (NF)	**4**	95.585%	52.540%	24.857%	*N/A*	3.301%	0.000%
SP/QB-EDF (FF)	**5**	96.574%	99.713%	94.739%	96.688%	*N/A*	85.099%
SP/QB-EDF (NF)	**6**	96.522%	98.787%	80.246%	95.312%	14.761%	*N/A*

(a) Inequality (<)

$M = 16$	#	**1**	**2**	**3**	**4**	**5**	**6**
SWSP	**1**	*N/A*	0.000%	0.000%	0.000%	0.000%	0.000%
SP	**2**	0.000%	*N/A*	0.000%	0.000%	0.000%	0.000%
SP/QB-EPDF (FF)	**3**	0.000%	0.000%	*N/A*	0.169%	5.261%	0.011%
SP/QB-EPDF (NF)	**4**	0.000%	0.000%	0.169%	*N/A*	0.011%	4.688%
SP/QB-EDF (FF)	**5**	0.000%	0.000%	5.261%	0.011%	*N/A*	0.140%
SP/QB-EDF (NF)	**6**	0.000%	0.000%	0.011%	4.688%	0.140%	*N/A*

(b) Equality (=)

Figure B.12: Pairwise comparison of the number of tested samples favoring each approach in the $M = 16$ case and (R1) holds. Tables summarize the **(a)** inequality and **(b)** equality relationships.

APPENDIX C

Reweighting under the Staggered Model

To evaluate the impact of using staggered slots (discussed in Chapter 7) on the amount of reweighting overhead experienced under the FP-EDF supertasking scenario (see Chapter 4), we repeated the first study from Chapter 4 using only the FP-EDF scenario.[1] Inflation was measured under each of the aligned and staggered scheduling models. All aspects of this study are identical to that of the corresponding study in Chapter 4, except that the number of samples collected for each combination of parameters was increased from four to ten.

C.1 Measurement under the Staggered Model

As explained in Chapter 7, the reweighting framework presented in Chapter 4 can be (pessimistically) adapted for use under the staggered model by simply reducing the supply estimate by $\Delta \overset{\text{def}}{=} \frac{M-1}{M}$ (see Section 7.2). For this study, the supertask supply was actually reduced by one rather than Δ so that the reweighting process would not depend on M. In addition, this enables the use of the selected weight in systems consisting of any number of processors. (This property follows from the fact that $\Delta < 1$ holds for all M.)

The following definition of $\Delta(\mathcal{S}, L)$ for the staggered model follows easily from the deriva-

[1]We focus on FP-EDF because the QB-EPDF and QB-EDF scenarios include other forms of overhead, while inflation under the FP-EDF is due solely to reweighting overhead.

tions presented earlier in Section 4.3.1.

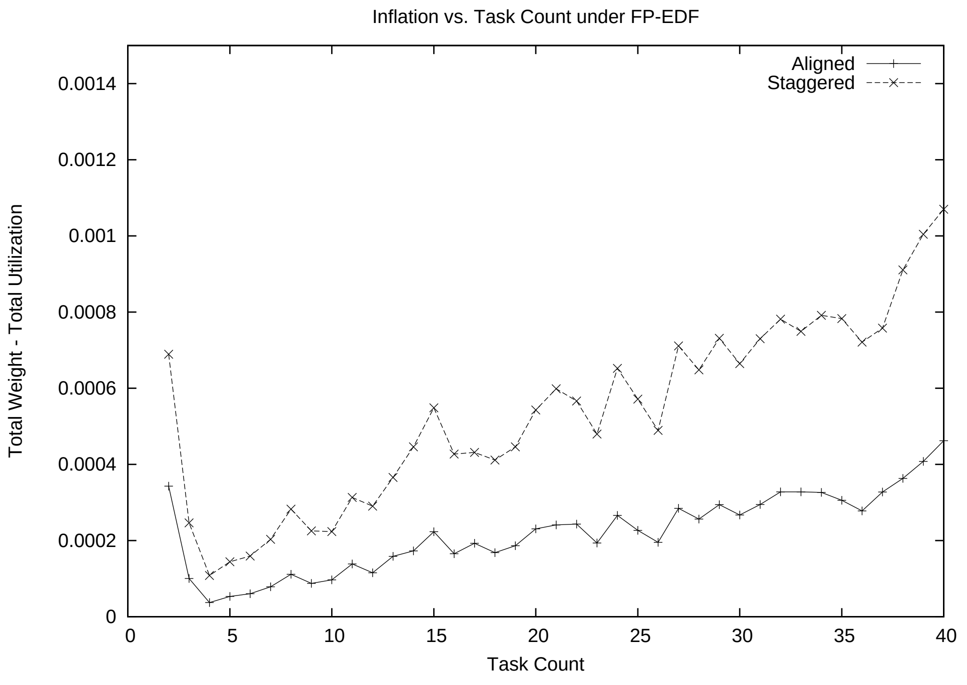
$$\Delta(\mathcal{S}, L) \overset{\text{def}}{=} \frac{\left\lceil \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L - T.c}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) \right\rceil + \beta}{\lfloor L \rfloor - 1 - \epsilon}$$

