Minimizing Response Times of Automotive Dataflows on Multicore

Glenn A. Elliott†, Namhoon Kim†, Jeremy P. Erickson†, Cong Liu†, and James H. Anderson†
†Dept. of Computer Science, University of North Carolina at Chapel Hill
‡Dept. of Computer Science, University of Texas at Dallas

Abstract

Dataflow software architectures are prevalent in prototypes of advanced automotive systems, for both driver-assisted and autonomous driving. Safety constraints of these systems necessitate real-time performance guarantees. Automotive prototypes often ensure such constraints through over-provisioning and dedicated hardware; however, a commercially viable system must utilize as few low-cost multicore processors as possible to meet size, weight, and power constraints. In short, these platforms must do more with less. To this end, we develop cache-aware and overhead-cognizant scheduling techniques that lessen guaranteed response times without unnecessarily constraining platform utilization. We implement these techniques in PGM[12], a portable middleware framework for managing real-time dataflow applications on multicore platforms. The efficacy of our techniques is demonstrated through overhead-aware schedulability experiments and runtime observations. Results for our test platform show that cache-aware clustered scheduling outperforms naïve partitioned and global approaches in terms of schedulability and end-to-end response times of dataflows.

1 Introduction

Graph-based software architectures, often referred to as dataflow architectures, are common to software applications that process continual streams of data or events. In such architectures, vertices represent sequential code segments that operate upon data, and edges express the flow of data among vertices. The flexibility offered by such an architecture’s inherent modularity promotes code reuse and parallel development. Also, these architectures naturally support concurrency, since parallelism can be explicitly described by the graph structure. These characteristics have made dataflow architectures popular in multimedia technologies [13, 24] and the emerging field of computational photography [2, 27]. Dataflow architectures are also prevalent in the sensor-processing components in prototypes of advanced automotive systems, for both driver-assisted and autonomous driving (e.g., [20, 25, 26]). While many domains with dataflow architectures have timing requirements, the automotive case is set apart since timing violations may result in loss of life or property.

In addition to timing requirements, an automotive platform has size, weight, and power (SWaP) and manufacturing cost constraints. Such constraints are largely ignored in many existing prototypes of advanced automotive systems, which typically rely on over-provisioning and dedicated hardware to support real-time dataflow computations [20, 25, 26]. If advanced automotive features are to be commercially viable, then such computations must instead be consolidated onto as few low-cost processors as possible. However, consolidation only bolsters the need for efficient real-time scheduling techniques since more computations compete for fewer available processors.

Real-time dataflows. A number of methods for modeling the real-time behavior of dataflow applications have been developed for multiprocessor and distributed systems [4, 12, 15, 23]. Typically, such applications are modeled as directed acyclic graphs (DAGs), with nodes denoting tasks and edges denoting producer/consumer relationships. The predominant approach has been to map such a DAG onto a set of connected, but independently scheduled, processors. Under this partitioned approach, each task (DAG node) is statically assigned to a single processor. As with traditional partitioned multiprocessor scheduling, this method may suffer from utilization loss due to bin-packing problems.

More recently, Liu and Anderson explored dataflow applications on globally scheduled multiprocessors, wherein DAG tasks share a single run queue, and thus may migrate among processors [18]. Liu et al. showed that task response times are bounded under global earliest-deadline-first (G-EDF) scheduling, without utilization loss. Liu et al. extended these results to apply in distributed systems comprised of globally scheduled multicore machines [19]; these results can also be applied to a single cluster-scheduled multicore system (where groups of processors share a run queue). However, solutions optimized for shared-memory communication have not been investigated.

A similar DAG-based model has been used to analyze the hard real-time scheduling under G-EDF of tasks with parallelism within jobs [14, 17, 21, 22].

Sporadic task systems and minimizing response-time bounds. Devi et al. were the first to demonstrate that deadline-tardiness bounds (i.e., response-time bounds) can be derived for ordinary sporadic tasks scheduled by G-EDF, without utilization loss [6]. However, Bastoni et al. showed that algorithms like G-EDF may incur high runtime overheads and suffer from heavy cache-affinity loss [3]. As a result of these factors, real-time schedulability is reduced and the response-time bounds of schedulable task sets increase. Clustered schedulers, such as clustered EDF (C-EDF), can balance utilization loss against high overheads, striking a middle ground between partitioned and global scheduling.

Recently, Erickson et al. showed that “G-EDF-like” (GEL) schedulers, such as the global fair-lateness (G-FL)
scheduling, have provably smaller response-time bounds than G-EDF [9]. Under GEL schedulers, priority points (PPs) distinct from deadlines are defined to minimize worst-case response times. A GEL scheduler uses these PPs as pseudo-deadlines. Erickson et al. also showed that job splitting may further minimize bounds on response times [8]. Here, the scheduling priority of a job decreases as the job consumes discrete chunks of execution budget, effectively splitting each “real” job into a sequence of sub-jobs. Clustered variants of GEL schedulers can also be created by using the GEL scheduler for each cluster of processors.

Contributions. Future advanced automotive systems need computing platforms that support efficient real-time scheduling of dataflow applications on highly-utilized multicore processors. With this firmly in mind, we consider the problem of scheduling sporadic DAGs on multicore platforms with the goals of: (i) minimizing bounds on end-to-end latency; (ii) maximizing system utilization; (iii) improving analytical schedulability and observed performance; and (iv) providing a real-time software platform from which we may draw practical conclusions and facilitate future research. We meet (i) and (ii) by extending Liu et al.’s dataflow analysis and integrating it with Erickson et al.’s techniques to lessen DAG end-to-end response-time bounds. We avoid the high overheads of global scheduling through the use of the clustered fair-lateness (C-FL) scheduler. We also use job splitting to further reduce response times. We address (iii) through a cache-aware heuristic to assign tasks to clusters that promotes cache reuse and reduces communication costs. To meet (iv), we created portable real-time dataflow scheduling middleware called PGM\textsuperscript{RT}, which we run atop LITMUS\textsuperscript{RT}, a Linux-based real-time operating system jointly developed by UNC and MPI.\textsuperscript{1} Using this implementation, we gathered overhead measurements, which we then integrated into overhead-aware schedulability experiments. We also conducted experiments in which observed end-to-end DAG response times were recorded under various configurations of PGM\textsuperscript{RT} to determine which configurations offer the best real-time behaviors while maximizing system utilization.

Organization. In the rest of the paper, we first provide needed background and review relevant prior work (Secs. 2–4). We then present a heuristic for mapping DAG tasks to clusters to promote efficient cache reuse (Sec. 5), discuss the implementation of PGM\textsuperscript{RT} (Sec. 6), present our experimental results (Sec. 7), and conclude (Sec. 8).

2 Background

We begin with additional motivation behind the graph-based scheduling of automotive applications. Fig. 1 depicts a hypothetical vehicle and pedestrian detection and tracking component of an advanced automotive system. A video camera feeds the source of the graph with video frames at 30Hz. $T_1^1$ converts the raw camera data into the common YUV color image format. $T_1^1$ hands the image to $T_1^2$ for display. It also passes the data to $T_2^2$ for computer vision processing. Vision algorithms often operate only on grayscale images, so $T_2^2$ strips the color components (U and V) from the image to produce a more compact grayscale image. $T_3^2$ then sends the grayscale image to $T_3^3$ and $T_4^4$ for vehicle and pedestrian detection, respectively. Although vehicle and pedestrian detectors may use similar algorithms, $T_3^3$ requires less execution time because it limits its search to the immediate road area before the vehicle. Information on detected vehicles and pedestrians is passed to $T_5^5$ and $T_6^6$, respectively, for tracking. Here, detected objects are correlated with historical data to produce an estimated trajectory and speed for each object. This data is sent to $T_7^7$, where the relevant information is overlaid on top of the original color image.

In this paper, we explore the use of C-FL scheduling in safety-critical applications like the one above. This may seem like an odd choice since EDF-like global and clustered schedulers, like C-FL, are commonly associated with “soft real-time” systems (i.e., those generally regarded as non-safety-critical) because in the absence of severe utilization constraints, analysis “only” provides bounds on deadline tardiness. However, C-FL is a viable choice in this domain for several reasons. First, C-FL still provides provable bounds on end-to-end latency for DAGs—timing properties can be guaranteed. Second, a survey of automotive studies on driver reactions places an alert driver’s reaction time around 700ms [11]. Some automotive software components may only have to react to events within this relatively lax time window in order to ensure safe operation. Finally, C-FL enables a high degree of system utilization. This is important in light of SWaP concerns. This last reason motivates our departure from partitioned fixed-priority scheduling, as prescribed by the AUTOSAR automotive standard [1], which suffers from well-known utilization constraints. We hope this work may be informative to future automotive standards.

3 System Model

In this section, we present the implicit-deadline sporadic DAG task model, which is our focus herein. Such a task

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Vehicle and pedestrian detection and tracking component, expressed as a DAG. $e_j^i$ denotes the execution time of task $T_j^i$.}
\end{figure}

\footnote{\textsuperscript{1}All source code is available at www.litmus-rt.org.}
system is comprised of a set \( \tau = \{T_1, T_2, \ldots, T_n\} \) of \( n \) DAGs. Each DAG is a set \( T_i = \{T_i^1, T_i^2, \ldots, T_i^z\} \) of \( z_i \) tasks, with producer/consumer relationships. Each task releases a (potentially infinite) series of jobs \( T_i^{j,1}, T_i^{j,2}, \ldots \). An example DAG \( T_1 \) is depicted in Fig. 2. Each edge is directed from a producer task that produces data to a consumer task that consumes that data. A particular task \( T_i^j \)'s producers, \( prod(T_i^j) \), are those on edges for which \( T_i^j \) is the consumer, and its consumers, \( cons(T_i^j) \), are those on edges for which \( T_i^j \) is the producer. The maximum number of bytes that a job of \( T_i^j \) can produce for \( T_i^k \) to consume is denoted as \( b_i^{j \to k} \). Each job must wait to begin execution until one job from each of its producers has completed, so that the necessary input data is available. For example, in Fig. 2, for any \( k \), \( T_i^{1,k} \) needs input data from each of \( T_i^{2,k} \) and \( T_i^{3,k} \), so it must wait until those jobs complete.

To simplify analysis, each DAG \( T_i \) has exactly one source task \( T_i^1 \), which only has outgoing edges, and one sink task \( T_i^z \), which has only incoming edges. Multi-source-sink DAGs may be supported with the addition of singular “virtual” sources and sinks that connect multiple sources and sinks, respectively. The depth of a task \( T_i^j \) is the number of edges on the longest path between \( T_i^j \) and \( T_i^1 \), and the height of a DAG is the depth of its sink task. For example, the depth of \( T_i^4 \) in Fig. 2 is two, and \( T_i^4 \) is the sink task so the depth of \( T_i \) is two. Also, each DAG has a common minimum separation time \( p_i \) for all of its tasks. Each job of any task in \( T_i \) has a deadline \( p_i \) units of time after it is released. Each task \( T_i^j \) also has a parameter \( c_i^j \), which denotes the worst-case execution time (WCET) for any of its jobs. This parameter does not include the cost of consuming data from \( prod(T_i^j) \)—we account for this later in analysis. We assume that \( \tau \) is scheduled on an identical multiprocessor.

Prior work [10, 18, 19] considered the more general processing graph method (PGM) to describe DAGs, where DAG nodes may have different priorities. The implicit-deadline sporadic DAG model above is a restricted case of PGM DAGs. Our restricted model is sufficient to schedule the automotive workloads we consider. Further, the end-to-end latency bounds we show in Sec. 4 are tighter than would otherwise be possible using the more generalized analysis [18, 19]. However, the techniques herein can be applied to DAGs with nodes of different periods, so we present end-to-end latency bounds for such DAGs in [7].

4 Determining End-to-End Latency

We now discuss how to apply results from [19] on PGM-based systems to analyze the end-to-end response time (or latency) of a DAG \( T_i \). In [19], job release times and deadlines are computed on-the-fly such that tasks \( T_i^j \in T_i \) behave as independent sporadic tasks. Here, we present the ideas behind this technique and describe how it can be applied in our context.

The analysis from [19] is general enough to apply to clustered algorithms wherein an intra-cluster scheduler is used from a broad class of window-constrained [16] global scheduling algorithms. Under a window-constrained global algorithm, all jobs share a single run queue, and each job is prioritized on the basis of a priority point (PP) that can move during execution, but that must remain within a constant distance of the job’s deadline. The released but unfinished jobs with the earliest PP are scheduled for execution. The most well-known window-constrained algorithm is G-EDF, which uses the deadline of each job as its PP. However, Erickson et al. [9] demonstrated that allowing PPs set earlier than deadlines can result in better response-time bounds. The G-FL scheduler, mentioned in Sec. 1, functions in this way. Erickson et al. [8] also showed that response time bounds can be further improved by simulating a G-FL schedule of a task system with more frequent releases of smaller jobs. This method is referred to as job splitting. Unlike G-EDF and basic G-FL, G-FL with job splitting moves the PP of a job during execution. However, it does so in a way that maintains the window-constrained property.

In a DAG \( T_i \), the source task \( T_i^1 \), which is assumed to follow the implicit-deadline sporadic task model: its releases are separated by at least \( p_i \) time units, and each job’s deadline is \( p_i \) time units after its release. We define for each job an ideal release and an ideal deadline that are identical to the release time and deadline of the corresponding job from the source task. We also define for each job \( T_i^{j,k} \), an actual release that is sometimes later than its ideal release, and a corresponding actual deadline \( p_i \) units later. The source task is a special case in that its actual and ideal releases are identical, and thus its actual and ideal deadlines are identical. Scheduler decisions are always based on PPs derived from actual deadlines rather than ideal ones.

