On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study*

Björn B. Brandenburg, John M. Calandrino, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Multicore platforms are predicted to become significantly larger in the coming years. Given that real-time workloads will inevitably be deployed on such platforms, the scalability *of the scheduling algorithms used to support such workloads warrants investigation. In this paper, this issue is considered and an empirical evaluation of several global and partitioned scheduling algorithms is presented. This evaluation was conducted using a Sun Niagara multicore platform with 32 logical CPUs (eight cores, four hardware threads per core). In this study, each tested algorithm proved to be a viable choice for some subset of the workload categories considered.*

1 Introduction

Most major chip manufacturers have embraced multicore technologies as a way to continue performance improvements in their product lines, given the heat and thermal problems that plague single-core chip designs. To date, several manufacturers have released dual-core chips, Intel and AMD each have quad-core chips on the market, and Sun's Niagara and more recent Niagara 2 systems have eight-core chips with multiple hardware threads per core. In the future, per-chip core counts are expected to increase significantly. Indeed, Intel has announced plans to release chips with as many as 80 cores by 2011 [20].

With the arrival of multicore chips such as those just mentioned, multiprocessors are becoming commonplace in both servers and personal machines. This has led to renewed interest in multiprocessor real-time scheduling. In work on this topic, the greatest focus (by far) has been on algorithmic issues. While clearly important, such research must be augmented by experimental efforts that address implementation concerns. To facilitate this, our research group recently developed a Linux extension called LITMUS^{RT} (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), which allows different (multiprocessor) scheduling algorithms to be linked as plugin components [13, 16, 28]. LITMUS^{RT} has been used by our research group to conduct empirical evaluations of a variety of different scheduling policies [10, 11, 13] and synchronization protocols [12, 14].

Focus of this paper. All of the studies just cited were conducted using a conventional (not multicore) four-processor[1] Intel SMP. While such a platform is probably sufficient for drawing valid conclusions about current multicore designs (as most have modest per-chip core counts), it is not large enough to expose scalability weaknesses that may arise as larger platforms become more common. It is also not sufficient for drawing conclusions about platforms that are architecturally very different from current Intel designs. Additionally, many multicore architectures have hardware features that make them quite different from conventional SMPs. Perhaps the most important such difference is the presence of on-chip shared caches, which can lessen task migration overheads.

Motivated by these observations, we present in this paper a case-study comparison of several multiprocessor scheduling algorithms that was conducted using a Sun UltraSPARC T1 " Niagara" multicore platform. The Niagara is an interesting test platform for several reasons. First, it has 32 logical CPUs (eight cores and four hardware threads per core) and thus would certainly qualify as a "large" platform today. Second, it has features that are likely to be common in future multicore designs, such as hardware multi-threading and on-chip shared-cache support—in the case of the Niagara, a 3 MB L2 cache that is shared by all eight cores. Finally, each core on the Niagara is based on a RISC-like design that is quite different from current Intel designs.

In the study described herein, scheduling algorithms are compared on the basis of real-time *schedulability*, assuming runtime overheads as measured under LITMUS^{RT} on our Niagara test platform. Our usage of this particular test platform has enabled several commonly-raised questions to be addressed for the first time. For example: It is often claimed that global algorithms (see below) will not scale well with increasing processor counts due to run-queue length and contention—if the focus is *schedulability*, is this true? It is likewise often claimed that Pfair algorithms (again, see below) are not practical because they potentially migrate tasks frequently—on a multicore platform with shared caches, is this true? In addition to addressing these and other questions, we also report on changes made to LITMUS^{RT} to rebase it to the Linux 2.6.24 kernel (from 2.6.20) and to port it

*Work supported by IBM, Intel, and Sun Corps., NSF grants CCF 0541056, and CNS 0615197, and ARO grant W911NF-06-1-0425.

[1]We use the terms "core" and "processor" somewhat interchangeably. We often use "core" to emphasize that we are referring to a processor that is one of several on a single chip.

to the Niagara machine. Given the nature of the case study presented here, it is important to acknowledge that the scope of this paper is *specifically* limited to LITMUS$^{\rm RT}$ as implemented on the Niagara. While we do comment on larger multicore chips that we expect other manufacturers such as Intel to release in the future, such comments must naturally be regarded as somewhat speculative at this point.

Scheduling algorithms considered. In this paper, we consider the schedulability of sporadic task systems. A *sporadic task* repeatedly generates work to schedule in the form of sequential *jobs*. A sporadic task system can be scheduled via a *partitioned*, *global*, or *clustered* approach. Under partitioning, tasks are statically assigned to processors and each processor is scheduled separately. Under global scheduling, all jobs are scheduled using a single run queue, and interprocessor migration is allowed. In a clustered approach, tasks are partitioned onto clusters of processors, and within each cluster, assigned tasks are globally scheduled [3, 15].

In our study, we considered the global PD^2 Pfair algorithm [27] and partitioned, global (preemptive and non-preemptive), and clustered variants of the *earliest-deadline-first* (EDF) algorithm. Regarding schedulability, we considered both *hard* real-time systems wherein deadlines should not be missed, and *soft* real-time systems wherein bounded deadline tardiness is permissible. As explained later, the algorithms considered in our study were selected for both theoretical and pragmatic reasons.

Our methodology. The focus of this study is to compare the aforementioned algorithms in terms of schedulability considering real-world overheads. This was done via a four-phase approach: **(i)** we ported LITMUS$^{\rm RT}$ to our Niagara platform and implemented the algorithms under investigation; **(ii)** once the system was stable, we ran several hundred (synthetic) task sets with randomly generated parameters on the test machine and recorded over 70 GB of raw system overhead samples; **(iii)** we analyzed the recorded overhead samples and distilled expressions for average and worst-case overheads; and **(iv)** we conducted simulations in which the schedulability of randomly-generated task sets was checked, with the overheads determined in Step (iii) taken into account, using the best schedulability tests currently known for each tested algorithm.

Summary of results. In the first part of the paper, we describe our efforts to port LITMUS$^{\rm RT}$ to the Linux 2.6.24 kernel and our Niagara platform, and comment on a few newly-added extensions. In the second part of the paper we present our experimental results. As in the prior SMP-based study [16], we found that *for each tested algorithm, scenarios exist in which it is a viable choice*. Partitioned EDF usually performed best for hard real-time systems; clustered EDF usually performed best for soft real-time systems. Perhaps most surprisingly, a "staggered" variant of PD^2 (see Sec. 2) was among the best performing algorithms overall. Its good performance is due to the theoretical optimality of PD^2, which counterbalances the higher runtime overheads of Pfair algorithms. This result calls into question the common belief that global algorithms are not viable on platforms with more than just a few processors.

We present these findings later in Sec. 4 after first providing needed background in Sec. 2 and describing our modifications to LITMUS$^{\rm RT}$ in Sec. 3.

2 Background

We consider the scheduling of a system of *sporadic tasks*, $T_1, \ldots, T_N$, on m processors. The j^{th} job (or invocation) of task T_i is denoted T_i^j. Such a job T_i^j becomes available for execution at its *release time*, $\mathsf{r}(T_i^j)$. Each task T_i is specified by a *worst-case (per-job) execution cost*, $\mathsf{e}(T_i)$, and *period*, $\mathsf{p}(T_i)$. Job T_i^j should complete execution by its *absolute deadline*, $\mathsf{r}(T_i^j) + \mathsf{p}(T_i)$; otherwise, it is *tardy*. The spacing between job releases must satisfy $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$; if it further satisfies $\mathsf{r}(T_i^{j+1}) = \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$, then T_i is called *periodic*. Task T_i's *utilization* or *weight* reflects the processor share it requires and is given by $\mathsf{e}(T_i)/\mathsf{p}(T_i)$. In this paper, we assume that tasks are independent and do not self-suspend.