C.2 Results

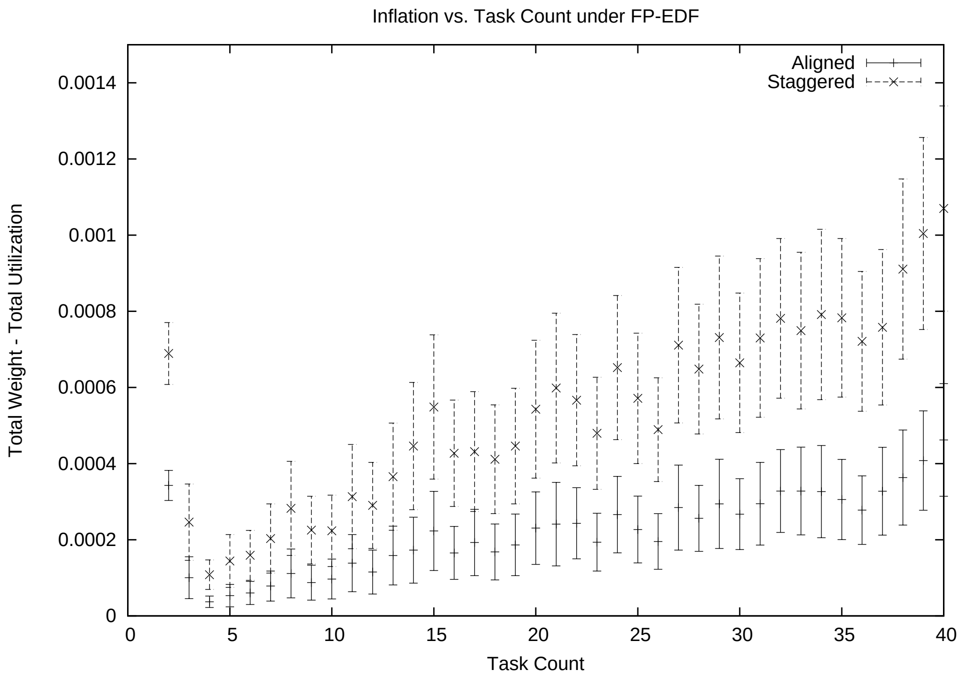
Figures C.1, C.2, and C.3 show how inflation varies with the task count, total utilization, and minimum task period, respectively. Sampling validity was approximately unity for all sample means. Because of this, the graphs of sampling validity are omitted.

In all graphs, the staggered model appears to increase the reweighting overhead by a factor of approximately 2.5. Indeed, the mean inflation based on all samples was 0.000220783 under the aligned model and 0.000536318 under the staggered model, which implies a factor of 2.43 increase. However, notice that the mean inflation is still consistently low (*i.e.*, less than 1.5% of a single processor's capacity) throughout. Because of this, staggering is not expected to increase reweighting overhead substantially under the QB-EDF and FP-EDF approaches, under which reweighting overhead is typically very low.[2] However, since reweighting overhead is significantly higher under QB-EPDF, staggering may produce prohibitively high inflation.