The primary purpose of delaying the actual release is to model the fact that a job cannot run before its producers have completed. Because G-EDF is the simplest window-constrained algorithm, we show in Fig. 3(a) the G-EDF scheduling of a DAG with the same structure as the DAG from Fig. 2 (ignoring overheads for simplicity). Observe that the actual releases of \( T_i^{2,1} \) and \( T_i^{3,1} \) are at time 6, when \( T_i^{1,1} \) completes. Actual releases and deadlines must follow the sporadic task model. For example, consider time 33 in Fig. 3(a), when \( T_i^{3,3} \) finishes early. Because all producers of \( T_i^{3,3} \) are now complete, \( T_i^{4,3} \) commences execution at time 33. However, because \( T_i^{3,3} \)'s last actual release was at time 24, the actual release of \( T_i^{4,3} \) is not until time 34, and its actual deadline is at time 44. This is to ensure that the deadline used by the scheduler follows the sporadic task model. Prior response-time analysis [9] remains correct when jobs may
CPU 1  CPU 2  Job completion  Priority point (PP)

Actual release  Actual deadline  Ideal release  Ideal deadline

Figure 3: Example schedules of the DAG depicted in Fig. 2 with 

execute before their actual release times, as long as their ac-
tual deadlines follow the sporadic task model. Thus, if a job’s
ideal release precedes its actual release, and all of its prece-
dence constraints are satisfied, then it may commence exe-
cution if it has sufficient priority, as at time 33 in Fig. 3(b).

If the tasks in τ can be assigned onto clusters of CPUs such
that no individual cluster is overutilized, then be-
cause actual deadlines follow the sporadic task model, any
scheduler using a window-constrained scheduling algorithm
within each cluster can ensure that no job T_{i}^{j,k}
completes more than some per-task constant R_{i}^{j}
units (i.e., the response time bound) after its actual release, as shown in [16].

Liu et al. [19] demonstrated that this property can be
used to provide a bound relative to each job’s ideal release,
by showing that actual and ideal releases only differ by a
bounded amount. The following theorem is similar to Thm. 3 in [19], but is much less pessimistic. (The improvements are
described in an online appendix [7].)

Theorem 1. Suppose that the tasks in τ can be assigned onto
clusters of CPUs such that no individual cluster is overuti-
лизed. If Θ is the set of all tasks along the worst-case path2
from T_{j}^{z} to T_{i}^{j}, including both T_{i}^{z} and T_{j}^{j}, then any job T_{i}^{j,k}
completes within


units of its ideal release.

We consider the end-to-end latency for an entire graph,
rather than for a single job. However, by our definition of
ideal releases, T_{i}^{z} has the same ideal release as T_{j}^{z}. There-

2That is, the path that maximizes the given sum.

Figure 4: Example memory hierarchy. L2 clustering depicted.

fore, the latency of the graph is simply the bound provided
by Thm. 1 with j = z.

Although the analysis in [19] was shown to be cor-
rect whenever a window-constrained scheduling algorithm
is used within each cluster, it was only experimentally eval-
uated using C-EDF (and it was not evaluated based on a real
scheduler implementation). As mentioned above, C-FL can
provide superior response-time bounds by using PPs that are
earlier than deadlines. C-FL biases scheduling priority to-
wards tasks with larger e_i values. Recall that, because there
is only one cluster in the example we are considering, G-FL
is identical to C-FL. An example of the operation of G-FL is
depicted in Fig. 3(b). Observe that, in comparison with the
G-EDF schedule in Fig. 3(a), T_{1}^{1,2} finishes at time 28 instead
of time 30, T_{4}^{1,3} finishes at time 37 instead of time 39, and
so on. Thus, using G-FL or C-FL can reduce the end-to-end
latency of a graph, as the theoretical bounds in [9] indicate.
Although not pictured due to space constraints, job splitting
can further reduce the latency of each graph. In this paper,
we explore the use of both techniques.

5 Assigning Tasks to Clusters

We now examine the problem of assigning DAG tasks to pro-
cessor clusters, and present an assignment heuristic designed
to promote efficient cache sharing among DAG tasks. We
also develop a method for determining overheads pertaining
to the data passed between tasks.

In a shared-memory system, node output data is written
directly to memory—there is no explicit message passing of
data among nodes. Data may reside anywhere within the
system’s memory hierarchy, from the private L1 cache of
a processor to main memory. The cost of reading this data
depends upon (i) the location of data within the memory hi-
erarchy, and (ii) the processor from which the read is made.
Reads become more costly as the distance between the data
and the reader increases. For example, the cost of reading data
from a processor’s own L1 cache is cheap, while reading
data from an L3 cache is more expensive.

Consider the memory hierarchy of a dual-socket twelve-
core system depicted in Fig. 4. Here, there are private per-CPU 192KB L1s for data (L1-D) and instructions (L1-I),
pair-wise shared 3MB L2s, per-socket 12MB L3s, and fi-
ally, main memory. For simplicity, let us assume the caches
are coherent, inclusive, fully associative, and follow a least-

recently-used (LRU) eviction policy. Suppose processors that share a common L2 are cluster-scheduled, as depicted in Fig. 4, and we wish to map the DAG depicted in Fig. 2 to this system. Further suppose that \( T^1 \) produces 128KB and 1MB of data for \( T^2 \) and \( T^3 \), respectively, and \( T^2 \) and \( T^3 \) produce 128KB and 1MB of data respectively for \( T^4 \).\footnote{We address the practicality of these assumptions later in this paper.} Worst-fit decreasing is a commonly studied partitioning heuristic, in which tasks are allocated to clusters in order of decreasing utilization, and each task is allocated to the cluster with the largest available capacity. Under such a heuristic, each task would be put in a separate cluster: \( T^1 \) in \( C_1 \), \( T^2 \) in \( C_2 \), \( T^3 \) in \( C_3 \), and \( T^4 \) in \( C_4 \). With this assignment, the read costs of \( T^1 \) and \( T^3 \) can be no less than reads from the L3 shared by \( C_1 \), \( C_2 \), and \( C_3 \), as this is their closest common cache. Costs are greater for \( T^1 \) since \( C_4 \) has no common cache with \( C_2 \) or \( C_3 \). A far more cache-efficient strategy is to put all tasks in \( C_1 \), since all the written data (2.25MB) can fit within \( C_1 \)’s 3MB L2 cache. Thus, the end-to-end latency of this graph may be less since each task executes for a shorter duration.

This illustrates that a cache-efficient partitioning heuristic for mapping nodes to clusters should be cognizant of the system’s memory hierarchy and associated costs. Ultimately, this heuristic should place related nodes in the same or closest adjacent cluster, as long as schedulability is maintained. This might not actually enhance schedulability if conservative assumptions are made with respect to caches in timing analysis, but runtime performance may still be positively impacted, which increases safety margins.

### 5.1 Cache-Aware Task Assignment

In this section, we present a cache-aware task assignment heuristic, which is shown in Fig. 5. This heuristic was designed assuming the system of \( m \) processors consists of \( m/c \) clusters, each containing \( c \) processors. The heuristic reverts to the worst-fit decreasing heuristic after all clusters have total utilization exceeding an aggressiveness factor \( h \). Since system overheads are charged later in schedulability analysis, the aggressiveness factor prevents loading a cluster to the degree that even small overheads lead to overutilization while other clusters remain underutilized.

We keep track of the following quantities. \( T \) is the set of unassigned tasks from all DAGs, initially \( T = \cup_{T_i \in T} T_i \). \( C = \{C_0, C_1, \ldots, C_{m/c}\} \) is the set of all clusters. \( a(T_i) \) for each \( T_i \) is the cluster to which \( T_i \) is assigned, initially \( \text{Nil} \), indicating that it has not been assigned to a cluster.

We also utilize several functions. \( \text{cost}(T_i) \) is the cost of reading all data produced for \( T_i \) by its producers, according to the cluster assignments of each \( T_j \in \text{prod}(T_i) \) and \( T_i \). If some \( a(T_i) \) or \( a(T_j) \) is \( \text{Nil} \), then the tasks are assumed to be partitioned the farthest apart with respect to the memory hierarchy. \( w(T_i) \triangleq (c^i + \text{cost}(T_i))/p_i \) is the approximated utilization of \( T_i \) after accounting for communication overheads. \( u(C_k) \triangleq \sum_{T_j \in C_k} w(T_j) \) is the total utilization of a cluster, including communication overheads. Finally, \( \text{min}(C) \) is the set of clusters with minimum \( u(C_k) \).

**Algorithm description.** We first describe \textsc{PickCluster}, which is used to select the best cluster for a particular task \( T_i \). It accepts a parameter \( Q \), a list of candidate clusters. In line 1, it first attempts to select the cluster that will result in the smallest value of \( w(T_i) \). If there is a tie, then line 2 will continue to select the cluster that minimizes the cost of any consumers that are already assigned. If there is still a tie, then line 3 will select the cluster that places \( T_i \) closest to other tasks in \( T_i \), even if those tasks are neither producers nor consumers of \( T_i \). Finally, if there is still a tie, the cluster with minimum utilization is selected in line 4.

We now describe \textsc{CacheAwarePartition}, which performs the actual partitioning. The first part of our heuristic is based on some of the details from the response-time analysis in [9]. That analysis considers the number of tasks that may execute while a single processor is left idle. Ultimately, the \( c - 1 \) tasks of greatest utilization within a cluster have the most significant impact on response-time bounds. A cluster with a disproportionate number of high-utilization tasks may result in larger response-time bounds in comparison to clusters with lighter-utilization tasks, even if these clusters are equally loaded. We want to avoid this problem. Thus, we spread the first \((m/c) \cdot (c - 1) = m - m/c \) tasks as evenly as possible across clusters by using the worst-fit decreasing heuristic. This is done in lines 1–3. Ties are broken using \textsc{PickCluster}. For each remaining task (see line 4), we select in line 5 the set of all clusters for which adding \( T_i \) will not cause the cluster’s utilization to exceed the aggressiveness factor \( h \). If there are such clusters, then line 7 selects the best cluster using \textsc{PickCluster}. If not, then the algorithm reverts to a worst-fit heuristic (line 9), using \textsc{PickCluster} to break ties.

**5.2 Assessing Data Passing Costs**

The performance of \textsc{CacheAwarePartition} hinges on the function \( \text{cost}(T_i) \) used to estimate the cost of passing data between tasks. We now describe a method for estimating \( \text{cost}(T_i) \) that assesses the costs associated with passing data between tasks on a multicore platform with a complex
memory hierarchy.

The value of \( \text{cost}(T^j_k) \) is obtained from careful measurements of cache behavior. We use an experimental method modeled after the “synthetic method” described in [5] that is used to assess cache preemption and migration delays. A non-preemptive instrumented process records the time taken to read a prescribed amount of sequential data from a “hot” cache. The process suspends for a short duration, resumes on a random processor, and rereads data from the now “cold” cache. A cost is determined by subtracting the hot measurement from the cold. We classify this measurement according to the closest shared cache between the read-issuing processors. Measurements are classified as “memory” if there is no shared cache between them. After many thousands of measurements, we derive a lookup table indexed by closest common cache and data size. We collect two datasets using this method: (i) an “idle” dataset where the instrumented process runs alone, and (ii) a “polluter” dataset where “cold” measurements are taken in the presence of cache-trashing processes that introduce contention for both caches and memory bus.

These two datasets actually yield two variants of the \( \text{cost}(T^j_k) \) function that provide lower and upper bounds of actual communication overhead. The lower bounds we obtain are based upon certain ideal assumptions (see below) regarding cache behavior. This is acceptable, because our main intent is to produce task assignments that improve run-time performance and hence increase safety margins, rather than to determine precise overhead costs. Indeed, rigorous analysis of cache behavior for the purpose of precise timing analysis is still an unresolved issue in the context of multicore platforms. In the experiments presented in Sec. 7, we present schedulability results for the two extremes of “idle” and “polluter” overheads. If adequate timing analysis tools were available for multicore platforms, they would likely yield schedulability results between these two extremes.

The \( \text{cost}(T^j_k) \) function cannot be derived by simply summing individual costs between each \( T^j_k \) and \( T^i_l \) due to cache sharing. We illustrate this with an example.

**Example.** Consider the memory hierarchy from Fig. 4. For simplicity, assume that caches are inclusive, fully associative, use an LRU eviction policy, and that no cache interference occurs due to instruction caching or operations from other processors. Suppose we wish to compute \( \text{cost}(T^3_1) \) for the subgraph depicted in Fig. 6. Here, an L1-D cache is shared between \( T^2_1 \) and \( T^3_2 \). Likewise, an L3 is shared between \( T^1_1 \) and \( T^3_3 \). We will now walk through a worst-case sequence of operations that maximizes \( \text{cost}(T^3_1) \).

**Step 1:** \( T^2_1 \) writes 1024KB of data for \( T^3_1 \); the L1-D is only 192KB in size, so 192KB of this data is stored in the L1-D, while the remaining 832KB spills to the L2.

**Step 2:** \( T^1_1 \) writes 512KB of data for \( T^3_1 \). Although \( T^1_1 \) executes on a remote processor, a copy of this data resides in the L3 due to inclusivity.

**Step 3:** \( T^2_3 \) writes 2560KB of data for \( T^3_1 \). This causes evictions on two levels. First, all of \( T^3_1 \)'s data in the L1-D spills to the L2 after the first 192KB of \( T^3_1 \)'s data is written. The L1-D continues to spill as remaining data is written. Ultimately, 2368KB of \( T^3_1 \)'s data spills to the L2. The L2 must maintain a copy of the L1 (both instruction and data caches) due to inclusivity, so the L2 can only store 2688KB of spilled data. \( T^3_1 \)'s data from Step 1 is evicted first according to the LRU policy, so 768KB of \( T^3_1 \)'s data is spilled from the L2 to the L3, and 256KB of it remains in the L2.

**Step 4:** \( T^3_1 \) reads the 512KB of data from \( T^3_1 \). This L3-read triggers 192KB of data from Step 3 to spill from the L1-D to the L2, which in turn triggers 192KB of \( T^3_1 \)'s data from Step 1 to spill from the L2 to the L3. \( T^3_1 \)'s remaining 64KB of data in the L2 is self-evicted as \( T^3_1 \) reads the rest of the data from \( T^1_1 \).