Scheduling. A *hard* real-time (HRT) system is considered to be *schedulable* iff it can be shown that no job deadline is ever missed. A *soft* real-time (SRT) system is considered (in this paper) to be *schedulable* iff it can be shown that deadline tardiness is bounded. Any algorithm for checking schedulability necessarily depends on the scheduling algorithm being used and must be designed to account for overheads that arise in practice, such as context-switching times, cache-related overheads, *etc.* Such overheads are typically accounted for by inflating per-job execution costs.

As noted earlier, multiprocessor real-time scheduling algorithms can follow a partitioned, global, or clustered approach. Below, we describe the representative algorithms from each category considered in our study. Note that we do not consider static-priority algorithms in this paper. This is intentional: such algorithms are inferior to those described below from the standpoint of schedulability generally [17], and cannot guarantee bounded tardiness without severely restricting overall utilization in the SRT case [19].

EDF scheduling. In EDF scheduling algorithms, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily. In this paper, we consider a *clustered* preemptive EDF algorithm, denoted $\mathsf{C\text{-}EDF}$, in which tasks are partitioned onto clusters of processors and those within each cluster are scheduled globally within it. We also consider two special cases of $\mathsf{C\text{-}EDF}$: *partitioned* EDF, denoted $\mathsf{P\text{-}EDF}$, where each cluster consists of only one processor, and *global* EDF, denoted $\mathsf{G\text{-}EDF}$, where all processors form

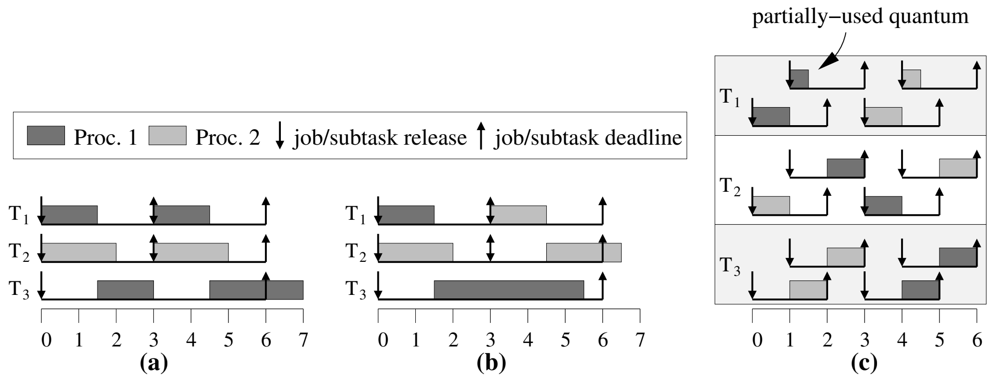

Figure 1: **(a)** G-EDF, **(b)** G-NP-EDF, and **(c)** PD^2 schedules of a two-processor system with three tasks: T_1, with an execution cost of 1.5 and period of 3.0, T_2 with an execution cost of 2.0 and a period of 3.0, and T_3 with an execution cost of 4.0 and a period of 6.0.

one cluster. In addition to these preemptive algorithms, we consider non-preemptive G-EDF, denoted G-NP-EDF. Non-preemptive algorithms allow task migrations, but once a job commences execution on a processor, it runs to completion on that processor without preemption. Thus, *jobs* may not migrate.

Partitioning algorithms require that a bin-packing problem be solved to assign tasks to processors. As a result, there exist task systems with total utilization slightly higher than $m/2$ that no such algorithm can schedule, *even if bounded deadline tardiness is allowed* [19]. Thus, under P-EDF, restrictive caps on total utilization are required to ensure timing constraints, for both HRT and SRT systems. Similar caps are required for G-EDF in the HRT case. However, in the SRT case, G-EDF ensures bounded deadline tardiness, as long as total utilization is at most m [19]. The same is true of G-NP-EDF, which is significant, because under it, overheads are lower (since jobs do not migrate). C-EDF was proposed as a compromise between P-EDF and G-EDF for large multicore platforms in which some cores share a high-level cache, while others share only a low-level cache (or no cache) [15]. In such a platform, it is best to consider each set of cores that share a high-level cache to be a "cluster." In this way, lower migration costs can be ensured than for G-EDF and the bin-packing limitations of P-EDF are ameliorated. Regarding the latter, note that each cluster is viewed as a bin; thus, clustering increases bin sizes and decreases the number of bins, which makes bin-packing easier.

Pfair scheduling. Pfair scheduling algorithms [6, 26] can be used to optimally schedule HRT (and thus SRT) systems, provided all timing parameters are multiples of the system's quantum size. The term "optimal" means that no deadline is missed when scheduling any feasible system (*i.e.*, any system with total utilization at most m). In a Pfair algorithm, a job T_i^j of a task T_i of weight $\mathsf{wt}(T_i)$ is scheduled one quantum at a time in a way that approximates an *ideal* allocation in which it receives $L{\cdot}\mathsf{wt}(T_i)$ time over any interval of length L within $[\mathsf{r}(T_i^j), \mathsf{r}(T_i^j) + \mathsf{p}(T_i))$. This is accomplished by sub-dividing each task into a sequence of quantum-length *subtasks*, each of which must execute within a certain time *window*, the end of which is its *deadline*. Subtasks are scheduled on an EDF basis, and tie-breaking rules are used in case of a deadline tie. A task's subtasks may execute on any processor, but not at the same time (tasks are sequential). In this paper, we consider the PD^2 [1, 26] algorithm, which is the most efficient known optimal Pfair algorithm, and a variant of it called *staggered* PD^2 (S-PD^2) [23]. Under PD^2 (and most other Pfair algorithms), quanta are assumed to align across processors, *i.e.*, all processors cross the same quantum boundary at the same time. With S-PD^2, quanta are staggered across processors, *i.e.*, quantum boundaries are spread out evenly across a full quantum. Under both PD^2 and S-PD^2, if a task is allocated a quantum when it requires less execution time, the unused part of the quantum is "wasted." In contrast, under the EDF schemes considered above, such a task would relinquish its assigned quantum "early," allowing another task to be scheduled.

Example. To see some of the differences in the algorithms described above, consider Fig. 1, which depicts two-processor G-EDF, G-NP-EDF, and PD^2 schedules for a system of three tasks, T_1, T_2, and T_3, as defined in the figure's caption. There are several things worth noting here. First, these three tasks cannot be partitioned onto two processors, so this system is not schedulable under P-EDF (so we do not depict a schedule for this case). (Recall that C-EDF generalizes both G-EDF and P-EDF.) Second, under each of G-EDF and G-NP-EDF a deadline is missed. Third, in the G-NP-EDF schedule in inset (b), task T_2's second job cannot execute at time 3 since T_3's job must execute non-preemptively (there is actually a deadline tie here). Fourth, each task has the same window structure in inset (c). For tasks T_2 and T_3, this is easily explained: a task's window structure is determined by its weight and both of these tasks have a weight of 2/3. As for task T_1, under Pfair scheduling, windows are defined by assuming that each task's execution cost is an integral number of quanta. Thus, we must round up T_1's cost to 2.0, giving it a weight of 2/3. Because of this, some quanta allocated to task T_1 are only half-used.

