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$^{\mathrm{RT}}$, 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)

[*]Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135, with additional support from General Motors.

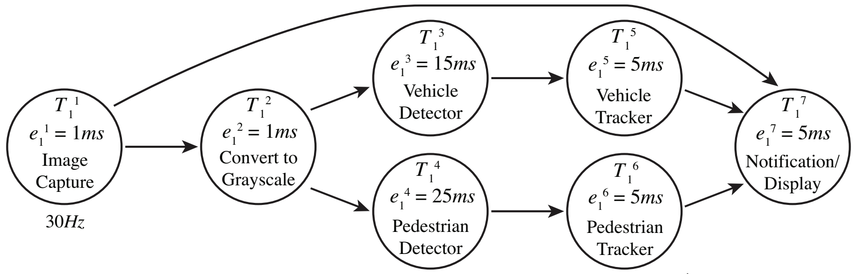


Figure 1: Vehicle and pedestrian detection and tracking component, expressed as a DAG. e_i^j denotes the execution time of task T_i^j.

scheduler, 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$^{\mathrm{RT}}$, which we run atop LITMUS$^{\mathrm{RT}}$, a Linux-based real-time operating system jointly developed by UNC and MPI.[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$^{\mathrm{RT}}$ 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$^{\mathrm{RT}}$ (Sec. 6), present our experimental results (Sec. 7), and conclude (Sec. 8).

[1]All source code is available at `www.litmus-rt.org`.

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_1^7 for display. It also passes the data to T_1^2 for computer vision processing. Vision algorithms often operate only on grayscale images, so T_1^2 strips the color components (U and V) from the image to produce a more compact grayscale image. T_1^2 then sends the grayscale image to T_1^3 and T_1^4 for vehicle and pedestrian detection, respectively. Although vehicle and pedestrian detectors may use similar algorithms, T_1^3 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_1^5 and T_1^6, 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_1^7, 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 system is comprised of a set $\tau = \{T_1, T_2, \ldots, T_n\}$ of n DAGs. Each DAG is a set $T_i = \{T_i^1, T_i^2, \ldots, T_i^{z_i}\}$ 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^ℓ to consume is denoted as $b_i^{j\to\ell}$. 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, $T_1^{4,k}$ needs input data from each of $T_1^{2,k}$ and $T_1^{3,k}$, so it must wait until those jobs complete.

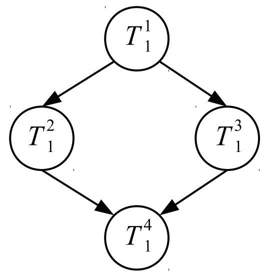


Figure 2: Precedence constraints within a DAG.

To simplify analysis, each DAG T_i has exactly one *source task* T_i^1, which only has outgoing edges, and one *sink task* $T_i^{z_i}$, 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^1 and T_i^j, and the *height* of a DAG is the depth of its sink task. For example, the depth of T_1^4 in Fig. 2 is two, and T_1^4 is the sink task so the depth of T_1 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 time units after it is released. Each task T_i^j also has a parameter e_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 τ 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 periods. 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].

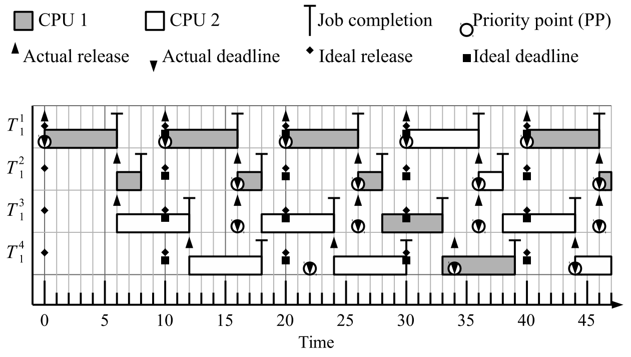
4 Determining End-to-End Latency

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.