**Step 5:** \( T^3_1 \) reads the 1024KB of data from \( T^2_1 \). All 1024KB of data from Step 1 resides in the L3 due to spills. All of \( T^3_1 \)'s data is read from the L3, so we charge \( T^3_1 \) for 1536KB of data read from the L3, according to the overhead dataset.

Our general algorithm to compute \( \text{cost}(T^j_k) \) is outlined as follows. First, we assume all \( T^i_l \in \prod(T^j_k) \) collectively write data for \( T^i_l \) prior to data for any \( T^k_j \in \cons(T^j_k) \), and we model evictions to track the location of \( T^j_k \)'s data. Next, we assume that \( T^j_k \) reads the most distant data first, and we continue to track the location of \( T^j_k \)'s remaining data by modeling self-evictions. We sum overhead costs according to the location of data as it is read by \( T^j_k \).

**A remark on optimism.** We have made optimistic assumptions in computing \( \text{cost}(T^j_k) \): full associativity, LRU, no instruction caching, etc. Since \( \text{cost}(T^j_k) \) conservatively estimates best-case conditions in the “idle” case, it can provide a lower bound on overhead costs. However, this optimism is of no consequence under our “polluter” dataset, where all costs are equivalent: every read of a new cache line results in a cache miss at all levels. Thus, \( \text{cost}(T^j_k) \) can provide both an upper and lower bound on overheads, according to the dataset. Deeply involved timing analysis techniques are necessary to provide tighter bounds.

## 6 Implementation

PGMRT is a portable lightweight middleware layer we developed to manage the coordinated execution of dataflow applications. We describe the subset of PGMRT necessary to support the automotive workloads that we have discussed. However, PGMRT supports the full PGM specification; please see the online version of this paper for a full description of PGMRT’s API and capabilities [7].

**Graphs, nodes, and edges.** Each graph is identified by a unique name and path, similar to a UNIX named pipe. Applications use PGMRT’s API to create new graphs described by nodes and edges. Real-time tasks, as unique threads of execution within the same address space or separate pro-
cesses, use PGM\textsuperscript{RT}'s API to access information about a named graph and claim/bind to a node and its edges.

**Precedence constraints.** As described in Sec. 4, non-source nodes have two types of precedence constraints: job and producer constraints. Job constraints are satisfied in PGM\textsuperscript{RT} since a single thread binds to each node—jobs are naturally serialized by this thread.

Predecessor constraints are tracked by *tokens*. A producer generates one token on each outbound edge upon job completion. A consumer’s producer constraints are satisfied when there is at least one token on each of its incoming edges. The consumer blocks (suspends execution) whenever the requisite tokens are unavailable. When the needed tokens are available, the consumer consumes one token from each of its incoming edges and the task is ready to execute.

Token production/consumption is realized through increment/decrement operations on per-edge token counters, similar to counting semaphores. Although these tokens do not transmit data explicitly, tokens can coordinate data sharing in application-level logic. Since consumers may suspend execution, producers must have a mechanism to signal consumers of new tokens. This is achieved using a monitor synchronization primitive, specifically, a POSIX (pthread) condition variable, one per consumer.\footnote{On LITMUS\textsuperscript{RT}, PGM\textsuperscript{RT} uses a low-overhead implementation of a monitor that combines spinlocks with Linux’s “fast userspace mutex” interface and LITMUS\textsuperscript{RT}’s special support for non-preemptive code sections.} A consumer blocks on its condition variable if it does not have its requisite tokens. A producer signals the condition variable whenever it is the last producer to satisfy all of its consumer’s producer constraints.

**Real-time concerns.** PGM\textsuperscript{RT} described as above can be used with general-purpose schedulers. However, additional enhancements are required to ensure deterministic real-time behavior. These relate to deterministic token signaling and proper deadline assignment.

The preemption of a consumer while it is signaling a sequence of consumers may leave processors in remote clusters idle. To avoid this, producers are non-preemptive during the signaling process. In the LITMUS\textsuperscript{RT}-based version of PGM\textsuperscript{RT} used in the experiments in Sec. 7, we use LITMUS\textsuperscript{RT}’s capabilities to quickly enter and exit these non-preemptive code sections. For a portable alternative, interrupts may be disabled from userspace (e.g., sti/cli instructions on x86 processors). These short durations of non-preemption must be accounted for in real-time analysis, since they may momentarily delay higher priority work from being scheduled.

Recall from Sec. 4 that a job’s ideal (actual) deadline is computed as \( p_t \) time units after its ideal (actual) release. The ideal release time of a job can be tracked with a high-resolution timer. However, the actual release time depends upon token arrival, so the actual deadline must be computed on-the-fly. In our LITMUS\textsuperscript{RT}-based implementation, immediately before a consumer blocks for tokens, it sets a “token-wait” flag in memory shared by userspace and the kernel. The kernel checks this flag whenever a real-time task is awoken from a sleeping state. If set, and the current time is later than the ideal release time, the kernel automatically computes the actual release and deadline for the job and clears the flag. Otherwise, the ideal and actual deadlines coincide. Observe that this computation requires the current time to approximate the arrival time of the last token—this is ensured by the producer’s non-preemption discussed earlier. For a portable approach, producers may set the deadline of a consumer directly prior to signaling, but this comes at the cost of system call overheads and requires support from the operating system.

### 7 Experimental Results

In this section, we evaluate PGM\textsuperscript{RT} via an overhead-aware schedulability study and a case-study of observed end-to-end latencies.

#### 7.1 Schedulability Experiments

We evaluate C-FL cluster scheduling, job splitting, and our cache-aware cluster assignment techniques through overhead-aware schedulability experiments. We randomly generated DAGs of varying characteristics and tested them for schedulability using the methods described in [8]. We now describe the experimental process we used.

**Overheads.** Our implementation of PGM\textsuperscript{RT} on LITMUS\textsuperscript{RT} was run on a four-socket (24-core) Intel Xeon L7455 running at 2.13GHz. Each socket has the same memory hierarchy as that in Fig. 4. Every core has a private 8-way set-associative 192KB L1-data and L1-instruction cache. Core pairs share a 12-way set-associative 3MB L2 cache. Finally, the cores on each socket share a 12-way set-associative 12MB L3 cache. We measured worst-case runtime overheads and preemption/migration delays in the manner described in [5]. We also measured data-passing overheads, as described in Sec. 5.

We made several changes to the overhead-accounting techniques for C-FL as described in [8] to support DAGs under PGM\textsuperscript{RT}. First, we account for non-preemptivity during consumer signaling, as this can delay higher priority work from being scheduled. Second, we account for the system calls made by producers to wake consumers. Finally, we only charge for release-timer overheads of graph source tasks; non-source nodes do not use release timers.

**Experimental setup.** Random DAGs for schedulability experiments were generated according to several parameters in a multistep process. Task utilizations were generated using three uniform and three bimodal distributions. The ranges of the uniform distributions were [0.001, 0.1] (*light*), [0.1, 0.4] (*medium*), and [0.5, 0.9] (*heavy*). The bimodal distributions randomly selected from two uniform distributions with ranges [0.001, 0.5] and [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (*bimo-light*), 6/9 and 3/9 (*bimo-medium*), and 4/8 and 5/9 (*bimo-heavy*). Task periods were generating using three uniform distributions with ranges [3ms, 33ms] (*uni-short*), [10ms, 100ms] (*uni-moderate*), [50ms, 250ms] (*uni-long*). The number of DAGs in a task set were selected uniformly in the range [1,12]. The height of a DAG was determined by a “height-factor.” The height-factor was generated using four uniform distributions, with ranges of
[1/3, 1/2] (short-height), [1/2,3/4] (medium-height), [3/4, 1] (tall-height) [1,1] (pipeline). DAG height is computed as \( \text{height-factor} \cdot \# \text{of DAG nodes} \). A uniform distribution of range [1,3] controlled the number of consumers for each non-sink task. The amount of data passed from producer to consumer, or “edge working set size” (EWSS), was controlled by three uniform and three bimodal distributions. The uniform distributions were [1KB, 64KB] (light-weight), [256KB, 1024KB] (medium-weight), and [2MB, 8MB] (heavy-weight). The bimodal distributions randomly selected from two uniform distributions with ranges [64KB, 256KB] and [2MB, 8MB] with respective probabilities of 8/9 and 1/9 (bimo-light-weight), 6/9 and 3/9 (bimo-medium-weight), and 4/8 and 5/9 (bimo-heavy-weight). Tasks were generated by selecting a utilization and period. Tasks were added to a task set until a specified system utilization was reached. After selecting the number of DAGs to generate, tasks were assigned to a random DAG, with a constraint ensuring that each DAG had at least one node. A height-factor was then selected for each DAG and its height determined. After selecting a random source and sink for each DAG, a random number of consumers was selected for each non-sink task. We then selected the amount of data transmitted on each edge. The “working set size” of each task (used to compute preemption/migration overheads) was computed as the sum of a task’s in-bound EWSS. Finally, the period of each non-source task was adjusted to match that of its source node—we scaled execution time to maintain utilization.

A unique combination of the above distributions defined a set of experiment settings. For each 0.1 increment in the system utilization range (0,24], we generated 1,000 task sets. Schedulability under C-FL for each task set was checked under a combination of the following conditions: (i) according to cluster configuration, where the L1s, L2s, L3s, and main memory defined clusters of size one, two, six, and 24, respectively; (ii) task cluster assignment heuristics of worst-fit (WF) in decreasing task utilization and our cache-aware (CA) method (with an aggressiveness factor 75% of cluster capacity); and (iii) cache overheads under idle and polluter conditions. We used a university compute cluster to test the schedulability of over 800 billion task sets.

For each schedulable task set, we computed the end-to-end latency of each DAG according to Eq. (1) from Sec. 4, with \( T_i^j = T_i^{j_1} \), when scheduled with job splitting. The extent of splitting was determined from overhead data (which limits the extent of splitting) using a procedure from [8]. A proportional end-to-end latency metric was computed by dividing the computed the end-to-end latency by \( p_i \cdot (\text{height}(T_i) + 1) \).

**Results.** We present a selection of results from our experiments that illustrate the general trends observed across all data. All schedulability results are available in [7]. Fig. 7 shows results for pipelined shaped DAGs with “uni-long” periods, “uni-medium” task utilizations, and “medium-weight” edges. The top (bottom) of Fig. 7(a) and Fig. 7(b) depict schedulability (proportional end-to-end latency under job splitting) results under idle and cache-polluter overheads, respectively.

**Obs. 1.** Cache-aware assignment improves schedulability.

In the top inset of Fig. 7(a), compare CA and WF cluster assignment for L1 clustering (lines 1 and 4, respectively). Observe that under CA:L1, 50% of task sets with a utilization of about 22.0 were schedulable. In comparison, under WF:L1, only 50% of task sets with a utilization of about 18.5 were schedulable. Moreover, no task sets with a utilization of 22.0 were schedule under WF:L1. CA cluster assignment outperforms WF cluster assignment for L2 and L3 clustering as well, although the gap in performance is smaller. For example, CA:L3 (line 3) offers only slightly better schedulability than WF:L3 (line 6).

As expected, schedulability across all methods under cache-polluter overheads is worse, as depicted in the top inset of Fig. 7(b). However, at no point does WF cluster assignment outperform CA cluster assignment for the same cluster configuration. This may come as a surprise since no parti-
tioning scheme can reduce the cache-related overheads incurred under the cache-polluter dataset (see end of Sec. 5). However, the CA heuristic also assigns tasks according to an approximated utilization (\( w(T_i) \)) that includes cache-related costs. Thus, the CA heuristic is able to make better task assignments, even if it does not reduce incurred overheads.

**Obs. 2.** Cache-aware cluster assignment is resistant to bin-packing utilization loss.

Cluster configuration strongly affects schedulability under WF cluster assignment. This is observed by comparing WF:L1 (line 4), WF:L2 (line 5), and WF:L3 (line 6) within the top insets of Fig. 7. WF:L3 outperforms WF:L2 by a wide margin, and WF:L2 similarly outperforms WF:L1. In contrast, compare CA:L1 (line 1), CA:L2 (line 2), and CA:L3 (line 3) within the top insets of Fig. 7. The differences in schedulability are much less.

**Obs. 3.** Computed proportional end-to-end latency is less for smaller clusters.

L1 clustering gave the smallest overall proportional latencies, as shown by lines 1 and 4 in both bottom insets of Fig. 7. This is expected since deadlines are never missed under partitioned EDF scheduling if no cluster is overutilized. However, we also observe that proportional latency is very small (no more than about 1.2) under CA and WF cluster assignment under L2 clustering (lines 2 and 5, respectively). Proportional latency is greater under L3 clustering (lines 3 and 6), but not as great as under global scheduling (line 7) in either bottom inset of Fig. 7.

**Obs. 4.** There are trade-offs between schedulability and proportional end-to-end latency.

Schedulability is best under CA:L3 in Fig. 7. However, besides global scheduling (line 7), CA:L3 has worst proportional end-to-end latency (line 3). The converse is true for smaller clusters. These results suggest that the system designer is faced with three interesting choices: (i) minimize end-to-end latency by using the smallest cluster configuration where their application is schedulable; or (ii) increase safety margins in provisioned task execution time and use larger clusters to maintain schedulability; or (iii) decrease processor speed (effectively increasing system utilization) and use larger clusters to maintain schedulability. Choices (ii) and (iii) are extremely attractive in an automotive setting. Choice (ii) increases safety, while choice (iii) may reduce SWaP and manufacturing costs.

### 7.2 Runtime Evaluation

We now describe our runtime evaluation of PGM\(^{\text{RTT}}\). We implemented a PGM\(^{\text{RTT}}\) application that executes a DAG on LITMUS\(^{\text{RTT}}\). Upon invocation, a node performs a repeated summation over all input data from producers, and then writes output for all consumers. The invocation executes for at least a specified length of time, but will execute for longer durations under poor cache performance.