Earlier experimental study. In the earlier SMP-based study [16], all of the algorithms described above were considered, except C-EDF. In this prior study, P-EDF and PD^2 tended to be the best choices for HRT workloads. This is due to the lower runtime overheads of P-EDF (because tasks do not migrate and processors are scheduled using uniprocessor algorithms) and the optimality of PD^2. For SRT workloads, G-EDF and G-NP-EDF were usually preferable. This is because these algorithms do not require restrictive utilization caps in this case, and on the platform considered in [16], were found to have lower runtime overheads than Pfair algorithms. In this paper, we seek to re-examine these algorithms on a "large" multicore platform with shared caches and also determine the viability of clustered approaches.

3 Implementation

Before presenting our experimental results, we briefly discuss our efforts in re-basing LITMUS$^{\mathrm{RT}}$ to Linux 2.6.24, porting it to the Niagara, and adding clustered-scheduling support.

LITMUS$^{\mathrm{RT}}$ design. The pre-existing LITMUS$^{\mathrm{RT}}$ code base that was modified is based on Linux 2.6.20. A detailed description of that LITMUS$^{\mathrm{RT}}$ version can be found in [13]. Here, we provide a brief description. LITMUS$^{\mathrm{RT}}$ has been implemented via changes to the Linux kernel and the creation of user-space libraries. Since LITMUS$^{\mathrm{RT}}$ is concerned with real-time scheduling, most kernel changes affect the scheduler and timer-interrupt code. The kernel modifications can be split into roughly three components. The *core infrastructure* consists of modifications to the Linux scheduler, as well as support structures and services such as tracing and priority queues. The scheduling policies are implemented as *scheduler plugins* that provide functions that implement needed scheduling-related methods. Finally, a collection of *system calls* provides an API for real-time tasks to interact with the kernel. This basic structure has been retained in the new version of LITMUS$^{\mathrm{RT}}$. Some of the more notable changes are discussed next.

Integrating with the new Linux scheduler. The new Linux scheduler, introduced in version 2.6.23, selects a task to schedule by traversing a list of *scheduling classes* and selecting the highest-priority task from the first non-idle class. In stock Linux, there are three classes: POSIX RT, CFS, and IDLE (listed in order of decreasing priority). We integrated LITMUS$^{\mathrm{RT}}$ within Linux by inserting a new LITMUS$^{\mathrm{RT}}$ scheduling class as the top scheduling class. Thus, jobs scheduled by LITMUS$^{\mathrm{RT}}$ always have higher priority than other (background) jobs.

While scheduling classes provide a basic extension mechanism, implementing global schedulers is still difficult because Linux fundamentally assumes partitioning—all changes to a task's state are serialized through the lock that protects the run queue where the task resides (there is one run queue per processor). Thus, in order for a task to migrate, two run queue locks (source and target run queue) must be held. The LITMUS$^{\mathrm{RT}}$ core needs to migrate a task when a global scheduler plugin selects a real-time task that has executed previously on a different processor. However, the Linux scheduler acquires the local (target) run queue lock *before* invoking the scheduling classes. If the LITMUS$^{\mathrm{RT}}$ migration code were to simply acquire the source's run queue lock, then deadlock could quickly ensue. Instead, the local run queue's lock must be released before both locks can be acquired together (in a specific order). This creates a race window, between the invocation of the active plugin and the actual context switch, in which the state of both local tasks and the migrating task can change. For example, if the previously-scheduled task blocked (possibly due to loading shared libraries, *etc.*) and resumes in the short time while its run queue lock is available, then it may not be noticed by the scheduler and be "forgotten" (*i.e.*, it may never be scheduled again) since the stock Linux scheduler does not expect tasks to resume while scheduling is in progress (the run queue lock normally protects against this scenario). Such untimely task state changes and similar races can be detected (and properly handled) by examining the state of the involved tasks prior to releasing and after reacquiring the local run queue lock.

Job releases. Sporadic job releases are now controlled through the use of the high-resolution timers that were introduced in version 2.6.16 of Linux. Such timers allow events to occur at an exact time as the result of a timer interrupt. The times at which events can be specified to occur is limited only by the precision and accuracy of the underlying timer hardware. As a result, job releases can now occur within a few hundred nanoseconds of any specified release time, and periods do not have to be a multiple of the quantum size.

Quantum alignment. In Linux, a *quantum* is defined to be the interval between two *ticks*, which are periodic timer interrupts. In LITMUS$^{\mathrm{RT}}$, the quantum length is set at $1ms$. Some scheduling policies implemented in LITMUS$^{\mathrm{RT}}$ (*e.g.*, PD^2) require that quanta are aligned across CPUs. This provides a unified and consistent notion of time across the system. The Linux kernel uses high-resolution timers (when such hardware is available) to set up a periodic tick on each CPU in such a way that interrupts are uniformly distributed over the first half of the tick length—the result is a form of staggered quanta. We modified the tick initialization code to optionally have ticks occur at the same time on each CPU, resulting in aligned quanta, or to stagger them evenly across entire quanta. The choice between aligned or staggered quanta is made at boot time.

Clustered scheduling. Clustered scheduling was implemented as a hybrid of the existing G-EDF and P-EDF plugins. Individual ready queues are used as in P-EDF, but with one ready queue per cluster. Within each scheduling domain, the G-EDF implementation is used to make scheduling decisions. In the current implementation, a cluster consists of the four hardware threads of one core (since they share an L1 cache). In the rest of the paper, references to "C-EDF" are assumed to refer to this variant. In a future version, we plan to support runtime changes to the cluster size (when there is no real-time workload present).

Other LITMUS$^{\mathrm{RT}}$ updates. Various other updates have taken place since the last revision of LITMUS$^{\mathrm{RT}}$. These updates include the following. First, we now allow scheduling policies to be changed without rebooting whenever there is no real-time workload present. Second, task execution costs and periods are now specified in nanoseconds, so that more accurate methods of handling job releases and task overruns can be employed (such as the use of high-resolution timers for job releases, discussed earlier). Third, for increased timing accuracy, the notion of time used in LITMUS$^{\mathrm{RT}}$ is now that provided by the high-resolution timers. Fourth, we now allow both processes and threads to be used to define LITMUS$^{\mathrm{RT}}$ tasks—processes have independent address spaces, while threads share the same address space. Fifth, we rewrote the PD^2 plugin to enable sporadic job releases (previously, only periodic tasks were supported). Sixth, we implemented FIFO ticket locks for the `sparc64` architecture (stock Linux provides ticket locks only on Intel's `x86` architecture). Seventh, we introduced proper support for synchronous and asynchronous task-system initialization. The new API allows real-time tasks to finish their initialization and then wait for the task-system release (phases can be specified for tasks to support asynchronous releases). In previous versions of LITMUS$^{\mathrm{RT}}$, a more ad hoc approach was used that involved transitioning the system between non-real-time and real-time mode.

Niagara-related challenges. In performing the above changes, several Niagara-related challenges arose. First, several Linux tools that are typically used for debugging were either not available or broken on the Niagara. (These problems will hopefully be fixed in mainline Linux soon.) Most notable was `lockdep`, a debugging facility that detects potential causes of kernel deadlock at runtime. An alternative is to use `netconsole`, which allows debugging messages to be sent to another machine before the local console freezes (bugs in scheduling code often bring down the system before any debug output can be written locally). However, `netconsole` could not be reliably used due to memory management bugs in the underlying network driver.