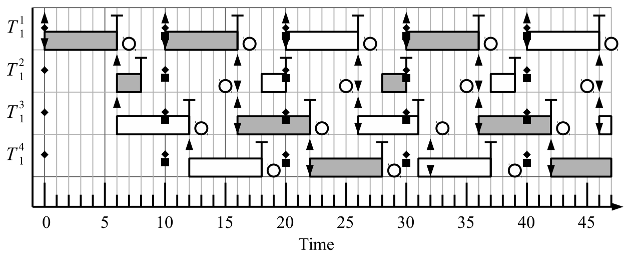
The analysis from [19] is general enough to apply to clustered algorithms wherein an intra-cluster scheduler is used from a broad class of *window-constrainted* [16] global scheduling algorithms. Under a window-constrained global algorithm, all jobs share a single run queue, and each job is prioritized 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 PPs 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. As noted above, clustered variants of these global algorithms can be obtained by using the global algorithm to schedule the tasks within each cluster. For the simple examples provided in this section, there is only one cluster, so G-EDF and G-FL are identical to C-EDF and C-FL, respectively, their clustered counterparts.

In a DAG T_i, the source task T_i^1 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 $T_1^{2,1}$ and $T_1^{3,1}$ are at time 6, when $T_1^{1,1}$ completes. Actual releases and deadlines must follow the sporadic task model. For example, consider time 33 in Fig. 3(a), when $T_1^{3,3}$ finishes early. Because all producers of $T_1^{4,3}$ are now complete, $T_1^{4,3}$ commences execution at time 33. However, because T_1^4's last actual release was at time 24, the actual release of $T_1^{4,3}$ 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



(a) System scheduled under G-EDF.



(b) System scheduled under G-FL.

Figure 3: Example schedules of the DAG depicted in Fig. 2 with $m = 2$, $p_1 = 10$, $e_1^1 = 6$, $e_1^2 = 2$, $e_1^3 = 6$, and $e_1^4 = 6$.

execute before their actual release times, as long as their actual deadlines follow the sporadic task model. Thus, if a job's ideal release precedes its actual release, and all of its precedence constraints are satisfied, then it may commence execution 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 because *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 overutilized. If Θ is the set of all tasks along the worst-case path*[2] *from T_i^1 to T_i^j, including both T_i^1 and T_i^j, then any job $T_i^{j,k}$ completes within*

$$\sum_{T_i^\ell \in \Theta} R_i^\ell$$

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, $T_i^{z_i}$ has the same ideal release as T_i^1. Therefore, the latency of the graph is simply the bound provided by Thm. 1 with $j = z_i$.

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

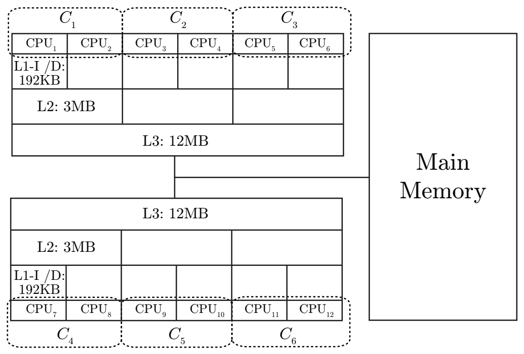


Figure 4: Example memory hierarchy. L2 clustering depicted.

Although the analysis in [19] was shown to be correct whenever a window-constrained scheduling algorithm is used within each cluster, it was only experimentally evaluated 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 towards tasks with larger e_i^j 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), $T_1^{4,2}$ finishes at time 28 instead of time 30, $T_1^{4,3}$ 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 processor 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 hierarchy, 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 finally, main memory. For simplicity, let us assume the caches are coherent, inclusive, fully associative, and follow a least-recently-used (LRU) eviction policy.[3] 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_1^1 produces 128KB and 1MB of data for T_1^2 and T_1^3, respectively, and T_1^2 and T_1^3 produce 128KB and 1MB of data respectively for T_1^4. *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_1^1 in C_1, T_1^2 in C_2, T_1^3 in C_3, and T_1^4 in C_4. With this assignment, the read costs of T_1^2 and T_1^3 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_1^4 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. $\mathcal{T}$ is the set of unassigned tasks from all DAGs, initially $\mathcal{T} = \bigcup_{T_i \in \tau} T_i$. $C = \{C_0, C_1, \ldots, C_{m/c}\}$ is the set of all clusters. $a(T_i^j)$ for each T_i^j is the cluster to which T_i^j is assigned, initially *Nil*, indicating that it has not been assigned to a cluster.