**Experimental setup.** We generated task sets composed of several DAGs according to two experimental parameters: DAG shape and the amount of data passed between nodes. DAG shape was pipelined or rectangular. In a pipelined DAG, every interior node has only one producer and one consumer. Rectangular DAGs are made of several pipelined sub-DAGs, joined by a source and sink. The amount of data passed between a producer and consumer was tested at points of [256KB, 512KB, 768KB, 1024KB, 1280KB, 1536KB, 2048KB]. We generated five task sets (trials) for each combination of DAG shape and amount of data passed. Each task set was deemed schedulable, according to our overhead-aware analysis under polluter overheads, for our 24-core evaluation platform. We assigned tasks to clusters using both the cache-aware and worst-fit decreasing heuristics. Each task set was executed for 30 seconds (both with and without job splitting) and we measured the observed end-to-end latency. Only the top-performing approaches for pipelined DAGs were tested on rectangular DAGs.

**Results.** We present the proportional end-to-end latencies observed in our experiments for pipelined DAGs and rectangular DAGs made of three pipelines. We omit results for job splitting because it provided little improvement due to low split factors. Fig. 8 depicts our results for task sets with a total utilization of 13.0 (prior to overhead accounting). The x-axis is EWSS. The y-axis is observed proportional latencies. The 99th percentile of observed proportional latencies (99% of observed latencies are less than a given y-value), averaged across the five trials, is plotted. We make the following observations.

**Obs. 5.** Observed end-to-end latencies are better for larger clusters.

In Fig. 8(a), we see that performance is largely dictated by cluster configuration rather than the cluster assignment method. For example, the observations for WF:L2 (line 5)
and CA:L2 (line 2) are relatively similar. The same trend holds for L1 and L3 clustering. We also note that L3 clustering performs well, completing the positive results from schedulability experiments for L3 clustering. Finally, we observe that global scheduling performs surprisingly well (line 7), considering that analytical results suggested that global scheduling would perform poorly.

**Obs. 6.** Cache-aware methods may yield smaller latencies.

The cache-aware heuristic outperforms the worst-fit heuristic for smaller EWSSs. However, there appears to be a trend where worst-fit performs better for larger EWSSs. In Fig. 8(a), observe the intersections between CA:L1 (line 1) and WF:L1 (line 4) at 768KB, and CA:L2 (line 2) and WF:L2 (line 5) at 1280KB. CA:L3 (line 3) and WF:L3 (line 6) do not intersect in Fig. 8(a), but do in Fig. 8(b) at 1536KB. The cache-aware heuristic sacrifices load balancing across clusters for a chance at cache efficiency. However, as cache utilization grows, actualized gains become more difficult to realize.

We conclude with the observation that cache-aware clustered scheduling methods strongly affect schedulability and end-to-end latency bounds. On our experiments, we see that these methods maximize schedulability and are competitive in analytical and observed latency. This suggests that future automotive standards should consider clustered deadline-oriented schedulers to maximize resource utilization.

### 8 Conclusion

In this paper, we have presented solutions to problems of scheduling real-time dataflow applications on multicore processors. We explored dataflow applications in advanced automotive systems, and discussed how best to meet SWaP and manufacturing costs constraints, while providing timing guarantees for safe operation and responsiveness. We derived analytical bounds on the end-to-end latency of such applications. We also developed a high-level model of cache behavior, and the overheads thereof, to drive a cache-aware heuristic we use to map dataflow computations to processors. We also presented PGMRT, a portable middleware for managing real-time dataflow applications. Using overheads obtained through PGMRT running on LITMUSRT, we performed overhead-aware schedulability experiments. The results validate our approach, and demonstrate that cache-aware techniques, coupled with clustered processor scheduling, can yield better timing properties than naïve partitioned or global scheduling. We also presented results from runtime experiments that bolster the benefits of our techniques.

In future work, we would like to extend support to accelerator coprocessors (e.g., GPUs and FPGAs), and also perform a case study with real-world automotive applications.

**Acknowledgement:** We thank Dr. Shige Wang of General Motors for his helpful comments on drafts of this paper.

### References


A End-to-End Response Time Details

In this appendix, we present theorems and proofs in support of those presented in Sec. 4.

**Theorem 2.** Suppose that the tasks in \( r \) can be assigned onto clusters of CPUs such that no individual cluster is overutilized. If \( \Theta \) is the set of all tasks along the worst-case path from \( T_1 \) to \( T_2 \), including both \( T_1 \) and \( T_2 \), then any job \( T_{1:k} \) completes within

\[
\sum_{T_i^j \in \Theta} R_i^j
\]

units of its ideal release.

**Proof.** We denote the ideal release of \( T_{1:k} \) as \( r_{1:k} \) and the actual release of \( T_{1:k} \) as \( d(T_{1:k}) \).

The theorem can be proved by induction on task depth. As the base case, we consider the source task \( T_{1:k} \). By our definition of actual releases, \( r(T_{1:k}) = r_{1:k} \). Therefore, by the definition of \( R_i^j \), \( T_{1:k} \) completes within \( R_i^j \) units of its ideal release.

For the induction step, assume that for any \( T_{q:w} \), where \( T_q \) is a producer of \( T_p \) and \( \theta \) is the set of all tasks along the worst-case path from \( T_1 \) to \( T_q \), including both \( T_1 \) and \( T_q \),

\[
t_f(T_{q:w}) - r_{q:w} \leq \sum_{T_i^j \in \Theta} R_i^j.
\]

We want to prove that

\[
t_f(T_{1:k}) - r_{1:k} \leq \sum_{T_i^j \in \Theta} R_i^j.
\]

We consider three cases.

**Case 1:** \( r(T_{1:k}) = r_{1:k} \). In this case, the lemma follows by the definition of \( R_i^j \).

**Case 2:** \( r(T_{1:k}) = t_f(T_{1:w}) \) for one of \( T_{1:k} \)'s producers \( T_{q:w} \). We have

\[
t_f(T_{1:k}) - r_{1:k} = \{\text{Rearranging}\}
\]

\[
t_f(T_{1:k}) - r(T_{1:k}) + r_{1:k} \]

\[
\leq \{\text{By the definition of } R_i^j\}
\]

\[
R_i^j + r(T_{1:k}) - r_{1:k} \]

\[
= \{\text{By the case we are considering}\}
\]

\[
R_i^j + t_f(T_{1:w}) - r_{1:k} \]

\[
= \{\text{Because } T_{1:w} \text{ is a producer of } T_{1:k}\}
\]

\[
R_i^j + t_f(T_{1:w}) - r_{1:w} \]

\[
\leq \{\text{By (1)}\}
\]

\[
R_i^j + \sum_{T_i^j \in \Theta} R_i^j
\]

\[
\leq \{\text{By the definitions of } \theta \text{ and } \Theta\}
\]

\[
\sum_{T_i^j \in \Theta} R_i^j.
\]

**Case 3:** \( k > 1 \) and \( r(T_{1:k}) = r(T_{1:k-1}) + p_i \).

We denote as \( T_{3:w} \) the last job of \( T_{1:k} \) released before \( T_{1:k} \) such that either \( r(T_{1:k}) = r_{1:w} \) or, for some producer \( T_{q:w} \) of \( T_{1:k} \),

\[
r(T_{1:w}) = t_f(T_{1:w}).
\]

Such a job must exist because \( T_{1:k} \) must satisfy this condition. By the definition of \( T_{1:w} \), we have

\[
r(T_{1:k}) = r(T_{1:w}) + (k-w) \cdot p_i.
\]

Furthermore, because \( T_{1:k} \) and \( T_{1:w} \) are from the same task, and because ideal releases follow the sporadic task model,

\[
r_{1:k} \geq r_{1:w} + (k-w) \cdot p_i.
\]

We have

\[
t_f(T_{1:k}) - r_{1:k} = \{\text{Rearranging}\}
\]

\[
t_f(T_{1:k}) - r(T_{1:k}) + r(T_{1:k}) - r_{1:k} \]

\[
\leq \{\text{By the definition of } R_i^j\}
\]

\[
R_i^j + r(T_{1:k}) - r_{1:k} \]

\[
\leq \{\text{By (3) and (4)}\}
\]

\[
R_i^j + r(T_{1:w}) + (k-w) \cdot p_i - r_{1:w} - (k-w) \cdot p_i
\]

\[
= \{\text{Rearranging}\}
\]

\[
R_i^j + r(T_{1:w}) - r_{1:w}.
\]

We now consider two subcases, based on the value of \( r(T_{1:w}) \).

**Case 3.1:** \( r(T_{1:w}) = r_{1:w} \). In this case, we have

\[
t_f(T_{1:k}) - r_{1:k} = \{\text{By (5)}\}
\]

\[
R_i^j + r(T_{1:w}) - r_{1:w}
\]

\[
= \{\text{By the case we are considering}\}
\]

\[
R_i^j + r_{1:w} - r_{1:w}
\]

\[
= \{\text{Rearranging}\}
\]

\[
R_i^j.
\]

**Case 3.2:** \( r(T_{1:w}) = t_f(T_{1:w}) \) for some producer \( T_{q:w} \) of \( T_{1:w} \). In this case, we have

\[
t_f(T_{1:k}) - r_{1:k} = \{\text{By (5)}\}
\]

\[
R_i^j + t_f(T_{1:w}) - r_{1:w}
\]

\[
= \{\text{Rearranging}\}
\]

\[
R_i^j + r_{1:w} - r_{1:w}.
\]

\[
= \{\text{Rearranging}\}
\]

\[
R_i^j.
\]
\[ R_i^j + r(T_i^{j,w}) - r_i^{j,w} \]
\[ = \{ \text{By the case we are considering} \} \]
\[ R_i^j + t_f(T_i^{j,v}) - r_i^{j,v} \]
\[ = \{ \text{Because} T_i^{j,v} \text{ is a producer of} T_i^{j,w} \} \]
\[ R_i^j + t_f(T_i^{j,v}) - r_i^{j,v} \]
\[ \leq \{ \text{By (1)} \} \]
\[ R_i^j + \sum_{T_i^j \in \theta} R_i^j \]
\[ \leq \{ \text{By the definitions of} \theta \text{ and} \Theta \} \]
\[ \sum_{T_i^j \in \Theta} R_i^j. \]

\[ \square \]

B End-to-End Response Time Bounds for DAGs with Different Periods

In this appendix, we present theorems and proofs for end-to-end response-time bounds for sporadic DAGs graphs where nodes may have different periods. Although nodes may have different periods, we assume that the source and sink node share a common period (the meaning of an end-to-end response time bound is ambiguous, otherwise).

B.1 Additional Notation

We adapt analysis for rate-based DAGs for analysis of sporadic DAGs. We require additional notation to describe rate-based a DAG. Unless stated otherwise, the same notation and graph restrictions from Sec. 3 are assumed.

A DAG-based RB task system is comprised of a set \( \tau^\text{RB} = \{ T_1, T_2, \ldots, T_n \} \) of \( n \) independent DAG-based RB tasks. Each RB task \( T_i^j \) is specified by four parameters: \( (x_i^j, y_i^j, d_i^j, c_i^j) \). The pair \( (x_i^j, y_i^j) \) represents the maximum execution rate of \( T_i^j \): \( x_i^j \) is the maximum number of invocations of the task in any interval \( t \cdot y_i^j, (t + 1) \cdot y_i^j \) \((t \geq 0)\) of length \( y_i^j \); such an invocation is called a job of the task. \( x_i^j \) and \( y_i^j \) are assumed to be non-negative integers. Additionally, \( d_i^j \) is the task’s relative deadline. The utilization of each RB task \( T_i^j \) in \( T_i \), denoted \( u_i^j \), is \( c_i^j \cdot x_i^j \). The utilization of the task system \( \tau^\text{RB} \) is \( U_{sum}(\tau^\text{RB}) = \sum_{T_i \in \tau^\text{RB}} \sum_{T_i^j \in T_i} u_i^j \). We denote the depth of a node in a DAG (as defined in Sec. 3) with the parameter \( K_i^j \).

The release time of the \( k^{th} \) job of \( T_i^j \) is at time \( r^\text{RB}(T_i^{j,k}) \) and has a deadline at time \( d^\text{RB}(T_i^{j,k}) \). For any job \( T_i^{j,k} \), \( d^\text{RB}(T_i^{j,k}) = r^\text{RB}(T_i^{j,k}) + d_i^j \). Because RB tasks may execute at different rates, and their job release times are not specifically defined (only maximum rates are specified). However, as shown in [10], job precedence constraints can be explicitly determined from PGM graph descriptions.

Finally, observe that a rate-based DAG is a sporadic DAG iff \( x_i^j = 1 \) for all tasks. In this case, \( y_i^j \) becomes equivalent to \( p_i \), the minimum separation time between a task’s jobs under the sporadic task model.

B.2 Response Time Bounds

We now derive bounds for DAGs with nodes of different periods.

Theorem 3. [18] In any G-EDF schedule for the sporadic task system \( \tau \) on \( m \) processors, if \( U_{sum}(\tau^\text{RB}) \leq m \), then the response time of any job \( T_i^{j,k} \), with respect to its delayed release \( r(T_i^{j,k}) \), is at most \( \Delta_i^j \), where \( \Delta_i^j \) is an expression depending on system parameters specified in [18], i.e.,

\[ t_f(T_i^{j,k}) - r(T_i^{j,k}) \leq \Delta_i^j + y_i^j / x_i^j, \]

where \( \Delta_i^j = \frac{\sum_{m-1 \text{ largest } e_i - \text{min } T_i \cdot e_i e_i}{m - \sum_{m-2 \text{ largest } u_i}} \).