Second, the Niagara uses a weak memory consistency model. To deal with this, memory barrier instructions had to be added to the existing LITMUS$^{\mathrm{RT}}$ code. Finally, the different architecture, higher degree of parallelism, and slower core speeds of the Niagara (as compared to our prior SMP testbed) caused several race conditions to be exposed that were actually present in the pre-existing LITMUS$^{\mathrm{RT}}$ code base (but never triggered in testing). Dealing with all of these various issues was made more challenging by the lack of good debugging support.

4 Experiments

In this section, we report on the results of experiments conducted using LITMUS$^{\mathrm{RT}}$ to compare the scheduling algorithms introduced previously. We compared these algorithms on the basis of both schedulability and tardiness, under consideration of various overheads determined empirically on our test platform.

4.1 Overheads

In real systems, task execution times are affected by the following sources of overhead. At the beginning of each quantum, *tick scheduling overhead* is incurred, which is the time needed to service a timer interrupt. When a job is released, *release overhead* is incurred, which is the time needed to service the interrupt routine that is responsible for releasing jobs at the correct times. Whenever a scheduling decision is made, a *scheduling overhead* is incurred, which is the time taken to select the next job to schedule. Whenever a job is preempted, *context-switching overhead* is incurred, as is either *preemption* or *migration overhead*; the former term includes any non-cache-related costs associated with the preemption, while the latter two terms account for any costs due to a loss of cache affinity. Preemption (resp., migration) overhead is incurred if the preempted job later resumes execution on the same (resp., a different) processor.

Limitations of real-time Linux. To satisfy the strict definition of HRT, all worst-case overheads must be known in advance and accounted for. Unfortunately, this is currently not possible in Linux, and it is highly unlikely that it ever will be.[2] This is due to the many sources of unpredictability within Linux (such as interrupt handlers and priority inversions within the kernel), as well as the lack of determinism on the hardware platforms on which Linux typically runs. The latter is especially a concern, regardless of the OS, on multiprocessor platforms. Indeed, research on timing analysis has not matured to the point of being able to analyze

[2]By "Linux," we mean modified versions of the stock Linux kernel with improved real-time capability, not paravirtualized variants such as RTLinux [29] or L^{4}Linux [22], where real-time tasks are not actually Linux tasks. Stronger notions of HRT can be provided in such systems, at the expense of a more restricted and less familiar development environment.

Plugin	Tick AVG	Schedule AVG	Context Switch AVG	Release AVG
PD^2	$4.336 + 0.026N$	4.667	$2.597 + 0.001N$	—
S-PD^2	$2.169 + 0.016N$	4.247	$2.539 + 0.001N$	—
G-EDF	$2.080 + 0.002N$	$11.836 + 0.056N$	7.550	$5.840 + 0.127N$
C-EDF	2.863	$6.080 + 0.010N$	3.174	16.533
P-EDF	$2.059 + 0.002N$	$2.678 + 0.008N$	$4.728 + 0.005N$	$3.952 + 0.005N$

Plugin	Tick WC	Schedule WC	Context Switch WC	Release WC
PD^2	$11.271 + 0.301N$	31.720	$3.107 + 0.010N$	—
S-PD^2	$4.838 + 0.321N$	43.182	$3.219 + 0.003N$	—
G-EDF	$3.043 + 0.003N$	$55.284 + 0.263N$	29.233	$45.025 + 0.314N$
C-EDF	3.230	$14.758 + 0.011N$	6.077	30.347
P-EDF	$2.727 + 0.002N$	$8.565 + 0.014N$	$14.917 + 0.036N$	$4.727 + 0.009N$

Table 1: Measured average (SRT) and worst-case (HRT) kernel overheads, in μs. N is the number of tasks. In our PD^2 and S-PD^2 implementation, periodic tasks do not incur release overheads (see Sec. 4.4).

complex interactions between tasks due to atomic operations, bus locking, and bus and cache contention. Despite these observations, there are now many advocates of using Linux to support applications that require some notion of real-time execution. As noted by McKenney [25],

> *I believe that Linux is ready to handle applications requiring sub-millisecond process-scheduling and interrupt latencies with 99.99+ percent probabilities of success. No, that does not cover every imaginable real-time application, but it does cover a very large and important subset.*

Our objectives in designing LITMUS$^{\mathrm{RT}}$ are in agreement with McKenney's viewpoint.

Measuring overheads. Given the comments above about timing analysis, overheads must be determined experimentally. Doing so is not as easy as it may seem. In particular, in repeated measurements of some overhead, a small number of samples may be "outliers." This may be due to a variety of factors, such as warm-up effects in the instrumentation code and the various non-deterministic aspects of Linux itself noted above. In light of this, we determined each overhead term by discarding the top 1% of measured values, and then taking the maximum (for HRT) or average (for SRT) of the remaining values. This same approach was used previously in [14]. We believe that this is approach is reasonable, given the way in which Linux-based systems are likely to be used in supporting real-time workloads. Moreover, the obtained overheads should be more than sufficient for obtaining a valid comparison of different scheduling approaches, which is our main focus.

All overheads were measured on the Sun Niagara (UltraSPARC T1) system mentioned earlier. The Niagara is a 64-bit machine containing eight cores on one chip running at 1.2 GHz. Each core supports four hardware threads. On-chip caches include a 16K (resp., 8K) four-way set associative L1 instruction (resp., data) cache per core, and a shared, unified 3 MB 12-way set associative L2 cache. Our test system is configured with 16 GB of off-chip main memory. In contrast to Sun's upcoming Niagara-successor "Rock," our first-generation Niagara does not employ advanced cache-prefetching technology.

Below, we discuss the overheads that were measured. In this discussion, G-NP-EDF is not considered because it shares a common implementation with G-EDF (non-preemptivity is enabled by simply setting a flag [13]) and thus they have the same overheads.

Kernel overheads. We obtained kernel overheads by enabling aligned quanta and measuring the system's behavior for periodic task sets consisting of between 50 and 450 tasks in steps of 50. For each scheduling algorithm and task-set size, we measured ten task sets generated randomly (as described in Sec. 4.2), for a total of 90 task sets per scheduling algorithm. Each task set was traced for 30 seconds. We repeated the same experiments with staggered quanta. In total, more than 45 GB of trace data and 600 million individual overhead measurements were obtained during more than seven hours of tracing.

For each overhead term, we plotted all measured values for both aligned and staggered quanta as a function of task-set size (discarding outliers, as discussed above). We noted that average overheads were substantially less for staggered quanta. This is due to reduced bus/lock contention in this case (since kernel invocations become more spread out). Given that we assume average overheads for SRT systems, we opted to use average overheads obtained with staggered quanta for SRT systems (except for PD^2). Staggering causes an additional quantum of potential tardiness, which we regarded to be of negligible impact. As tardiness should be zero in HRT systems, worst-case overheads were defined assuming aligned quanta (except for S-PD^2). When plotted, most overheads showed a clear linear trend. To capture these trends, we computed expressions for maximum and average values using linear regression analysis. For the few overheads where measurements did not reveal a conclusive trend, we assumed a constant value equal to the maximum observed value. All kernel overhead expression so computed are shown in Table 1.