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

[3]We address the practicality of these assumptions later in this paper.

```
PICKCLUSTER($T_i^j$: task, $Q$: list of clusters)
1  $X$ := from $Q$, select clusters that minimize $w(T_i^j)$
2  $X$ := from $X$, select clusters that minimize:
        $\sum_{T_i^\ell \in cons(T_i^j)} w(T_i^\ell)$
3  $X$ := from $X$, select clusters that places $T_i^j$ closest to
        other tasks of the same graph
4  return first cluster in $min(X)$

CACHEAWAREPARTITION
1  Sort $\mathcal{T}$ by $w(T_i^j)$ in decreasing order
2  for each $T_i^j$ in the first $(m - m/c)$ ordered tasks $\in \mathcal{T}$
3      $a(T_i^j)$ := PICKCLUSTER($T_i^j, min(C)$)
4  for each remaining $T_i^j \in \mathcal{T}$
5      $Q := \{q \mid (q \in C) \wedge (u(q) + w(T_i^j) \leq h)\}$
6      if $Q \neq \emptyset$
7          $a(T_i^j)$ := PICKCLUSTER($T_i^j, Q$)
8      else
9          $a(T_i^j)$ := PICKCLUSTER($T_i^j, min(C)$)
```

Figure 5: Cache-aware task assignment heuristic.

Algorithm description. We first describe PICKCLUSTER, which is used to select the best cluster for a particular task T_i^j. 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 $w(T_i^j)$. 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^j closest to other tasks in T_i, even if those tasks are neither producers nor consumers of T_i^j. Finally, if there is still a tie, the cluster with minimum utilization is selected in line 4.

We now describe 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 PICKCLUSTER. For each remaining task (see line 4), we select in line 5 the set of all clusters for which adding T_i^j 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 PICKCLUSTER. If not, then the algorithm reverts to a worst-fit heuristic (line 9), using PICKCLUSTER to break ties.

5.2 Assessing Data Passing Costs

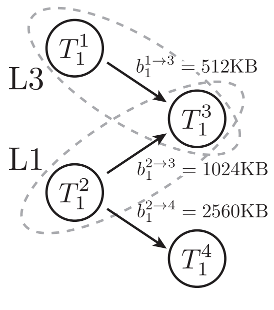
The performance of CACHEAWAREPARTITION hinges on the function $cost(T_i^j)$ used to estimate the cost of passing data between tasks. We now describe a method for estimating $cost(T_i^j)$ that assesses the costs associated with passing data between tasks on a multicore platform with a complex memory hierarchy.

The value of $cost(T_i^j)$ 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 said 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 $cost(T_i^j)$ 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 runtime 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 $cost(T_i^j)$ function *cannot* be derived by simply summing individual costs between each T_i^j and $T_i^\ell \in prod(T_i^j)$ 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 $cost(T_1^3)$ for the subgraph depicted in Fig. 6. Here, an L1-D cache is shared between T_1^2 and T_1^3. Likewise, an L3 is shared between T_1^1 and T_1^3. We will now walk through a worst-case sequence of operations that maximizes $cost(T_1^3)$.



L3
L1
T_1^1
T_1^2
T_1^3
T_1^4
$b_1^{1\to3} = 512\text{KB}$
$b_1^{2\to3} = 1024\text{KB}$
$b_1^{2\to4} = 2560\text{KB}$

Figure 6: Subgraph with data passed between nodes.

Step 1: T_1^2 writes 1024KB of data for T_1^3. 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_1^3. Although T_1^1 executes on a remote processor, a copy of this data resides in the L3 due to inclusivity.

Step 3: T_1^2 writes 2560KB of data for T_1^4. This causes evictions on two levels. First, all of T_1^3's data in the L1-D spills to the L2 after the first 192KB of T_1^4's data is written. The L1-D continues to spill as remaining data is written. Ultimately, 2368KB of T_1^4'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 store 2688KB of spilled data. T_1^3's data from Step 1 is evicted first according to the LRU policy, so 768KB of T_1^3's data is spilled from the L2 to the L3, and 256KB of it remains in the L2.

Step 4: T_1^3 reads the 512KB of data from T_1^1. 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_1^3's data from Step 1 to spill from the L2 to the L3. T_1^3's remaining 64KB of data in the L2 is self-evicted as T_1^3 reads the rest of the data from T_1^1.

Step 5: T_1^3 reads the 1024KB of data from T_1^2. All 1024KB of data from Step 1 resides in the L3 due to spills. All of T_1^3's data is read from the L3, so we charge T_1^3 for 1536KB of data read from the L3, according to the overhead dataset.

Our general algorithm to compute $cost(T_i^j)$ is outlined as follows. First, we assume all $T_i^\ell \in prod(T_i^j)$ collectively write data for T_i^j prior to data for any $T_i^k \in cons(T_i^\ell)$, and we model evictions to track the location of T_i^j's data. Next, we assume that T_i^j reads the most distant data first, and we continue to track the location of T_i^j's remaining data by modeling self-evictions. We sum overhead costs according to the location of data as it is read by T_i^j.