As discussed in Sec. 4, the above holds for any window-constrained scheduler, though the derivation for \( \Delta_i^j \) may change. Theorem 3 only gives a response time bound for any job \( T_i^{j,k} \) with respect to its delayed release, \( r(T_i^{j,k}) \). This bound can have greater response time with respect to its original release, \( r^\text{RB}(T_i^{j,k}) \). Therefore, we must bound the actual response time any job \( T_i^{j,k} \) may experience with respect to its original release. The following theorem [18] gives such a bound. Let \( y_i^{max} = \max(y_i^1, y_i^2, \ldots, y_i^n) \).

Def. 1. Let \( \lambda_i^j \) denote the set of tasks that form the simple path with the largest weight connecting the source task of \( T_i \) and \( T_i^j \). Any edge connecting nodes \( T_i^j \) and \( T_i^k \) has a weight \( W_i^{jk} \) equal to the response time bound of \( T_i^j \) w.r.t. its own release time. Let \( \lambda_i^j \) denote the \textit{number} of tasks in \( \lambda_i^j \).

Theorem 4. [18] In any G-EDF schedule for \( \tau^\text{RB} \) on \( m \) processors, if \( U_{sum}(\tau^\text{RB}) \leq m \), then the response time of any job \( T_i^{j,k} \) of a task \( T_i^j \), with respect to its original release, \( r^\text{RB}(T_i^{j,k}) \), is:

\[ t_f(T_i^{j,k}) - r^\text{RB}(T_i^{j,k}) \]
\[ \leq \sum_{T_i^j \in \lambda_i^j} \Delta_i^j \cdot \frac{3 \cdot L_i^{jk} \cdot y_i^{max} + y_i^j / x_i^j}{\sum_{T_i^j \in \lambda_i^j}}, \]

where \( \Delta_i^j \) denotes the response time bound of \( T_i^j \) w.r.t. its own release time.

Def. 2. The end-to-end response time bound of a DAG \( T_i \) is given by \( \max_{z,k} \left( f(T_i^{z,k}) - r^\text{RB}(T_i^{1,k}) \right) \), where \( f(T_i^{z,k}) \) denotes the completion time of \( T_i^{z,k} \) (i.e., the \( k^{th} \) job of the sink task of \( T_i \)). This end-to-end response time bound represents the maximum duration from the release time of any job \( T_i^{1,k} \) to the completion of the corresponding job of \( T_i^{z,k} \). Since we assume that source and sink share a common period, any job \( T_i^{z,k} \) is triggered (transitively speaking) due to the completion of job \( T_i^{1,k} \).
Theorem 5. In any window-constrained schedule for $\tau$ on $m$ processors, if $\sum (\Delta_i \cdot \ell_i) \leq m$, then the end-to-end response time of any DAG $T_i$, denoted $R_i$, is at most
\[
\sum_{T_i \in \ell_i} \Delta_i + 4 \cdot L_i^{z_i} \cdot y_i^{max},
\]
where $\Delta_i$ denotes the response time bound of $T_i$ w.r.t. its redefined release time.

Proof. This theorem directly follows from Theorem 4.

\[
R_i = \max_k \left( f(T_i^{z_i,k}) - r^{BB}(T_i^{1,k}) \right) \\
\leq \max_k \left( f(T_i^{z_i,k}) - r^{BB}(T_i^{z_i,k}) \right) + (L_i^{z_i} - 1) \cdot y_i^{max} \\
\quad \{ \text{By Theorem 4} \} \\
= \left( \sum_{T_i \in \ell_i} \Delta_i + 3 \cdot L_i^{z_i} \cdot y_i^{max} + y_i^{max} / x_i^{z_i} \right) \\
\quad \{ \text{Apply } x_i^{z_i} = 1, \text{ expand, and rearrange} \} \\
= \sum_{T_i \in \ell_i} \Delta_i + 3 \cdot L_i^{z_i} \cdot y_i^{max} + L_i^{z_i} \cdot y_i^{max} \\
\quad \{ \text{Apply } y_i^{max} \} \\
\leq \sum_{T_i \in \ell_i} \Delta_i + 4 \cdot L_i^{z_i} \cdot y_i^{max} \\
\]

C PGM\textsuperscript{RT} Implementation Details

We provide additional details on the implementation of PGM\textsuperscript{RT} in this appendix—namely, the available interprocess communication mechanisms and the PGM\textsuperscript{RT} API. An overview of PGM\textsuperscript{RT}'s basic features can be found in Sec. 6.

C.1 IPC Mechanisms

Consumers block (suspend execution) whenever they lack the requisite tokens. Producers must have a mechanism to signal consumers of new tokens. The appropriate underlying interprocess communication (IPC) mechanism depends upon how tokens are used: tokens may be event-signaling or data-passing. Regardless of the underlying IPC, nodes produce and consume tokens using a common API.

With event-signaling tokens, token production and consumption is realized through increment/decrement operations on per-edge token counters, similar to counting semaphores. Although these tokens do not transmit data implicitly, tokens can coordinate data sharing in application-level logic. For example, a token may signal the availability of new data in an out-of-band queue (i.e., a data structure outside the purview of PGM\textsuperscript{RT} shared between two nodes). Signaling is achieved via a monitor synchronization primitive; specifically, a POSIX (pthread) condition variable, one per consumer. A consumer blocks on its condition variable if it does not have requisite tokens on every edge. A producer signals the condition variable whenever it is the last producer to satisfy all of its consumer’s token constraints.

The use of pthread condition variables has the drawback in that threads that utilize the synchronization primitive must first acquire a pthread mutex. This suspension-based mutex can be problematic in a real-time setting. First, it can introduce heavy overheads, due to context switching, with respect to very short critical sections in PGM\textsuperscript{RT}. Such overheads may be pessimistically accounted for in real-time analysis. Second, suspensions are difficult to model under some methods of real-time analysis (e.g., “suspension-oblivious” analysis [5]). Finally, an operating system may offer little or no support for real-time priority inheritance for pthread mutexes. As an alternative to pthread-based monitor, PGM\textsuperscript{RT} also offers a FIFO-based spinlock-based monitor built upon Linux’s “fast userspace mutex.” Its use can eliminate costly context switches and problematic suspensions. Furthermore, the duration of priority inversions are bounded since spinning tasks wait non-preemptively. On non-LITMUS\textsuperscript{RT} platforms, non-preemptive waiting is achieved by disabling CPU interrupts from the userspace (e.g., cli/pushf/popf instructions on x86 processors). On LITMUS\textsuperscript{RT}, PGM\textsuperscript{RT} utilizes LITMUS\textsuperscript{RT}’s special support for non-preemptive code sections.

With data-passing tokens, each byte-sized token is interpreted as a byte of data. Byte-ordering is preserved through FIFO-ordered token consumption. PGM\textsuperscript{RT} supports data-passing through either named pipes (FIFOs), message queues (MQs), and stream sockets (e.g., TCP)—all IPCs are POSIX-standard. We classify these mechanisms collectively as IPC channels. One channel is created for each edge. Consumers block on the set of channels, represented as a list of file descriptors, from each inbound edge using select().\textsuperscript{5} The operating system wakes the consumer whenever data arrives on a previously empty channel. Consumers read()/recv() tokens from the channel once tokens are available on all edges. Under PGM\textsuperscript{RT}, all read/write operations are non-blocking in order to avoid excessive thread suspensions. Under non-blocking writes, producers continually write data from within a loop until all data has been written. Consumers similarly loop while reading. As a fallback, consumers suspend through select() if inbound data is exhausted before the requisite number of bytes have been read. Due to the general lack of introspective capabilities with the above IPC mechanisms (specifically, the inability to query the IPC channel regarding the amount of available data), PGM consumer thresholds greater than the number of tokens consumed per node invocation are not easily supported. However, PGM\textsuperscript{RT} offers a solution to this problem that we discuss next.

The use of select() for data-passing tokens can introduce additional thread suspensions since select() wakes a blocked thread when data becomes available on any one

\textsuperscript{5}The use of select() with MQs is not strictly POSIX-compliant, but such use is commonly supported.
Regardless of the underlying IPC mechanism, PGMRT allows requests to reside on different computers in a distributed system. This enables different processes and threads connected by these channels to coordinate and delegate different nodes and edges to different processes. That is, a single graph can be shared among multiple processes, and the execution of graph algorithms can be distributed across these processes. Moreover, PGMRT allows for dynamic distribution of graphs, nodes, and edges, and the production and consumption of tokens and the passing of data to be easily managed across these processes.

### C.2 The PGMRT API

Regardless of the underlying IPC mechanism, PGMRT provides a unified API (in the C language) for the creation of graphs, nodes, and edges, and the production and consumption of tokens and the passing of data.

#### Graph management

The PGMRT runtime is initialized by a call to `pgm_init()`. Graphs are created by `pgm_init_graph()`. Mirroring the design pattern used by named FIFOs and POSIX message queues, graphs are uniquely identified by a path and name. `pgm_init_node()` is called to add a named node to a given graph. Similarly, `pgm_init_edge()` is also used to add a named edge, connecting two nodes within the same graph. The token constraints and edge type (i.e., event-signaling or specified type of a data-passing edge) are provided to `pgm_init_edge()` by the user.

All graph meta-data (e.g., graph structure, nodes, token constraints, edge types, etc.) are stored in a memory-mapped file. Graphs that have a common path share the same file. Because the meta-data file is memory-mapped, different processes can coordinate, delegating different nodes of the same graph to different processes. That is, a single graph can be executed by multiple processes. Processes and threads claim exclusive ownership over a node by calling `pgm_claim_node()`. If the particular node claimed is unimportant, then `pgm_claim_any_node()` may be called to claim a single available node. A claimed node cannot be claimed by any other process or thread until the owner has released the node with a call to `pgm_release_node()`. PGMRT provides many APIs for querying the properties of graphs, nodes, and edges. This includes APIs for obtaining the inbound or outbound degree of a node, lists of node predecessors or successors, lists of inbound or outbound edges, edge token constraints, and edge type.

#### Graph execution

It is expected that one thread of execution claims one node from a graph. Threads wait for tokens to arrive for a specified node by calling `pgm_wait()`. The calling thread will block until all token constraints have been satisfied. Once `pgm_wait()` returns, data received through data-passing edges is available in per-edge data buffers. A consumer can obtain a pointer to each buffer by calling `pgm_get_edge_buf_c()` or `pgm_get_edge_buf_p()`, which returns a `void*`. The buffer is appropriately sized (in terms of bytes) to match the number tokens the consumer consumes on each invocation. For data-passing edges, producers write data to per-edge data buffers. A producer can obtain a pointer to each buffer by calling `pgm_get_buffer_c()` or `pgm_get_buffer_p()`, which also returns a `void*`. This buffer is also appropriately sized to match the number of tokens the producer produces on each invocation. Producers may write to these buffers as necessary during their execution. A thread calls `pgm_complete()` to signal the completion of a node invocation. At this point, `pgm_complete()` pushes (writes) any data stored in data-passing edge buffers over the associated data-passing IPC. `pgm_complete()` also generates the appropriate tokens for event-signaling edges, and wakes up any consumers for which the producer was the first to satisfy the consumer’s token constraints across all of the consumer’s inbound edges.

#### More on edge buffers

Edge buffer pointers remain unchanged unless the user explicitly assigns a new buffer. That is, buffers are reused each node invocation. Thus, a thread does not need to call `pgm_get_edge_buf_c()` or `pgm_get_edge_buf_p()` each time it consumes or produces data. However, a thread can allocate additional appropriately sized buffers by calling `pgm_malloc_edge_buf_c()` or `pgm_malloc_edge_buf_p()` for consumer and producer buffers, respectively. Buffers allocated by `pgm_malloc_edge_buf_c()` and `pgm_malloc_edge_buf_p()` are initially in an unassigned state in that they are not currently used by a particular edge. These buffers can be later assigned to an edge by calling `pgm_swap_edge_buf_c()` or `pgm_swap_edge_buf_p()`, to swap an unassigned buffer with the current consumer or producer buffer, respectively. These functions return the previously assigned buffer, which is now in an unassigned state. Buffers may also be swapped between edges by calling `pgm_swap_edge_bufs()`. This function is particularly useful when a thread needs to pass consumed data directly to one of its own consumers since this avoids memory copies. A thread is responsible for freeing the memory any unassigned buffers by calling `pgm_free()` when the buffer is no longer needed.

#### Graph termination

The termination of graph execution is realized by calling `pgm_terminate()` from each graph source node. The command to terminate is automatically propagated from node to node. However, PGMRT only propagates the termination command to a node after that node has exhausted all tokens that it...
can feasibly consume (any remaining tokens are discarded). A thread is notified of termination by the return code of `pgm_wait()`. Upon termination, a thread is responsible for freeing any unassigned buffers and calling `pgm_release_node()`. A graph is completely destroyed with a call to `pgm_destroy_graph()`. Finally, the entire PGMRT runtime is torn down with a call to `pgm_destroy()`.

**Real-time OS integration.** Although PGMRT has been explicitly designed with techniques that are friendly to real-time scheduling and analysis (e.g., non-preemption and non-blocking reads and writes), PGMRT does not directly manage real-time priorities, since the mechanisms for controlling priorities may change from scheduler to scheduler and operating system to operating system. Thus, it is the responsibility of the thread to establish its scheduling priority.\(^7\) Under a fixed-priority scheduler, such as POSIX’s `SCHED_FIFO`, a thread’s priority is set during initialization. However, under LITMUSRT with deadline-based scheduling, additional coordination is necessary. Here, a thread notifies LITMUSRT that it is about to wait for tokens immediately before calling `pgm_wait()`. Based on this notification, LITMUSRT assigns the appropriate priority (deadline) if and when the thread wakes from a suspension. This thread notifies LITMUSRT that it has acquired all of its tokens immediately after `pgm_wait()` returns, preventing LITMUSRT from reassessing the thread’s priority in the future.

This completes our description of PGMRT and its API. The code for PGMRT may be found at www.github.com/GElliott/pgm or www.litmus-rt.org.