Algorithm	Overall AVG	Overall WC	Preemption AVG	Preemption WC	Intra-Cluster Migr. AVG	Intra-Cluster Migr. WC	Inter-Cluster Migr. AVG	Inter-Cluster Migr. WC
PD^2	172.05	681.11	131.46	649.43	141.82	654.22	187.62	681.11
S-PD^2	89.396	104.13	86.214	103.37	87.844	103.44	90.233	104.13
G-EDF	73.047	375.44	95.196	375.44	73.454	326.82	72.592	321.14
C-EDF	67.047	171.62	78.507	171.62	64.772	167.34	—	—
P-EDF	72.362	139.12	72.362	139.12	—	—	—	—

Table 2: Measured average (SRT) and worst-case (HRT) preemption and migration costs, in μs. Note that, while PD^2, S-PD^2, and G-EDF are not clustered algorithms, we still distinguish between migrations within a cluster (staying on the same core) and across clusters (moving to a different core). In our schedulability experiments, the "overall" figures were used since in global algorithms it is generally not possible to predict if a preempted task will migrate.

Preemption and migration overheads. To measure the impact of losing cache affinity, we ran a number of task sets under LITMUS$^{\mathrm{RT}}$ in which each job repeatedly accesses a 64K working set (WS) while maintaining a 75/25 read/write ratio. We found that 64K WS sizes produced significant differences in the overheads measured for each algorithm. We considered larger WS sizes to be unreasonable given the memory access times and shared cache size of our platform. Smaller WS sizes would likely result in smaller overheads and better overall performance for global policies, but are not considered here—thus, our experiments could be seen as being biased towards partitioning.

After the first access by a job of its entire WS, subsequent references are cache-warm, until a preemption or migration occurs. Tasks were instrumented to detect and record these situations and the corresponding access times for the entire WS. The time to reference the WS immediately after resuming execution, minus the time to reference the WS when it is cache-warm, gives the cost of the preemption or migration (depending on whether the corresponding job resumed on a different CPU) that occurred. In total, over 105 million preemption and migration overheads were recorded for task sets consisting of between 50 and 450 tasks run for 60 seconds each. Again, we measured a total of 90 task sets per scheduling algorithm. (Aligned quanta were used for all algorithms except S-PD^2. Quantum placement does not affect preemption behavior in EDF schemes.) As noted earlier, we determined overheads for HRT (resp., SRT) systems by taking the maximum (resp., average) of the recorded values after discarding the top 1% of measured values. In the case of preemption/migration overheads, we also discarded any value that clearly was not correctly measured (*e.g.*, due to a preemption *while* measuring an overhead).

Table 2 indicates that the preemption and migration overheads are considerably higher than in the Intel platform considered in the earlier SMP study [16]. The Niagara does not perform any branch prediction or attempt to hide costs due to memory latency or pipeline stalls by allowing out-of-order instruction execution. Hardware multi-threading is intended as the main mechanism for alleviating these costs. However, while hardware multi-threading may increase overall throughput, it does not improve the performance of a single *sequential* task. These costs, combined with the low per-core clock frequencies relative to most Intel platforms, cause the Niagara to execute tasks much slower than any current Intel platform even in the absence of preemptions and migrations.

Note that, while the Niagara contains a shared on-chip lower-level cache, the size of this cache (3 MB) is quite small relative to those in the latest Intel multicore chips, where shared caches are often 2-3 MB *per-core* (*e.g.*, four cores might share a 12 MB cache). Thus, the impact of a shared cache on migration costs is relatively small, even with 64K per-job WS sizes, since cache affinity can be lost after relatively short preemptions. Additionally, *contention* for the bus and memory subsystem appears to be a major source of overhead—this is especially apparent when comparing the worst-case overheads of PD^2 and S-PD^2. In PD^2, all preempted jobs access their WS at the same time upon resuming (or beginning) execution, since all processors make scheduling decisions at the same time, and this results in significant worst-case contention on such a large platform.

In the average case, the overheads under each EDF algorithm are similar. We believe that the fluctuations in this case are due to the variation in the task sets that are run under each algorithm, and the non-determinism associated with attempting to measure these overheads within user-space tasks (these were the only overheads that could not be measured entirely within the kernel). In the case of PD^2, contention appears to have an impact even on average-case results—since scheduling decisions are only made at quantum boundaries, the chance that multiple tasks resume execution at the same time increases significantly, resulting in a corresponding increase in contention and overheads. In S-PD^2, scheduling decisions are evenly distributed across each quantum, which reduces contention both in the worst and average cases, resulting in lower overheads as compared to PD^2. It also results in the lowest worst-case overhead overall, as it is the only approach that *forces* scheduling decisions, and thus the times that jobs resume after being preempted, to be evenly distributed over time—this keeps contention low in all cases.

Interestingly, for G-EDF and C-EDF, the cost of a preemption is slightly *more* than the cost of a migration. We believe that this is due to the length of time that a job is preempted. Preemption lengths were longer when a task resumed on the same CPU (instead of a different CPU), and longer preemption lengths cause a greater loss of cache affinity and hence higher overheads. This makes sense under G-EDF and C-EDF: a job that is preempted will only migrate to another CPU if one becomes available before its current CPU is again available. Thus, when a job migrates, the total length of its preemption is reduced. In G-EDF in particular, an inability to migrate to another CPU to resume execution upon being preempted is rare on such a large platform, so such scenarios had a small impact on overall average-case costs. Interestingly, these trends were not observed for PD^2 or S-PD^2—since preemption lengths did not vary much for these algorithms, the bus and memory subsystem contention costs associated with migrations remain the dominant factor. Preemption lengths tend to be more uniform in these algorithms, due to the way in which jobs are scheduled as subtasks that only become eligible a fixed amount of time after a job release. As a final observation, note that costs are larger overall for algorithms that *allow* migrations, due to an increase in contention that impacts the preemption and migration overheads of all jobs, even those that do not migrate.

4.2 Experimental Set-Up

We determined the schedulability of randomly-generated task sets under each tested algorithm, for both HRT and SRT systems, using the overheads listed in Sec. 4.1. We used distributions proposed by Baker [2] to generate task sets. Task periods were uniformly distributed over $[10ms, 100ms]$. Task utilizations were distributed differently for each experiment using three uniform and three bimodal distributions. The ranges for the uniform distributions were [0.001, 0.1] (light), [0.1, 0.4] (medium), and [0.5, 0.9] (heavy). In the three bimodal distributions, utilizations were distributed uniformly over either [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy). Task execution costs excluding overheads were calculated from periods and utilizations (and may be non-integral). Each task set was created by generating tasks until a specified cap on total utilization (that varied between 1 and 32) was reached and by then discarding the last-added task, thereby allowing some slack to account for overheads. Sampling points (on average spaced 0.13 apart) were chosen such that sampling density is high in areas where curves change rapidly. For each distribution and each sampling point, we generated 2000 task sets, for a total of over 8.5 million task sets.

Schedulability tests. Schedulability was checked using schedulability tests that were augmented to account for the overheads listed in Sec. 4.1. Overheads were accounted for using standard techniques for inflating task execution costs (see [18] for an in-depth discussion). With the exception of C-EDF and G-EDF, the schedulability tests used in our experiments (which we briefly summarize below) are the same as those used in the earlier SMP-based study [16].