A remark on optimism. We have made optimistic assumptions in computing $cost(T_i^j)$: full associativity, LRU, no instruction caching, etc. Since $cost(T_i^j)$ 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, $cost(T_i^j)$ 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

PGM$^{\text{RT}}$ is a portable lightweight middleware layer we developed to manage the coordinated execution of dataflow applications. We describe the subset of PGM$^{\text{RT}}$ necessary to support the automotive workloads that we have discussed. However, PGM$^{\text{RT}}$ supports the full PGM specification; please see the online version of this paper for a full description of PGM$^{\text{RT}}$'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 PGM$^{\text{RT}}$'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 processes, use PGM$^{\mathrm{RT}}$'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$^{\mathrm{RT}}$ 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.[4] 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$^{\mathrm{RT}}$ 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$^{\mathrm{RT}}$-based version of PGM$^{\mathrm{RT}}$ used in the experiments in Sec. 7, we use LITMUS$^{\mathrm{RT}}$'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_i 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$^{\mathrm{RT}}$-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.

[4]On LITMUS$^{\mathrm{RT}}$, PGM$^{\mathrm{RT}}$ uses a low-overhead implementation of a monitor that combines spinlocks with Linux's "fast userspace mutex" interface and LITMUS$^{\mathrm{RT}}$'s special support for non-preemptive code sections.

7 Experimental Results

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

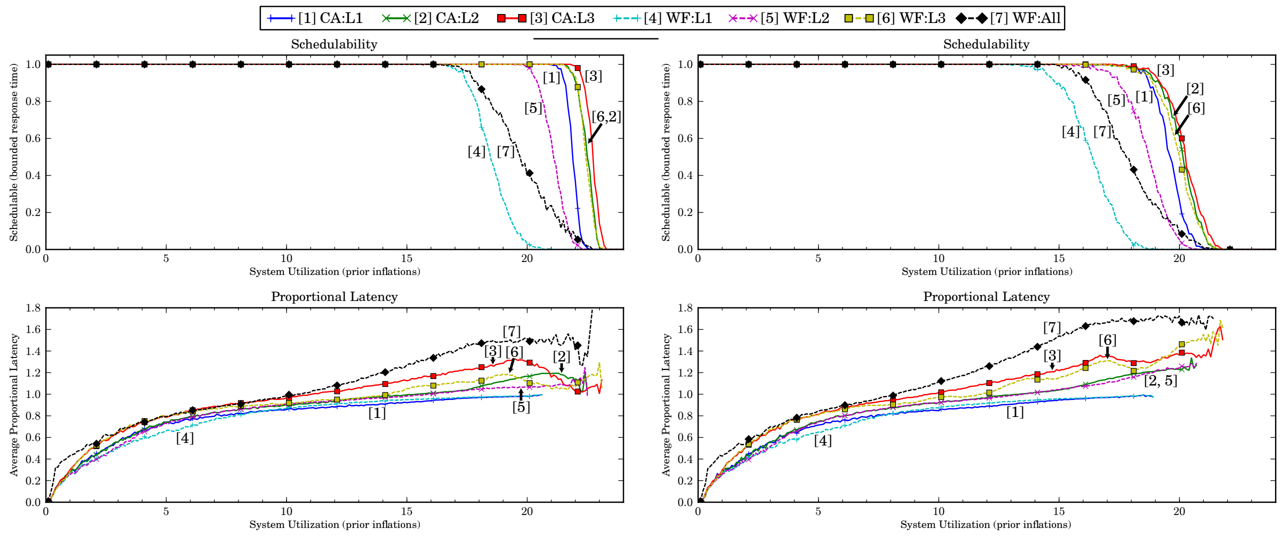
7.1 Schedulability Experiments

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$^{\mathrm{RT}}$ on LITMUS$^{\mathrm{RT}}$ 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$^{\mathrm{RT}}$. 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



(a) Without polluter overheads.

(b) With polluter overheads.

Figure 7: Schedulability of *pipeline* DAGs with *uni-long* periods, *uni-medium* task utilizations, and *medium-weight* edges.

[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 $\lfloor$height-factor $\cdot$ # of DAG nodes$\rfloor$. 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 $T_i^j = T_i^{z_i}$, 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 $p_i \cdot (\mathit{height}(T_i) + 1)$.