---

\(^7\) Because PGMRT is not strictly bound to a real-time operating system, it can also be used under a general (non-real-time) operating system scheduler.
D Extended Schedulability Experiment

In this appendix, we present all schedulability results for the experiments described in Sec. 7. The left-hand column in the following pages show schedulability and proportional end-to-end latencies for experiments where non-cache-polluter overheads are assumed. The corresponding right-hand column depicts results for the same schedulability experiment parameters, but under cache-polluter overheads. Cache polluter overheads have little effect on schedulability in experiments with light and medium per-task utilizations and small edge working set sizes (EWSSs). Fig. 30 illustrates one such case. Cache polluter overheads have a stronger effect in experiments with heavy per-task utilizations or large EWSSs. Fig. 339 depicts an example of this scenario.

In general, the most interesting results are those of experiments with bimodal distributions for EWSSs. Not only do cache polluter overheads generally have a stronger effect, but interesting differences among cluster configurations emerge. This can be observed in Fig. 259b, where per-task utilizations are heavy and EWSSs are bimodal. Here, cache-aware L1 and L2 clustering generally offers better schedulability for task sets with utilizations less than 17.0. However, cache-aware L3 clustering can still schedule about 30% of task sets with utilizations of 20.0, whereas cache-aware L1 and L2 clustering can schedule no more than 5% of task sets with this utilization.

Finally, we found that graph depth had little effect on proportional end-to-end latency.
Figure 9: Results for uni-light per-task utilization, uni-short period, light-weight EWSS, and uni-short height factor.

Figure 10: Results for uni-light per-task utilization, uni-short period, light-weight EWSS, and uni-medium height factor.

Figure 11: Results for uni-light per-task utilization, uni-short period, light-weight EWSS, and uni-tall height factor.
Figure 12: Results for uni-light per-task utilization, uni-short period, light-weight EWSS, and pipeline height factor.

Figure 13: Results for uni-light per-task utilization, uni-moderate period, light-weight EWSS, and uni-short height factor.

Figure 14: Results for uni-light per-task utilization, uni-moderate period, light-weight EWSS, and uni-medium height factor.
Figure 15: Results for uni-light per-task utilization, uni-moderate period, light-weight EWSS, and uni-tall height factor.

Figure 16: Results for uni-light per-task utilization, uni-moderate period, light-weight EWSS, and pipeline height factor.

Figure 17: Results for uni-light per-task utilization, uni-long period, light-weight EWSS, and uni-short height factor.
Figure 18: Results for uni-light per-task utilization, uni-long period, light-weight EWSS, and uni-medium height factor.

Figure 19: Results for uni-light per-task utilization, uni-long period, light-weight EWSS, and uni-tall height factor.

Figure 20: Results for uni-light per-task utilization, uni-long period, light-weight EWSS, and pipeline height factor.
Figure 21: Results for uni-medium per-task utilization, uni-short period, light-weight EWSS, and uni-short height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 22: Results for uni-medium per-task utilization, uni-short period, light-weight EWSS, and uni-medium height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 23: Results for uni-medium per-task utilization, uni-short period, light-weight EWSS, and uni-tall height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 24: Results for uni-medium per-task utilization, uni-short period, light-weight EWSS, and pipeline height factor.

Figure 25: Results for uni-medium per-task utilization, uni-moderate period, light-weight EWSS, and uni-short height factor.

Figure 26: Results for uni-medium per-task utilization, uni-moderate period, light-weight EWSS, and uni-medium height factor.
Figure 27: Results for uni-medium per-task utilization, uni-moderate period, light-weight EWSS, and uni-tall height factor.

Figure 28: Results for uni-medium per-task utilization, uni-moderate period, light-weight EWSS, and pipeline height factor.

Figure 29: Results for uni-medium per-task utilization, uni-long period, light-weight EWSS, and uni-short height factor.
Figure 30: Results for uni-medium per-task utilization, uni-long period, light-weight EWSS, and uni-medium height factor.

Figure 31: Results for uni-medium per-task utilization, uni-long period, light-weight EWSS, and uni-tall height factor.

Figure 32: Results for uni-medium per-task utilization, uni-long period, light-weight EWSS, and pipeline height factor.
Figure 33: Results for uni-heavy per-task utilization, uni-short period, light-weight EWSS, and uni-short height factor.

Figure 34: Results for uni-heavy per-task utilization, uni-short period, light-weight EWSS, and uni-medium height factor.

Figure 35: Results for uni-heavy per-task utilization, uni-short period, light-weight EWSS, and uni-tall height factor.
Figure 36: Results for uni-heavy per-task utilization, uni-short period, light-weight EWSS, and pipeline height factor.

Figure 37: Results for uni-heavy per-task utilization, uni-moderate period, light-weight EWSS, and uni-short height factor.

Figure 38: Results for uni-heavy per-task utilization, uni-moderate period, light-weight EWSS, and uni-medium height factor.
Figure 39: Results for uni-heavy per-task utilization, uni-moderate period, light-weight EWSS, and uni-tall height factor.

Figure 40: Results for uni-heavy per-task utilization, uni-moderate period, light-weight EWSS, and pipeline height factor.

Figure 41: Results for uni-heavy per-task utilization, uni-long period, light-weight EWSS, and uni-short height factor.
Results for *uni-heavy* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-medium* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 42:

Results for *uni-heavy* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-tall* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 43:

Results for *uni-heavy* per-task utilization, *uni-long* period, *light-weight* EWSS, and *pipeline* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 44:
Average Proportional Latency

Schedulable (bounded response time)

Figure 45: Results for bimo-light per-task utilization, uni-short period, light-weight EWSS, and uni-short height factor.

Figure 46: Results for bimo-light per-task utilization, uni-short period, light-weight EWSS, and uni-medium height factor.

Figure 47: Results for bimo-light per-task utilization, uni-short period, light-weight EWSS, and uni-tall height factor.
Figure 48: Results for *bimo-light* per-task utilization, *uni-short* period, *light-weight* EWSS, and *pipeline* height factor.

Figure 49: Results for *bimo-light* per-task utilization, *uni-moderate* period, *light-weight* EWSS, and *uni-short* height factor.

Figure 50: Results for *bimo-light* per-task utilization, *uni-moderate* period, *light-weight* EWSS, and *uni-medium* height factor.
Figure 51: Results for *bimo-light* per-task utilization, *uni-moderate* period, *light-weight* EWSS, and *uni-tall* height factor.

Figure 52: Results for *bimo-light* per-task utilization, *uni-moderate* period, *light-weight* EWSS, and *pipeline* height factor.

Figure 53: Results for *bimo-light* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-short* height factor.
Figure 54: Results for *bimo-light* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-medium* height factor.

Figure 55: Results for *bimo-light* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-tall* height factor.

Figure 56: Results for *bimo-light* per-task utilization, *uni-long* period, *light-weight* EWSS, and *pipeline* height factor.
Figure 57: Results for bimo-medium per-task utilization, uni-short period, light-weight EWSS, and uni-short height factor.

Figure 58: Results for bimo-medium per-task utilization, uni-short period, light-weight EWSS, and uni-medium height factor.

Figure 59: Results for bimo-medium per-task utilization, uni-short period, light-weight EWSS, and uni-tall height factor.
Figure 60: Results for \textit{bimo-medium} per-task utilization, \textit{uni-short} period, \textit{light-weight} EWSS, and \textit{pipeline} height factor.

Figure 61: Results for \textit{bimo-medium} per-task utilization, \textit{uni-moderate} period, \textit{light-weight} EWSS, and \textit{uni-short} height factor.

Figure 62: Results for \textit{bimo-medium} per-task utilization, \textit{uni-moderate} period, \textit{light-weight} EWSS, and \textit{uni-medium} height factor.
Figure 63: Results for *bimo-medium* per-task utilization, *uni-moderate* period, *light-weight* EWSS, and *uni-tall* height factor.

Figure 64: Results for *bimo-medium* per-task utilization, *uni-moderate* period, *light-weight* EWSS, and *pipeline* height factor.

Figure 65: Results for *bimo-medium* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-short* height factor.
Figure 66: Results for bimo-medium per-task utilization, uni-long period, light-weight EWSS, and uni-medium height factor.

Figure 67: Results for bimo-medium per-task utilization, uni-long period, light-weight EWSS, and uni-tall height factor.

Figure 68: Results for bimo-medium per-task utilization, uni-long period, light-weight EWSS, and pipeline height factor.
Figure 69: Results for bimo-heavy per-task utilization, uni-short period, light-weight EWSS, and uni-short height factor.

Figure 70: Results for bimo-heavy per-task utilization, uni-short period, light-weight EWSS, and uni-medium height factor.

Figure 71: Results for bimo-heavy per-task utilization, uni-short period, light-weight EWSS, and uni-tall height factor.
Figure 72: Results for bimo-heavy per-task utilization, uni-short period, light-weight EWSS, and pipeline height factor.

Figure 73: Results for bimo-heavy per-task utilization, uni-moderate period, light-weight EWSS, and uni-short height factor.

Figure 74: Results for bimo-heavy per-task utilization, uni-moderate period, light-weight EWSS, and uni-medium height factor.
Figure 75: Results for bimo-heavy per-task utilization, uni-moderate period, light-weight EWSS, and uni-tall height factor.

Figure 76: Results for bimo-heavy per-task utilization, uni-moderate period, light-weight EWSS, and pipeline height factor.

Figure 77: Results for bimo-heavy per-task utilization, uni-long period, light-weight EWSS, and uni-short height factor.
Figure 78: Results for *bimo-heavy* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-medium* height factor.

Figure 79: Results for *bimo-heavy* per-task utilization, *uni-long* period, *light-weight* EWSS, and *uni-tall* height factor.

Figure 80: Results for *bimo-heavy* per-task utilization, *uni-long* period, *light-weight* EWSS, and *pipeline* height factor.
Figure 81: Results for *uni-light* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *uni-short* height factor.

Figure 82: Results for *uni-light* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *uni-medium* height factor.

Figure 83: Results for *uni-light* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *uni-tall* height factor.
Figure 84: Results for uni-light per-task utilization, uni-short period, medium-weight EWSS, and pipeline height factor.

Figure 85: Results for uni-light per-task utilization, uni-moderate period, medium-weight EWSS, and uni-short height factor.

Figure 86: Results for uni-light per-task utilization, uni-moderate period, medium-weight EWSS, and uni-medium height factor.
Figure 87: Results for uni-light per-task utilization, uni-moderate period, medium-weight EWSS, and uni-tall height factor.

Figure 88: Results for uni-light per-task utilization, uni-moderate period, medium-weight EWSS, and pipeline height factor.

Figure 89: Results for uni-light per-task utilization, uni-long period, medium-weight EWSS, and uni-short height factor.
Figure 90: Results for uni-light per-task utilization, uni-long period, medium-weight EWSS, and uni-medium height factor.

Figure 91: Results for uni-light per-task utilization, uni-long period, medium-weight EWSS, and uni-tall height factor.

Figure 92: Results for uni-light per-task utilization, uni-long period, medium-weight EWSS, and pipeline height factor.
Figure 93: Results for uni-medium per-task utilization, uni-short period, medium-weight EWSS, and uni-short height factor.

Figure 94: Results for uni-medium per-task utilization, uni-short period, medium-weight EWSS, and uni-medium height factor.

Figure 95: Results for uni-medium per-task utilization, uni-short period, medium-weight EWSS, and uni-tall height factor.
Figure 96: Results for uni-medium per-task utilization, uni-short period, medium-weight EWSS, and pipeline height factor.

Figure 97: Results for uni-medium per-task utilization, uni-moderate period, medium-weight EWSS, and uni-short height factor.

Figure 98: Results for uni-medium per-task utilization, uni-moderate period, medium-weight EWSS, and uni-medium height factor.
Figure 99: Results for uni-medium per-task utilization, uni-moderate period, medium-weight EWSS, and uni-tall height factor.

Figure 100: Results for uni-medium per-task utilization, uni-moderate period, medium-weight EWSS, and pipeline height factor.

Figure 101: Results for uni-medium per-task utilization, uni-long period, medium-weight EWSS, and uni-short height factor.
Figure 102: Results for uni-medium per-task utilization, uni-long period, medium-weight EWSS, and uni-medium height factor.

Figure 103: Results for uni-medium per-task utilization, uni-long period, medium-weight EWSS, and uni-tall height factor.

Figure 104: Results for uni-medium per-task utilization, uni-long period, medium-weight EWSS, and pipeline height factor.
Figure 105: Results for uni-heavy per-task utilization, uni-short period, medium-weight EWSS, and uni-short height factor.

Figure 106: Results for uni-heavy per-task utilization, uni-short period, medium-weight EWSS, and uni-medium height factor.

Figure 107: Results for uni-heavy per-task utilization, uni-short period, medium-weight EWSS, and uni-tall height factor.
Figure 108: Results for uni-heavy per-task utilization, uni-short period, medium-weight EWSS, and pipeline height factor.

Figure 109: Results for uni-heavy per-task utilization, uni-moderate period, medium-weight EWSS, and uni-short height factor.

Figure 110: Results for uni-heavy per-task utilization, uni-moderate period, medium-weight EWSS, and uni-medium height factor.
Figure 111: Results for uni-heavy per-task utilization, uni-moderate period, medium-weight EWSS, and uni-tall height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 112: Results for uni-heavy per-task utilization, uni-moderate period, medium-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 113: Results for uni-heavy per-task utilization, uni-long period, medium-weight EWSS, and uni-short height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 114: Results for uni-heavy per-task utilization, uni-long period, medium-weight EWSS, and uni-medium height factor.

Figure 115: Results for uni-heavy per-task utilization, uni-long period, medium-weight EWSS, and uni-tall height factor.

Figure 116: Results for uni-heavy per-task utilization, uni-long period, medium-weight EWSS, and pipeline height factor.
Figure 117: Results for *bimo-light* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *uni-short* height factor.

Figure 118: Results for *bimo-light* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *uni-medium* height factor.

Figure 119: Results for *bimo-light* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *uni-tall* height factor.
Figure 120: Results for bimo-light per-task utilization, uni-short period, medium-weight EWSS, and pipeline height factor.