For PD^2 and S-PD^2, we simply checked if total utilization, including overheads, is at most m. Note that, under both Pfair schemes, execution costs must be rounded up to integral values after overheads are included. Additionally, for HRT correctness, periods must be reduced by one since sporadic releases only take effect at quantum boundaries. To compensate for the worst-case delay due to staggering under S-PD^2, task periods must be further reduced by $\frac{m-1}{m}$ (in HRT systems). Period reduction is not required for SRT correctness since it introduces only constant tardiness.

For P-EDF and C-EDF, we determined whether each task set could be partitioned using the *first-fit decreasing heuristic*. Under P-EDF, HRT and SRT schedulability differ only in the use of maximum or average overheads: under partitioning, if tardiness is bounded, then it is zero, so the only way to schedule a SRT task set is to view it as hard. Under C-EDF, schedulability within each cluster was checked by applying the appropriate G-EDF test(s) (HRT or SRT) within the cluster (see below). There exists a cyclic dependency between partitioning a task set and accounting for overheads. In our experiments, we first tried to partition and then inflate for overheads. If the inflation caused a processor (or cluster) to become over-utilized, we then tried to first inflate and then partition. Not that testing HRT schedulability by partitioning gives P-EDF a significant advantage since this method is considerably less pessimistic than applying closed-form tests.

New HRT schedulability tests for G-EDF have been published since the prior SMP-based study [16]. There now are five major sufficient (but not necessary) HRT schedulability tests for G-EDF [4, 5, 7, 8, 21]. Interestingly, for each of these tests, there exist task sets that are deemed schedulable by it but not the others [5, 8]. Thus, HRT schedulability under G-EDF and C-EDF was determined by testing whether a given task set passes at least one of these five tests. We omit G-NP-EDF from our HRT schedulability results because there currently exists no good HRT schedulability test for it. For SRT schedulability, since G-EDF and G-NP-EDF can guarantee bounded deadline tardiness if the system is not overloaded [18], only a check that total utilization is at most m is required.

4.3 Schedulability Results

HRT schedulability results are shown in Fig. 2. The first column of the figure (insets (a)–(c)) gives results for the three uniform distributions (light, medium, heavy) and the second column (insets (d)–(f)) gives results for the three bimodal distributions. The plots indicate the fraction of the generated task sets each algorithm successfully scheduled, as a function of total utilization. In the HRT case, aligned quanta are

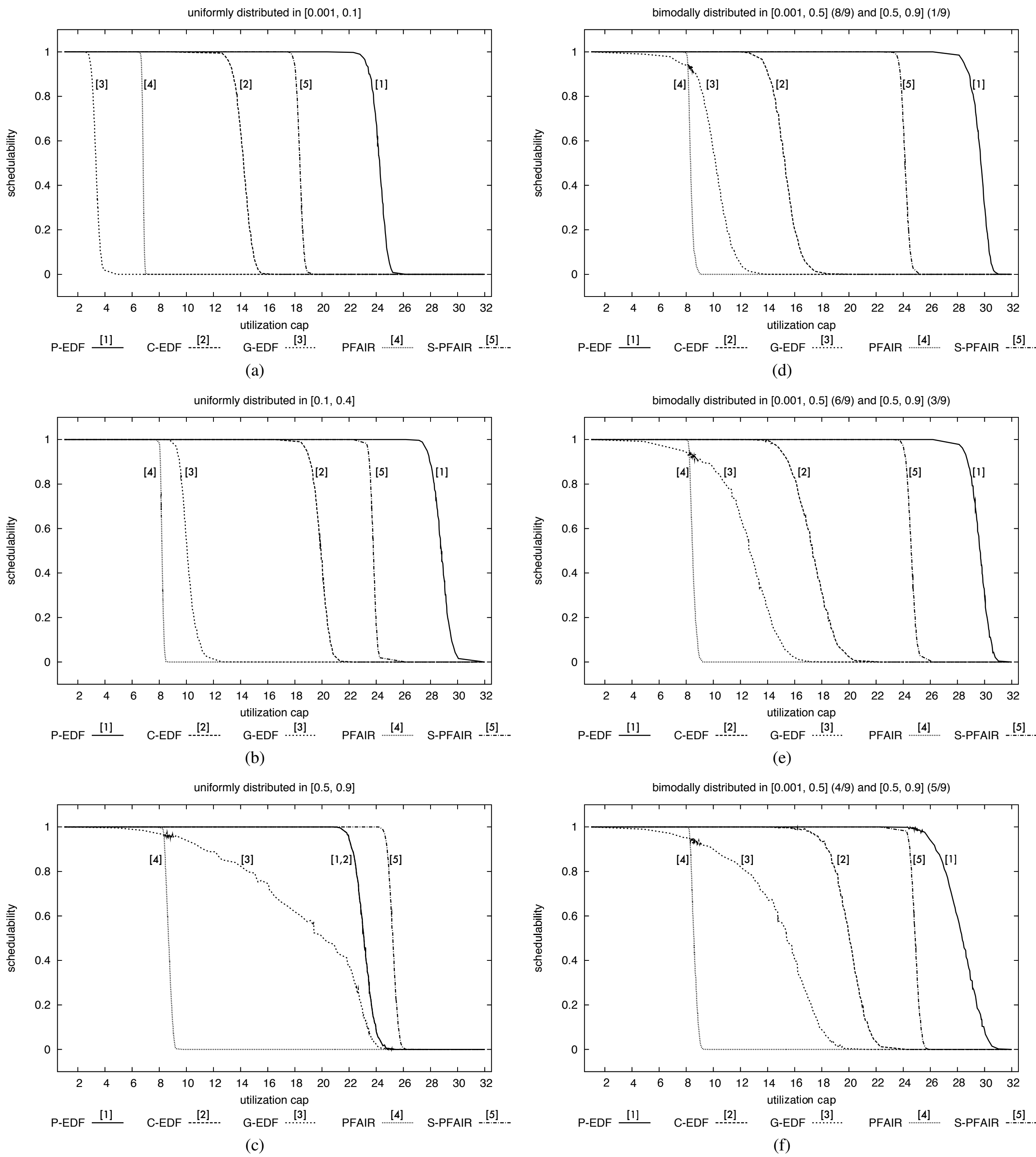

Figure 2: Hard real-time schedulability under various per-task weight distributions. **(a)** Uniform light. **(b)** Uniform medium. **(c)** Uniform heavy. **(d)** Bimodal light. **(e)** Bimodal medium. **(f)** Bimodal heavy.

assumed for all algorithms except for S-PD^2.

There are several things to notice here. First, PD^2 exhibits untenable performance due to its high preemption and migration costs. By preempting and scheduling subtasks at the same instant on all processors, PD^2 exposes the off-chip memory bandwidth to be a critical bottleneck. In contrast, S-PD^2—specifically designed to avoid bus contention—exhibits consistently good performance, suggesting that its slight tradeoff in optimality for reduced runtime overheads yields a viable alternative to partitioning. Second, G-EDF is consistent as well, but consistently poor. In the HRT case, the schedulability tests used for G-EDF are inferior to those used for the other schemes. Moreover, it suffers from high worst-case scheduling and migration costs. The former is especially significant when many light tasks are present (insets (a) and (d)) because in such cases there are comparatively more tasks overall. Third, P-EDF performs best, except when many heavy tasks are present (inset (c)). P-EDF benefits from low run-time overheads but suffers from bin-packing issues, which are somewhat ameliorated by using a bin-packing heuristic, instead of a closed-form test. Note that it outperforms S-PD^2 except for when most tasks are very heavy (inset (c)). Finally, notice that C-EDF exhibits performance that is intermediate between G-EDF and P-EDF, as might be expected.