[2]Recall that inflation under QB-EDF primarily consists of mapping overhead.

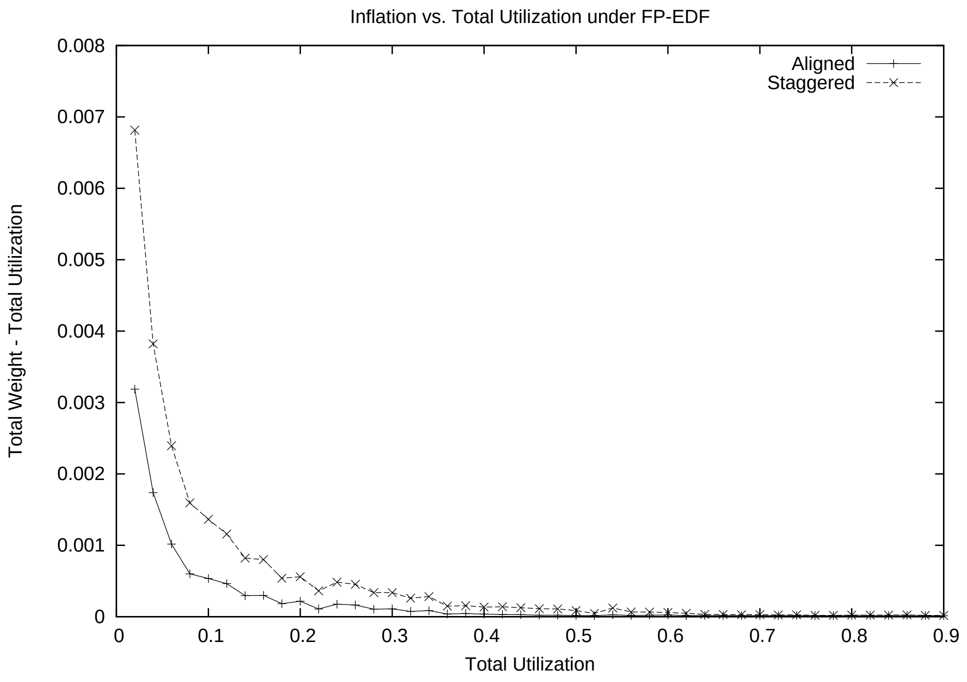


(a) Mean Inflation

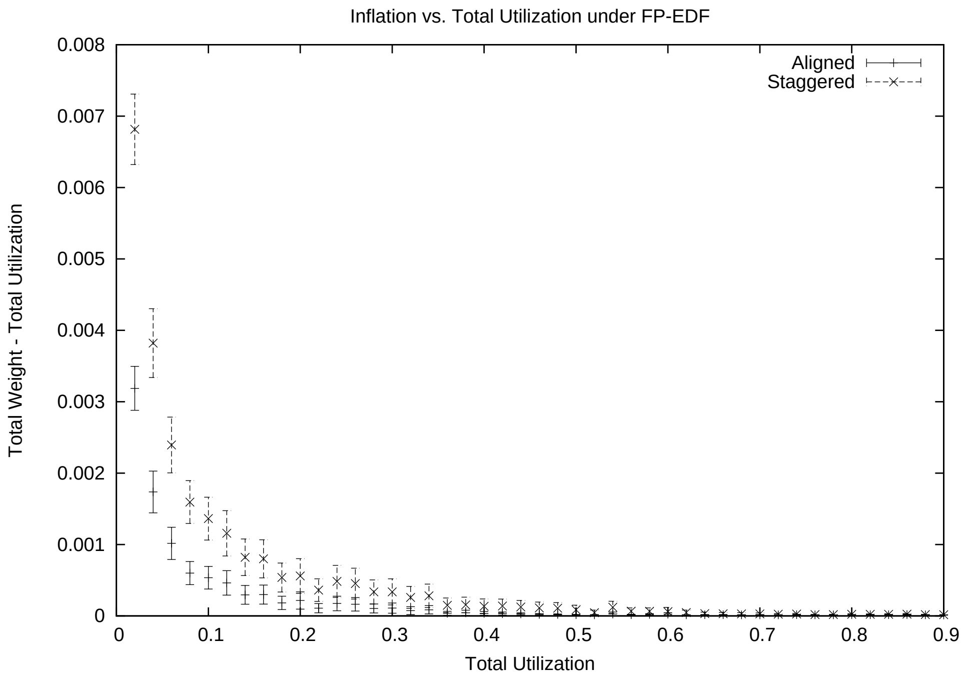


(b) 99% Confidence Interval

Figure C.1: Plots show how inflation varies with the task count under the FP-EDF scenario when using each of the aligned and staggered models. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each sample mean.