Figure 121: Results for bimo-light per-task utilization, uni-moderate period, medium-weight EWSS, and uni-short height factor.

Figure 122: Results for bimo-light per-task utilization, uni-moderate period, medium-weight EWSS, and uni-medium height factor.
Figure 123: Results for bimo-light per-task utilization, uni-moderate period, medium-weight EWSS, and uni-tall height factor.

Figure 124: Results for bimo-light per-task utilization, uni-moderate period, medium-weight EWSS, and pipeline height factor.

Figure 125: Results for bimo-light per-task utilization, uni-long period, medium-weight EWSS, and uni-short height factor.
Figure 126: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{medium-weight} EWSS, and \textit{uni-medium} height factor.

Figure 127: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{medium-weight} EWSS, and \textit{uni-tall} height factor.

Figure 128: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{medium-weight} EWSS, and \textit{pipeline} height factor.
Figure 129: Results for bimo-medium per-task utilization, uni-short period, medium-weight EWSS, and uni-short height factor.

Figure 130: Results for bimo-medium per-task utilization, uni-short period, medium-weight EWSS, and uni-medium height factor.

Figure 131: Results for bimo-medium per-task utilization, uni-short period, medium-weight EWSS, and uni-tall height factor.
Results for Figure 132: Results for bimo-medium per-task utilization, uni-short period, medium-weight EWSS, and pipeline height factor.

Results for Figure 133: Results for bimo-medium per-task utilization, uni-moderate period, medium-weight EWSS, and uni-short height factor.

Results for Figure 134: Results for bimo-medium per-task utilization, uni-moderate period, medium-weight EWSS, and uni-medium height factor.
Figure 135: Results for bimo-medium per-task utilization, uni-moderate period, medium-weight EWSS, and uni-tall height factor.

Figure 136: Results for bimo-medium per-task utilization, uni-moderate period, medium-weight EWSS, and pipeline height factor.

Figure 137: Results for bimo-medium per-task utilization, uni-long period, medium-weight EWSS, and uni-short height factor.
Figure 138: Results for *bimo-medium* per-task utilization, *uni-long* period, *medium-weight* EWSS, and *uni-medium* height factor.

Figure 139: Results for *bimo-medium* per-task utilization, *uni-long* period, *medium-weight* EWSS, and *uni-tall* height factor.

Figure 140: Results for *bimo-medium* per-task utilization, *uni-long* period, *medium-weight* EWSS, and *pipeline* height factor.
Figure 141: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-short} period, \textit{medium-weight} EWSS, and \textit{uni-short} height factor.

Figure 142: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-short} period, \textit{medium-weight} EWSS, and \textit{uni-medium} height factor.

Figure 143: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-short} period, \textit{medium-weight} EWSS, and \textit{uni-tall} height factor.
Figure 144: Results for *bimo-heavy* per-task utilization, *uni-short* period, *medium-weight* EWSS, and *pipeline* height factor.

Figure 145: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *medium-weight* EWSS, and *uni-short* height factor.

Figure 146: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *medium-weight* EWSS, and *uni-medium* height factor.
Figure 147: Results for bimo-heavy per-task utilization, uni-moderate period, medium-weight EWSS, and uni-tall height factor.

Figure 148: Results for bimo-heavy per-task utilization, uni-moderate period, medium-weight EWSS, and pipeline height factor.

Figure 149: Results for bimo-heavy per-task utilization, uni-long period, medium-weight EWSS, and uni-short height factor.
Figure 150: Results for bimo-heavy per-task utilization, uni-long period, medium-weight EWSS, and uni-medium height factor.

Figure 151: Results for bimo-heavy per-task utilization, uni-long period, medium-weight EWSS, and uni-tall height factor.

Figure 152: Results for bimo-heavy per-task utilization, uni-long period, medium-weight EWSS, and pipeline height factor.
(a) Without polluter overheads.

(b) With polluter overheads.

Figure 153: Results for *uni-light* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-short* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 154: Results for *uni-light* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-medium* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 155: Results for *uni-light* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-tall* height factor.
Figure 156: Results for uni-light per-task utilization, uni-short period, heavy-weight EWSS, and pipeline height factor.

Figure 157: Results for uni-light per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-short height factor.

Figure 158: Results for uni-light per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-medium height factor.
Figure 159: Results for uni-light per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-tall height factor.

Figure 160: Results for uni-light per-task utilization, uni-moderate period, heavy-weight EWSS, and pipeline height factor.

Figure 161: Results for uni-light per-task utilization, uni-long period, heavy-weight EWSS, and uni-short height factor.
Figure 162: Results for uni-light per-task utilization, uni-long period, heavy-weight EWSS, and uni-medium height factor.

Figure 163: Results for uni-light per-task utilization, uni-long period, heavy-weight EWSS, and uni-tall height factor.

Figure 164: Results for uni-light per-task utilization, uni-long period, heavy-weight EWSS, and pipeline height factor.
Figure 165: Results for uni-medium per-task utilization, uni-short period, heavy-weight EWSS, and uni-short height factor.

Figure 166: Results for uni-medium per-task utilization, uni-short period, heavy-weight EWSS, and uni-medium height factor.

Figure 167: Results for uni-medium per-task utilization, uni-short period, heavy-weight EWSS, and uni-tall height factor.
### Figure 168: Results for uni-medium per-task utilization, uni-short period, heavy-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

### Figure 169: Results for uni-medium per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-short height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

### Figure 170: Results for uni-medium per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-medium height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 171: Results for *uni-medium* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *uni-tall* height factor.

Figure 172: Results for *uni-medium* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *pipeline* height factor.

Figure 173: Results for *uni-medium* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-short* height factor.
Figure 174: Results for uni-medium per-task utilization, uni-long period, heavy-weight EWSS, and uni-medium height factor.

Figure 175: Results for uni-medium per-task utilization, uni-long period, heavy-weight EWSS, and uni-tall height factor.

Figure 176: Results for uni-medium per-task utilization, uni-long period, heavy-weight EWSS, and pipeline height factor.
Figure 177: Results for uni-heavy per-task utilization, uni-short period, heavy-weight EWSS, and uni-short height factor.

Figure 178: Results for uni-heavy per-task utilization, uni-short period, heavy-weight EWSS, and uni-medium height factor.

Figure 179: Results for uni-heavy per-task utilization, uni-short period, heavy-weight EWSS, and uni-tall height factor.
Figure 180: Results for uni-heavy per-task utilization, uni-short period, heavy-weight EWSS, and pipeline height factor

Figure 181: Results for uni-heavy per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-short height factor

Figure 182: Results for uni-heavy per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-medium height factor
Figure 183: Results for *uni-heavy* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *uni-tall* height factor.

Figure 184: Results for *uni-heavy* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and pipeline height factor.

Figure 185: Results for *uni-heavy* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-short* height factor.
Figure 186: Results for *uni-heavy* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-medium* height factor.

Figure 187: Results for *uni-heavy* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-tall* height factor.

Figure 188: Results for *uni-heavy* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *pipeline* height factor.
Figure 189: Results for *bimo-light* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-short* height factor.

Figure 190: Results for *bimo-light* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-medium* height factor.

Figure 191: Results for *bimo-light* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-tall* height factor.
Figure 192: Results for \textit{bimo-light} per-task utilization, \textit{uni-short} period, \textit{heavy-weight} EWSS, and \textit{pipeline} height factor.

(a) Without polluter overheads.
(b) With polluter overheads.

Figure 193: Results for \textit{bimo-light} per-task utilization, \textit{uni-moderate} period, \textit{heavy-weight} EWSS, and \textit{uni-short} height factor.

(a) Without polluter overheads.
(b) With polluter overheads.

Figure 194: Results for \textit{bimo-light} per-task utilization, \textit{uni-moderate} period, \textit{heavy-weight} EWSS, and \textit{uni-medium} height factor.

(a) Without polluter overheads.
(b) With polluter overheads.
Figure 195: Results for *bimo-light* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *uni-tall* height factor.  

Figure 196: Results for *bimo-light* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *pipeline* height factor.  

Figure 197: Results for *bimo-light* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-short* height factor.
Figure 198: Results for *bimo-light* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-medium* height factor.

Figure 199: Results for *bimo-light* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *uni-tall* height factor.

Figure 200: Results for *bimo-light* per-task utilization, *uni-long* period, *heavy-weight* EWSS, and *pipeline* height factor.
Figure 201: Results for *bimo-medium* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-short* height factor.

Figure 202: Results for *bimo-medium* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-medium* height factor.

Figure 203: Results for *bimo-medium* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-tall* height factor.
Figure 204: Results for *bimo-medium* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *pipeline* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 205: Results for *bimo-medium* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *uni-short* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 206: Results for *bimo-medium* per-task utilization, *uni-moderate* period, *heavy-weight* EWSS, and *uni-medium* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 207: Results for bimo-medium per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-tall height factor.

Figure 208: Results for bimo-medium per-task utilization, uni-moderate period, heavy-weight EWSS, and pipeline height factor.

Figure 209: Results for bimo-medium per-task utilization, uni-long period, heavy-weight EWSS, and uni-short height factor.
Figure 210: Results for bimo-medium per-task utilization, uni-long period, heavy-weight EWSS, and uni-medium height factor.

Figure 211: Results for bimo-medium per-task utilization, uni-long period, heavy-weight EWSS, and uni-tall height factor.

Figure 212: Results for bimo-medium per-task utilization, uni-long period, heavy-weight EWSS, and pipeline height factor.
Figure 213: Results for *bimo-heavy* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-short* height factor.

Figure 214: Results for *bimo-heavy* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-medium* height factor.

Figure 215: Results for *bimo-heavy* per-task utilization, *uni-short* period, *heavy-weight* EWSS, and *uni-tall* height factor.
Figure 216: Results for bimo-heavy per-task utilization, uni-short period, heavy-weight EWSS, and pipeline height factor.

Figure 217: Results for bimo-heavy per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-short height factor.

Figure 218: Results for bimo-heavy per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-medium height factor.
Figure 219: Results for bimo-heavy per-task utilization, uni-moderate period, heavy-weight EWSS, and uni-tall height factor.

Figure 220: Results for bimo-heavy per-task utilization, uni-moderate period, heavy-weight EWSS, and pipeline height factor.

Figure 221: Results for bimo-heavy per-task utilization, uni-long period, heavy-weight EWSS, and uni-short height factor.
Figure 222: Results for bimo-heavy per-task utilization, uni-long period, heavy-weight EWSS, and uni-medium height factor.

Figure 223: Results for bimo-heavy per-task utilization, uni-long period, heavy-weight EWSS, and uni-tall height factor.

Figure 224: Results for bimo-heavy per-task utilization, uni-long period, heavy-weight EWSS, and pipeline height factor.
Figure 225: Results for uni-light per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-short height factor.

Figure 226: Results for uni-light per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-medium height factor.

Figure 227: Results for uni-light per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-tall height factor.
Figure 228: Results for *uni-light* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *pipeline* height factor.

Figure 229: Results for *uni-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-short* height factor.

Figure 230: Results for *uni-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.
Figure 231: Results for *uni-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-tall* height factor.

Figure 232: Results for *uni-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *pipeline* height factor.

Figure 233: Results for *uni-light* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *uni-short* height factor.
Figure 234: Results for uni-light per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-medium height factor.

Figure 235: Results for uni-light per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-tall height factor.

Figure 236: Results for uni-light per-task utilization, uni-long period, bimo-light-weight EWSS, and pipeline height factor.
Figure 237: Results for uni-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-short height factor.

Figure 238: Results for uni-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-medium height factor.

Figure 239: Results for uni-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-tall height factor.
Figure 240: Results for uni-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and pipeline height factor.

Figure 241: Results for uni-medium per-task utilization, uni-moderate period, bimo-light-weight EWSS, and uni-short height factor.

Figure 242: Results for uni-medium per-task utilization, uni-moderate period, bimo-light-weight EWSS, and uni-medium height factor.
Figure 243: Results for uni-medium per-task utilization, uni-moderate period, bimo-light-weight EWSS, and uni-tall height factor.

Figure 244: Results for uni-medium per-task utilization, uni-moderate period, bimo-light-weight EWSS, and pipeline height factor.

Figure 245: Results for uni-medium per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-short height factor.
Figure 246: Results for uni-medium per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-medium height factor.

Figure 247: Results for uni-medium per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-tall height factor.

Figure 248: Results for uni-medium per-task utilization, uni-long period, bimo-light-weight EWSS, and pipeline height factor.
Figure 249: Results for uni-heavy per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-short height factor.

Figure 250: Results for uni-heavy per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-medium height factor.

Figure 251: Results for uni-heavy per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-tall height factor.
Figure 252: Results for *uni-heavy* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *pipeline* height factor.

Figure 253: Results for *uni-heavy* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-short* height factor.

Figure 254: Results for *uni-heavy* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.
Figure 255: Results for uni-heavy per-task utilization, uni-moderate period, bimo-light-weight EWSS, and uni-tall height factor.

Figure 256: Results for uni-heavy per-task utilization, uni-moderate period, bimo-light-weight EWSS, and pipeline height factor.

Figure 257: Results for uni-heavy per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-short height factor.
Average Proportional Latency

Schedulable (bounded response time)

0

0

1

1

2

2

3

3

4

4

5

5

6

6

7

7

8

8

9

9

10

10

11

11

12

12

13

13

14

14

15

15

16

16

17

17

18

18

19

19

20

20


System Utilization (prior inflations)

0.0

0.2

0.4

0.6

0.8

1.0

Schedulable (bounded response time)

Proportional Latency

Results for uni-heavy per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-medium height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 258:

Results for uni-heavy per-task utilization, uni-long period, bimo-light-weight EWSS, and uni-tall height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 259:

Results for uni-heavy per-task utilization, uni-long period, bimo-light-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 260:
Figure 261: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *uni-short* height factor.

Figure 262: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.

Figure 263: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *uni-tall* height factor.
Figure 264: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and pipeline height factor.