Schedulability results for the SRT case are shown in Fig. 3, which is organized similarly to Fig. 2. In the SRT case, staggered quanta are assumed for all algorithms except PD^2, which we have included for the sake of completeness. Again, there are several interesting things to notice. First, G-EDF is very negatively impacted by its higher scheduling overheads. This can be seen clearly in insets (a), (b), and (d). In these cases, most tasks are light, which means that the overall number of tasks is high. Note that, in contrast to the HRT case, G-EDF is not negatively impacted here by the schedulability test used for it. Second, G-NP-EDF does not exhibit significantly better performance than G-EDF (in fact, the graphs almost coincide in inset (a)). This strongly suggests that preemption and migration costs are not the limiting factor for the G-EDF family of scheduling algorithms (as implemented in LITMUS$^{\mathrm{RT}}$). Rather, they are limited by scheduling overheads. Third, P-EDF performs correspondingly poorly when task utilizations are high (inset (c)), due to bin-packing problems. Note that, for P-EDF, the only difference between Figs. 2 and 3 is that worst-case overheads are assumed in the former, and average-case in the latter. Fourth, C-EDF performs best in most cases (insets (b) and (d)-(f)), since it has lower runtime overheads than G-EDF and does not suffer from bin-packing limitations as severe as P-EDF. Consequently, its performance degrades when there are either very many light tasks (inset (a)) or mostly heavy tasks (inset (c)), but not as much as the performance of G-EDF and P-EDF, respectively. Fifth, S-PD^2 is usually among the better performing algorithms and the only algorithm that does not exhibit pathological scenarios. While PD^2 performs significantly better than in the HRT case, it is usually among the worst algorithms. This is due to PD^2 using aligned quanta and hence having higher overheads.

In the SRT case, tardiness is also of interest. Fig. 4 shows the average of the per-task worst-case tardiness bounds (computed using formulas in [19]) for the scenario in Fig. 3(b). (PD^2, S-PD^2, and P-EDF are not shown, because assuming staggered quanta, tardiness under each is at most one or two quanta.) As seen, tardiness is highest under G-NP-EDF and lowest under C-EDF. The "stair-step" pattern for C-EDF is due to a ceiling operator in the tardiness-bound expression.

4.4 Discussion

While PD^2 and G-EDF (and correspondingly G-NP-EDF) are both global algorithms, they are implemented quite differently in LITMUS$^{\mathrm{RT}}$. Our G-EDF implementation is mostly event-based. Job releases are implemented by using one-shot Linux high-resolution timers. Moreover, the scheduler may be invoked from different processors in reacting to these and other events. This design enables arbitrary task periods to be supported and job releases to be very accurately timed. One timer is used for each release time; if multiple releases coincide (*e.g.*, at a hyperperiod boundary) one timer is shared and only one release event is handled by the scheduler, which enables merging the released jobs into the ready queue in $O(\log N)$ time. In contrast, a simpler implementation that does not share timers has to perform $O(N)$ individual merges. We found that sharing timers reduces measured worst-case release overheads significantly (in some cases by up to 50 percent). However, our experiments revealed that G-EDF still suffers from high overheads due to long queue lengths and lock-contention. We are currently investigating approaches to mitigate these sources of overhead.

In contrast, the PD^2 plugin is very much time-driven. The scheduling logic is contained mostly in the tick handler, which is invoked at every quantum boundary. Since PD^2 scheduling decisions are made at well-known times, a master-slave approach is used where only one processor (the master) schedules the next quantum; this eliminates any potential run-queue contention. Moreover, queue-merge operations can be performed to add newly-released work to the run queue in $O(\log N)$ time. Such a design also has the advantage that strictly-periodic execution does not require per-release timers. However, one downside of the PD^2 implementation is that a sporadic job can experience up to one quantum of delay (the time to the next quantum boundary) before it becomes eligible for scheduling. (Since PD^2 and S-PD^2 are realized by the same implementation, similar

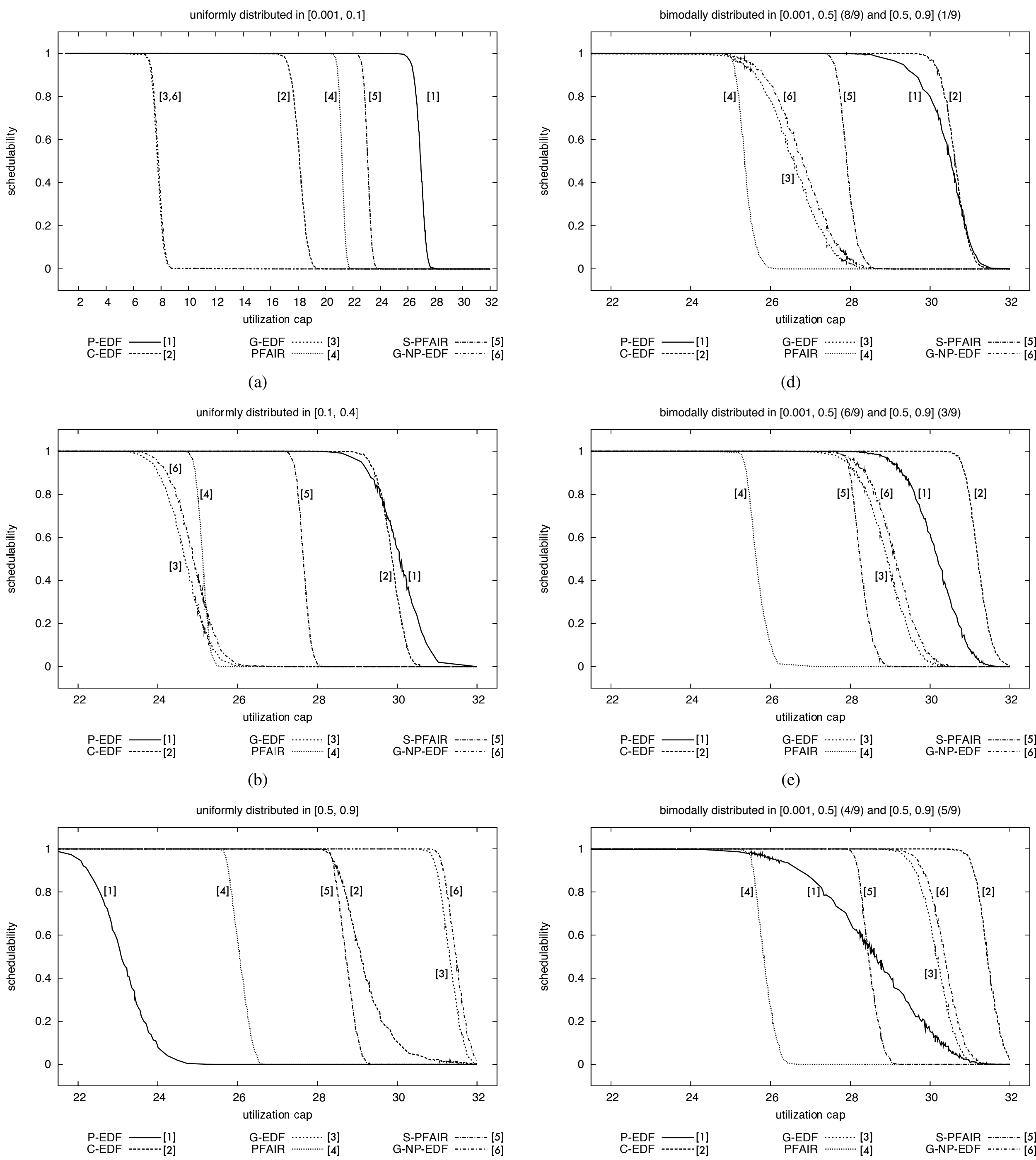

Figure 3: Soft real-time schedulability under various per-task weight distributions. **(a)** Uniform light. **(b)** Uniform medium. **(c)** Uniform heavy. **(d)** Bimodal light. **(e)** Bimodal medium. **(f)** Bimodal heavy. (Please note the different scale of (a). Also, note that the G-EDF and G-NP-EDF curves nearly coincide.)

comments apply to S-PD^2.)