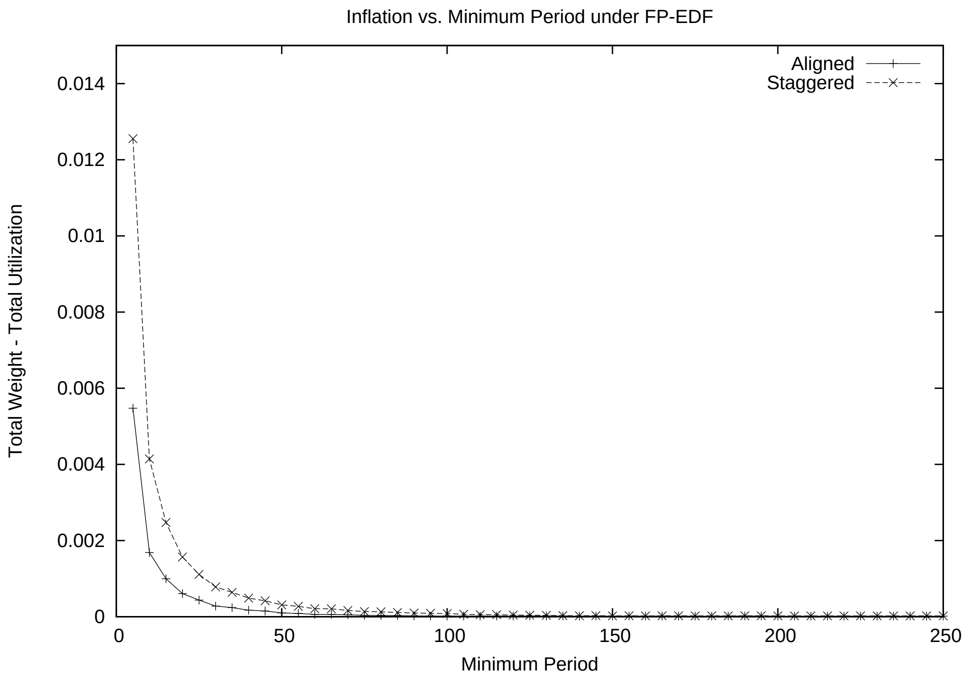


(a) Mean Inflation

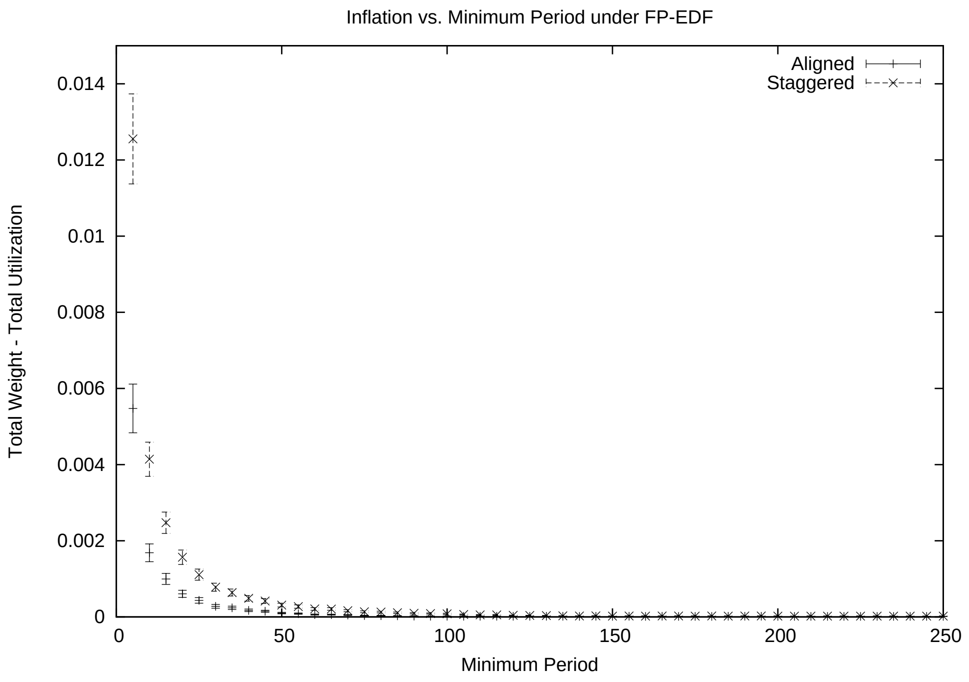


(b) 99% Confidence Interval

Figure C.2: Plots show how inflation varies with the utilization under the FP-EDF scenario when using each of the aligned and staggered models. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each sample mean.



(a) Mean Inflation



(b) 99% Confidence Interval

Figure C.3: Plots show how inflation varies with the minimum task period under the FP-EDF scenario when using each of the aligned and staggered models. The figure shows **(a)** the sample means and **(b)** the 99% confidence interval for each sample mean.

BIBLIOGRAPHY

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, December 1998.

[2] J. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In *Proceedings of the 24th IEEE Real-time Systems Symposium*, pages 130–141. December 2003.

[3] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-time Systems Symposium*, pages 92–105. December 1996.

[4] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. *ACM Transactions on Computer Systems*, 15(6):388–395, May 1997.

[5] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 35–43, June 2000.

[6] J. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *Proceedings of the Seventh International Conference on Real-time Computing Systems and Applications*, pages 297–306, December 2000.

[7] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-time Systems*, pages 76–85, June 2001.

[8] B. Andersson and J. Jonsson. The utilization bounds of partitioned and Pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 33–40, May 2003.

[9] T. Baker. Stack-based scheduling of real-time processes. *Real-time Systems*, 3(1):67–99, March 1991.

[10] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the 16th IEEE Real-time Systems Symposium*, pages 200–209. December 1995.

[11] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[12] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[13] A. Block and J. Anderson. Fast Task Reweighting in Fair-scheduled Multiprocessor Systems. In submission. April 2004.