Figure 265: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-short* height factor.

Figure 266: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.
Figure 267: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *uni-tall* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 268: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-light-weight* EWSS, and *pipeline* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 269: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *uni-short* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 270: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.

Figure 271: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *uni-tall* height factor.

Figure 272: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *pipeline* height factor.
Figure 273: Results for bimo-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-short height factor.

Figure 274: Results for bimo-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-medium height factor.

Figure 275: Results for bimo-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and uni-tall height factor.
Figure 276: Results for bimo-medium per-task utilization, uni-short period, bimo-light-weight EWSS, and pipeline height factor.

Figure 277: Results for bimo-medium per-task utilization, uni-moderate period, bimo-light-weight EWSS, and uni-short height factor.

Figure 278: Results for bimo-medium per-task utilization, uni-moderate period, bimo-light-weight EWSS, and uni-medium height factor.
Figure 279: Results for \textit{bimo-medium} per-task utilization, \textit{uni-moderate} period, \textit{bimo-light-weight} EWSS, and \textit{uni-tall} height factor.

Figure 280: Results for \textit{bimo-medium} per-task utilization, \textit{uni-moderate} period, \textit{bimo-light-weight} EWSS, and \textit{pipeline} height factor.

Figure 281: Results for \textit{bimo-medium} per-task utilization, \textit{uni-long} period, \textit{bimo-light-weight} EWSS, and \textit{uni-short} height factor.
Figure 282: Results for \textit{bimo-medium} per-task utilization, \textit{uni-long} period, \textit{bimo-light-weight} EWSS, and \textit{uni-medium} height factor.

Figure 283: Results for \textit{bimo-medium} per-task utilization, \textit{uni-long} period, \textit{bimo-light-weight} EWSS, and \textit{uni-tall} height factor.

Figure 284: Results for \textit{bimo-medium} per-task utilization, \textit{uni-long} period, \textit{bimo-light-weight} EWSS, and \textit{pipeline} height factor.
Figure 285: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *uni-short* height factor.

Figure 286: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.

Figure 287: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-light-weight* EWSS, and *uni-tall* height factor.
Figure 288: Results for `bimo-heavy` per-task utilization, `uni-short` period, `bimo-light-weight` EWSS, and `pipeline` height factor.

Figure 289: Results for `bimo-heavy` per-task utilization, `uni-moderate` period, `bimo-light-weight` EWSS, and `uni-short` height factor.

Figure 290: Results for `bimo-heavy` per-task utilization, `uni-moderate` period, `bimo-light-weight` EWSS, and `uni-medium` height factor.
Figure 291: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-moderate} period, \textit{bimo-light-weight} EWSS, and \textit{uni-tall} height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 292: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-moderate} period, \textit{bimo-light-weight} EWSS, and \textit{pipeline} height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 293: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-long} period, \textit{bimo-light-weight} EWSS, and \textit{uni-short} height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 294: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *uni-medium* height factor.

Figure 295: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *uni-tall* height factor.

Figure 296: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-light-weight* EWSS, and *pipeline* height factor.
Figure 297: Results for uni-light per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-short height factor.

Figure 298: Results for uni-light per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-medium height factor.

Figure 299: Results for uni-light per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-tall height factor.
Figure 300: Results for **uni-light** per-task utilization, **uni-short** period, **bimo-medium-weight** EWSS, and **pipeline** height factor.

Figure 301: Results for **uni-light** per-task utilization, **uni-moderate** period, **bimo-medium-weight** EWSS, and **uni-short** height factor.

Figure 302: Results for **uni-light** per-task utilization, **uni-moderate** period, **bimo-medium-weight** EWSS, and **uni-medium** height factor.
Figure 304: Results for *uni-light* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-tall* height factor.

Figure 305: Results for *uni-light* per-task utilization, *uni-long* period, *bimo-medium-weight* EWSS, and *uni-short* height factor.
Figure 306: Results for uni-light per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-medium height factor.

Figure 307: Results for uni-light per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-tall height factor.

Figure 308: Results for uni-light per-task utilization, uni-long period, bimo-medium-weight EWSS, and pipeline height factor.
Figure 309: Results for uni-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-short height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 310: Results for uni-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-medium height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 311: Results for uni-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-tall height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 312: Results for uni-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and pipeline height factor.

Figure 313: Results for uni-medium per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-short height factor.

Figure 314: Results for uni-medium per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-medium height factor.
Figure 315: Results for *uni-medium* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-tall* height factor.

Figure 316: Results for *uni-medium* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *pipeline* height factor.

Figure 317: Results for *uni-medium* per-task utilization, *uni-long* period, *bimo-medium-weight* EWSS, and *uni-short* height factor.
Figure 318: Results for uni-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-medium height factor.

Figure 319: Results for uni-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-tall height factor.

Figure 320: Results for uni-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and pipeline height factor.
Figure 321: Results for \textit{uni-heavy} per-task utilization, \textit{uni-short} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-short} height factor.

Figure 322: Results for \textit{uni-heavy} per-task utilization, \textit{uni-short} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-medium} height factor.

Figure 323: Results for \textit{uni-heavy} per-task utilization, \textit{uni-short} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-tall} height factor.
Figure 324: Results for uni-heavy per-task utilization, uni-short period, bimo-medium-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 325: Results for uni-heavy per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-short height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 326: Results for uni-heavy per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-medium height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 327: Results for uni-heavy per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-tall height factor.

Figure 328: Results for uni-heavy per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and pipeline height factor.

Figure 329: Results for uni-heavy per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-short height factor.
Figure 330: Results for uni-heavy per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-medium height factor.

Figure 331: Results for uni-heavy per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-tall height factor.

Figure 332: Results for uni-heavy per-task utilization, uni-long period, bimo-medium-weight EWSS, and pipeline height factor.
Figure 333: Results for bimo-light per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-short height factor.

Figure 334: Results for bimo-light per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-medium height factor.

Figure 335: Results for bimo-light per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-tall height factor.
Figure 336: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-medium-weight* EWSS, and pipeline height factor.

Figure 337: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-short* height factor.

Figure 338: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-medium* height factor.
Figure 339: Results for \textit{bimo-light} per-task utilization, \textit{uni-moderate} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-tall} height factor.

Figure 340: Results for \textit{bimo-light} per-task utilization, \textit{uni-moderate} period, \textit{bimo-medium-weight} EWSS, and \textit{pipeline} height factor.

Figure 341: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-short} height factor.
Figure 342: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-medium} height factor.

Figure 343: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{bimo-medium-weight} EWSS, and \textit{uni-tall} height factor.

Figure 344: Results for \textit{bimo-light} per-task utilization, \textit{uni-long} period, \textit{bimo-medium-weight} EWSS, and \textit{pipeline} height factor.
Figure 345: Results for bimo-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-short height factor.

Figure 346: Results for bimo-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-medium height factor.

Figure 347: Results for bimo-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and uni-tall height factor.
Figure 348: Results for bimo-medium per-task utilization, uni-short period, bimo-medium-weight EWSS, and pipeline height factor.

Figure 349: Results for bimo-medium per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-short height factor.

Figure 350: Results for bimo-medium per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-medium height factor.
Figure 351: Results for bimo-medium per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and uni-tall height factor.

Figure 352: Results for bimo-medium per-task utilization, uni-moderate period, bimo-medium-weight EWSS, and pipeline height factor.

Figure 353: Results for bimo-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-short height factor.
Figure 354: Results for bimo-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-medium height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 355: Results for bimo-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and uni-tall height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 356: Results for bimo-medium per-task utilization, uni-long period, bimo-medium-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 357: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-medium-weight* EWSS, and *uni-short* height factor.

Figure 358: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-medium-weight* EWSS, and *uni-medium* height factor.

Figure 359: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-medium-weight* EWSS, and *uni-tall* height factor.
Figure 360: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-medium-weight* EWSS, and *pipeline* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 361: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-short* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 362: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-medium* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 363: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *uni-tall* height factor.

Figure 364: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *bimo-medium-weight* EWSS, and *pipeline* height factor.

Figure 365: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-medium-weight* EWSS, and *uni-short* height factor.
Figure 366: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-medium-weight* EWSS, and *uni-medium* height factor.

Figure 367: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-medium-weight* EWSS, and *uni-tall* height factor.

Figure 368: Results for *bimo-heavy* per-task utilization, *uni-long* period, *bimo-medium-weight* EWSS, and *pipeline* height factor.
Figure 369: Results for uni-light per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-short height factor.

Figure 370: Results for uni-light per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-medium height factor.

Figure 371: Results for uni-light per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-tall height factor.
Figure 372: Results for \textit{uni-light} per-task utilization, \textit{uni-short} period, \textit{bimo-heavy-weight} EWSS, and \textit{pipeline} height factor.

Figure 373: Results for \textit{uni-light} per-task utilization, \textit{uni-moderate} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-short} height factor.

Figure 374: Results for \textit{uni-light} per-task utilization, \textit{uni-moderate} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-medium} height factor.
Average Proportional Latency

Schedulable (bounded response time)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 5 10 15 20

System Utilization (prior inflations)

Figure 375: Results for uni-light per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-tall height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 376: Results for uni-light per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 377: Results for uni-light per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-short height factor.
Figure 378: Results for *uni-light* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.

Figure 379: Results for *uni-light* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-tall* height factor.

Figure 380: Results for *uni-light* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and pipeline height factor.
Figure 381: Results for \textit{uni-medium} per-task utilization, \textit{uni-short} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-short} height factor.

Figure 382: Results for \textit{uni-medium} per-task utilization, \textit{uni-short} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-medium} height factor.

Figure 383: Results for \textit{uni-medium} per-task utilization, \textit{uni-short} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-tall} height factor.
Figure 384: Results for uni-medium per-task utilization, uni-short period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 385: Results for uni-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-short height factor.

Figure 386: Results for uni-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-medium height factor.
Figure 387: Results for uni-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-tall height factor.

Figure 388: Results for uni-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 389: Results for uni-medium per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-short height factor.
Figure 390: Results for uni-medium per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-medium height factor.

Figure 391: Results for uni-medium per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-tall height factor.

Figure 392: Results for uni-medium per-task utilization, uni-long period, bimo-heavy-weight EWSS, and pipeline height factor.
Figure 393: Results for uni-heavy per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-short height factor.

Figure 394: Results for uni-heavy per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-medium height factor.

Figure 395: Results for uni-heavy per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-tall height factor.
Figure 396: Results for uni-heavy per-task utilization, uni-short period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 397: Results for uni-heavy per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-short height factor.

Figure 398: Results for uni-heavy per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-medium height factor.
Figure 399: Results for uni-heavy per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-tall height factor.

Figure 400: Results for uni-heavy per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 401: Results for uni-heavy per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-short height factor.
Figure 402: Results for *uni-heavy* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.

Figure 403: Results for *uni-heavy* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-tall* height factor.

Figure 404: Results for *uni-heavy* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *pipeline* height factor.
Figure 405: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *uni-short* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 406: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 407: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *uni-tall* height factor.

(a) Without polluter overheads.

(b) With polluter overheads.
Figure 408: Results for *bimo-light* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *pipeline* height factor.

Figure 409: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-heavy-weight* EWSS, and *uni-short* height factor.

Figure 410: Results for *bimo-light* per-task utilization, *uni-moderate* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.
Figure 411: Results for bimo-light per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-tall height factor.

Figure 412: Results for bimo-light per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 413: Results for bimo-light per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-short height factor.
Figure 414: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.

Figure 415: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-tall* height factor.

Figure 416: Results for *bimo-light* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *pipeline* height factor.
Figure 417: Results for bimo-medium per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-short height factor.

Figure 418: Results for bimo-medium per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-medium height factor.

Figure 419: Results for bimo-medium per-task utilization, uni-short period, bimo-heavy-weight EWSS, and uni-tall height factor.
Figure 420: Results for bimo-medium per-task utilization, uni-short period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 421: Results for bimo-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-short height factor.

Figure 422: Results for bimo-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-medium height factor.
Figure 423: Results for bimo-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-tall height factor.

Figure 424: Results for bimo-medium per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and pipeline height factor.

Figure 425: Results for bimo-medium per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-short height factor.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Results for Figure 426:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Without polluter overheads.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) With polluter overheads.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 426: Results for *bimo-medium* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Results for Figure 427:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Without polluter overheads.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) With polluter overheads.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 427: Results for *bimo-medium* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *uni-tall* height factor.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Results for Figure 428:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Without polluter overheads.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) With polluter overheads.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 428: Results for *bimo-medium* per-task utilization, *uni-long* period, *bimo-heavy-weight* EWSS, and *pipeline* height factor.
Figure 429: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *uni-short* height factor.

Figure 430: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.

Figure 431: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *uni-tall* height factor.
Figure 432: Results for *bimo-heavy* per-task utilization, *uni-short* period, *bimo-heavy-weight* EWSS, and *pipeline* height factor.

Figure 433: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *bimo-heavy-weight* EWSS, and *uni-short* height factor.

Figure 434: Results for *bimo-heavy* per-task utilization, *uni-moderate* period, *bimo-heavy-weight* EWSS, and *uni-medium* height factor.
(a) Without polluter overheads.

Figure 435: Results for bimo-heavy per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and uni-tall height factor.

(b) With polluter overheads.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 436: Results for bimo-heavy per-task utilization, uni-moderate period, bimo-heavy-weight EWSS, and pipeline height factor.

(a) Without polluter overheads.

(b) With polluter overheads.

Figure 437: Results for bimo-heavy per-task utilization, uni-long period, bimo-heavy-weight EWSS, and uni-short height factor.
Results for Figure 438: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-long} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-medium} height factor.

Results for Figure 439: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-long} period, \textit{bimo-heavy-weight} EWSS, and \textit{uni-tall} height factor.

Results for Figure 440: Results for \textit{bimo-heavy} per-task utilization, \textit{uni-long} period, \textit{bimo-heavy-weight} EWSS, and \textit{pipeline} height factor.