Given that S-PD^2 has much higher tick overheads than G-EDF, and since ticks occur frequently, one might expect G-EDF to outperform S-PD^2 for SRT systems. However, our experiments show that this is not the case. In fact, G-EDF is only competitive in scenarios with low task counts (*e.g.*, insets (c), (e), and (f) of Fig. 3). This is due to the manner in which release overheads are accounted for in schedulability analysis. Linux's high-resolution timers are interrupt-based, and since interrupts always preempt a real-time task, a job can—in the worst case—be delayed by job releases from *all* other tasks. Since the standard analysis for interrupts [24] inflates execution costs on a per-task basis, this amounts to an overhead penalty of $O(N^2)$. This is less of an issue for P-EDF and C-EDF since their analysis only needs to account for interrupts from (fewer) local tasks. Also, due to reduced lock contention, release overheads are significantly lower with partitioned algorithms. Since PD^2 does not use high-resolution timers (apart from the periodic tick), it avoids these interrupt overheads altogether.

In our initial attempt at porting LITMUS$^{\mathrm{RT}}$ to the Niagara, all run queues were implemented using linked lists. However, such an implementation quickly proved to be untenable for several of the global algorithms. Experiments showed that a simple traversal of a 500-element list can take up to $50\mu s$ on the Niagara—clearly too much considering that the PD^2 plugin may have to perform 32 traversals every $1000\mu s$. Such problems necessitated a switch to binomial heaps. This reduced scheduling costs significantly (in benchmarks, most heap operations required at most $0.5\mu s$) and illustrates how sensitive global algorithms are to the manner in which run queues are supported.

We speculate that the results of this paper may extend to other multicore platforms where rather simple cores are used (as is the case in some embedded platforms). However, performance trends may be very different on platforms with cores of the complexity of x86-like architectures, which have features designed to hide memory latencies. On such a platform, preemption and migration costs may be very different from that seen on our test platform. However, if companies like Intel really do intend to produce chips with 80 cores as noted earlier, it remains to be seen if this can be done without resorting to simpler core designs. Note that the use of simpler core designs will likely result in higher execution costs and hence task utilizations. This does not bode well for P-EDF. On the other hand, we are doubtful that global algorithms can scale to such high core counts. Thus, we expect clustered algorithms to offer the best performance.

5 Conclusion

This paper has presented a case-study comparison of several multiprocessor real-time scheduling algorithms on a Niagara platform. To the best of our knowledge, this study is the first attempt by anyone to assess the *scalability* of such algorithms based upon empirically-derived overheads. The major conclusions of our study are as follows: **(i)** for HRT workloads on the Niagara, S-PD^2 and P-EDF are generally the most effective approaches; **(ii)** for SRT workloads on the Niagara, there is no single best overall algorithm, although S-PD^2 and C-EDF seem to be less susceptible to pathological scenarios than the other tested algorithms; **(iii)** for multicore platforms with on-chip shared caches, preemption and migration costs can still be considerably more costly in algorithms that allow migrations if caches are small or memory bandwidth is limited; **(iv)** quantum-staggering can be very effective in reducing such costs; **(v)** for global approaches, scheduling overheads are greatly impacted by the manner in which run queues are implemented. Although P-EDF fared relatively well in our study, we note that, in other work, we have found that partitioning approaches are more negatively impacted (in terms of schedulability) by task-synchronization requirements [12, 14] and runtime workload changes [9]. Such concerns have not been considered in this paper.

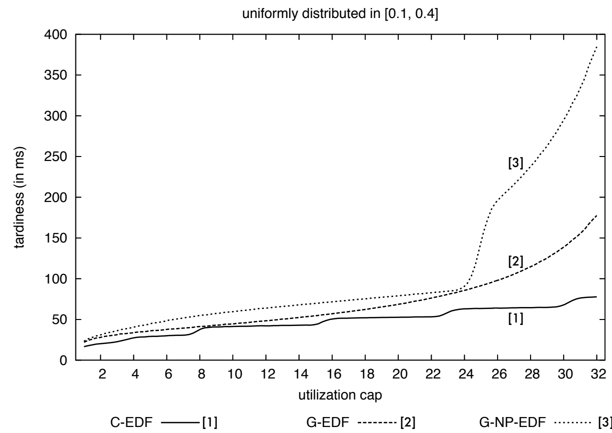

Figure 4: Tardiness as a function of the total utilization cap. This graph corresponds to inset (b) of Fig. 3 and is representative for other results obtained but not shown here due to space constraints.

There are numerous directions for future work. First, we would like to re-consider how run-queue support for global algorithms is provided in LITMUS$^{\mathrm{RT}}$. In particular, we would like to determine whether run-queue overheads can be lessened through the use of more clever data structures or more efficient synchronization techniques. Second, in related work, we have investigated the impact synchronization has on schedulability in work involving a conventional SMP [12, 14]; we would like to repeat that study on the Niagara. Third, we would like to repeat any Niagara-based studies we conduct on an Intel platform of comparable size, when one is available. In all of these studies, we would also like to consider the impact of dynamic workload changes.

Acknowledgement. We thank Sanjoy Baruah for helpful discussions regarding G-EDF HRT schedulability tests.

References

[1] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, February 2004.

[2] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Department of Computer Science, Florida State University, 2005.

[3] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In Sang H. Son, Insup Lee, and Joseph Y. Leung, editors, *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC, Boca Raton, Florida, 2007.

[4] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129, Dec. 2003.

[5] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 119–128, 2007.

[6] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[7] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[8] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2008. To appear.

[9] A. Block and J. Anderson. Accuracy versus migration overhead in multiprocessor reweighting algorithms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364, July 2006.

[10] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, July 2008.

[11] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70, July 2007.

[12] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, M-PCP, D-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 185–194, August 2008.

[13] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Workshop*, pages 107–123. Real-Time Linux Foundation, November 2007.

[14] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, April 2008.

[15] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256, July 2007.

[16] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, December 2006.

[17] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[18] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.

[19] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, February 2008.

[20] C. Farivar. Intel Developers Forum Roundup: Four cores now, 80 cores later. http://www.engadget.com /2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/, 2006.

[21] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[22] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*, September 1998.

[23] P. Holman and J. Anderson. Adapting Pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, 2005.

[24] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[25] P. McKenney. Shrinking slices: Looking at real time for Linux, PowerPC, and Cell. http://www-128.ibm.com/developerworks/power/library/pa-nl14-directions.html, 2005.

[26] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[27] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, September 2006.

[28] UNC Real-Time Group. LITMUS$^{\mathrm{RT}}$ project. http://www.cs.unc.edu/˜anderson/litmus-rt/.

[29] V. Yodaiken and M. Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, January 1997.