[14] G. Bollella *et al.*. *The Real-time Specification for Java*, Addison Wesley, 2000.

[15] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 161–170. December 2001.

[16] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. To appear in the *Handbook on Scheduling Algorithms, Methods, and Models*, Joseph Y. Leung (ed.), Chapman Hall/CRC.

[17] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: a proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, October 2000.

[18] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In *Proceedings of the 7th IEEE Real-time Technology and Applications Symposium*, May 2001.

[19] S. Davari and S. Dhall. An on-line algorithm for real-time tasks allocation. In *Proceedings of the 7th IEEE Real-time Systems Symposium*, pages 194–200. December 1986.

[20] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 171–180. December 2001.

[21] U. Devi. Personal communication. 2004.

[22] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. Presented at the *12th International Workshop on Parallel and Distributed Real-time Systems* (on CD-ROM), April 2004.

[23] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[24] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 26–35, December 2002.

[25] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Bell Telephone Laboratories, Inc. 1979.

[26] P. Goyal. Personal communication. 2001.

[27] C. Healy., D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 15th IEEE Real-time Systems Symposium*, pages 288–297. December 1995.

[28] J. Hennessy and D. Patterson. Pipelining. *Computer Architecture: A Quantitative Approach*, Chapter 3, pages 125-220. Morgan Kaufmann Publishers, Inc. 1996.

[29] J. Hennessy and D. Patterson. Multiprocessors. *Computer Architecture: A Quantitative Approach*, Chapter 8, pages 635-760. Morgan Kaufmann Publishers, Inc. 1996.

[30] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[31] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, December 2001.

[32] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 111-120, June 2002.

[33] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 149–158, December 2002.

[34] P. Holman and J. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 41–50, July 2003.

[35] P. Holman and J. Anderson. Implementing Pfairness on a symmetric multiprocessor. In *Proceedings of the 10th IEEE Real-time Technology and Applications Symposium*, pages 544–553, May 2004.

[36] Y. Hur, Y.H. Bae, S.-S. Lin, S.-K. Kim, B.-D. Rhee, S.L. Min, C.Y. Park, M. Lee, H. Shin, and C.S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *Proceedings of the Symposium on Real-time Systems*, pages 308–319, December 1995.

[37] I. Ikodinovic, D. Magdic, A. Milenkovic, and V. Milutinovic. Limes: a multiprocessor simulation environment for PC platforms. In *Proceedings of the 3rd International Conference on Parallel Processing and Applied Mathematics*, pp. 398–412, September 1999.

[38] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-time Systems Symposium*, pages 212–221. December 1993.

[39] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 130–140, August 1994.

[40] G. Lamastra, G. Lipari, and Luca Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 151–160. December 2001.

[41] C.-G. Lee, J. Hahn, Y.-M. Seo, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[42] J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, volume 2, pages 37–250. 1982.

[43] Y.-T. Li., S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-time Systems Symposium*, pages 298–307. December 1996.

[44] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 35–43, July 2003.

[45] J. Liu. Clock-driven scheduling. *Real-time systems*, Chapter 3, pages 85-114. Prentice Hall. 2000.

[46] J. Liu. Scheduling aperiodic and sporadic jobs in priority-driven systems. *Real-time systems*, Chapter 7, pages 190-276. Prentice Hall. 2000.

[47] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real–time environment. *Journal of the ACM*, 30:46–61, January 1973.

[48] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the Twentieth IEEE Real-time Systems Symposium*, pages 294–303, December 1999.

[49] A. Mok. Fundamental design problems for the hard real-time environment. Ph.D. Thesis, Massachussetts Institute of Technology. 1983.

[50] A. Mok and X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 128–138, December 2001.

[51] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the Real-time Technology and Applications Symposium*, pages 75–84, May 2001.

[52] M. Papamarcos and J. Patel. A low overhead solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 348–354, June 1984.

[53] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[54] R. Rajkumar. *Synchronization in Real-time systems – A priority inheritance approach.* Kluwer Academic Publishers, Boston, 1991.

[55] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Ninth IEEE Real-time Systems Symposium*, pages 259–269. 1988.

[56] S. Ramamurthy. A lock-free approach to object sharing in real-time systems. Ph.D. Thesis, University of North Carolina at Chapel Hill. 1997.

[57] S. Ramamurthy. Scheduling periodic hard real-time tasks with arbitrary deadlines on multiprocessors. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 59–68, December 2002.

[58] S. Ramamurthy and M. Moir. Static-priority periodic scheduling on multiprocessors. In *Proceedings of the 21st IEEE Real-time Systems Symposium*, pages 69–78, December 2000.

[59] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[60] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-time Systems Symposium*, pp. 2–13, December 2003.

[61] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pp. 189-198, May 2002.

[62] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. Presented at the *11th International Workshop on Parallel and Distributed Real-time Systems* (on CD-ROM), April 2003.

[63] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 51–59, June 2003.

[64] A. Srinivasan, P. Holman, and J. Anderson. Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 19–28, June 2002.

[65] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. Presented at the *11th International Workshop on Parallel and Distributed Real-time Systems* (on CD-ROM), April 2003.

[66] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the Seventh IEEE Real-time Systems Symposium*, pp. 288–299, December 1996.

[67] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 159–166, 1989.

[68] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2, no. 42, pp. 230–265; no. 43, pp. 544–546, 1936.

[69] W. Wolf. Embedded computing. *Computers as Components: Principles of Embedded Computing System Design*, Chapter 1, pages 1-56. Morgan Kaufmann Publishers, Inc. 2001.

[70] *LynxOS Application Writer's Guide*, Lynx Real-time Systems, Inc., 1993.