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 partitioning 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 ($w(T_i^j)$) 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.

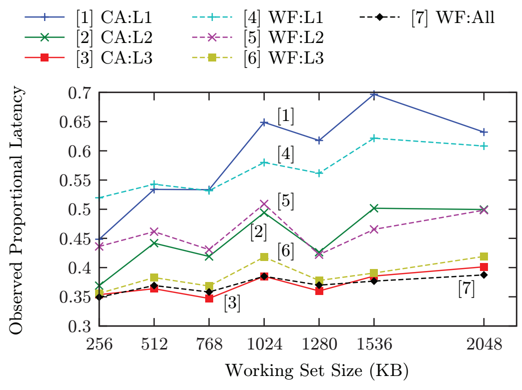
Obs. 4. There are trade-offs between schedulability and proportional end-to-end latency.

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.

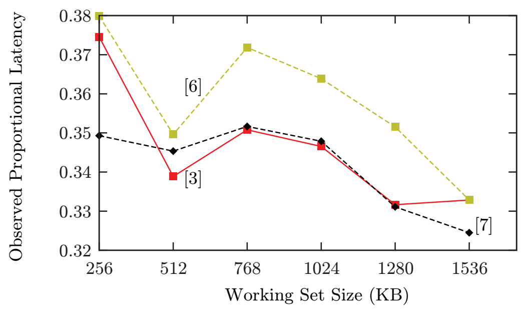
7.2 Runtime Evaluation

We now describe our runtime evaluation of PGM$^{\mathrm{RT}}$. We implemented a PGM$^{\mathrm{RT}}$ application that executes a DAG on LITMUS$^{\mathrm{RT}}$. 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.



(a) Pipelined DAGs



(b) Rectangular DAGs

Figure 8: Avg. 99*th* percentile of observed proportional latencies.

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 99*th* 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 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 PGM$^{\mathrm{RT}}$, a portable middleware for managing real-time dataflow applications. Using overheads obtained through PGM$^{\mathrm{RT}}$ running on LITMUS$^{\mathrm{RT}}$, 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

[1] AUTOSAR Release 4.1, Specification of Operating System. `http://www.autosar.org`, 2013.

[2] A. Adams, D. E. Jacobs, J. Dolson, M. Tico, K. Pulli, E. Talvala, B. Ajdin, D. Vaquero, H. Lensch, M. Horowitz, et al. The Frankencamera: an experimental platform for computational photography. *ACM Trans. on Graphics*, 2010.

[3] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *31st RTSS*, 2010.

[4] R. Bettati and J. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *12th ICDCS*, 1992.

[5] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, Univ. of North Carolina at Chapel Hill, 2011.

[6] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *26th RTSS*, 2005.

[7] G. Elliott, N. Kim, J. Erickson, C. Liu, and J. Anderson. Appendix to minimizing response times of automotive dataflows on multicore. `http://cs.unc.edu/~anderson/papers.html`, February 2014.

[8] J. Erickson and J. Anderson. Reducing tardiness under global scheduling by splitting jobs. In *25th ECRTS*, 2013.

[9] J. Erickson, J. Anderson, and B. Ward. Fair lateness scheduling: reducing maximum lateness in G-EDF-like scheduling. *Real-Time Sys.*, 50(1), 2014.

[10] S. Goddard. *On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs*. PhD thesis, Univ. of North Carolina, 1998.

[11] M. Green. "How long does it take to stop?" Methodological analysis of driver perception-brake times. *Transportation Human Factors*, 2(3), 2000.

[12] C. Hsueh and K. Lin. Scheduling real-time systems with end-to-end timing constraints using the distributed pinwheel model. *IEEE Trans. on Computers*, 50(1), 2001.

[13] Khronos Group Inc. *OpenMAX IL API Specification*, 2005.

[14] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *31st RTSS*, 2010.

[15] E. Lee and G. Messerschmitt. Synchronous data flow. *IEEE Transactions on Computers*, 75(9), 1987.

[16] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *28th RTSS*, 2007.

[17] J. Li, K. Agrawal, C. Lu, and C. Gill. Analysis of global EDF for parallel tasks. In *25th ECRTS*, pages 3–13, 2013.

[18] C. Liu and J. Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *31st RTSS*, 2010.

[19] C. Liu and J. Anderson. Supporting graph-based real-time applications in distributed systems. In *17th RTCSA*, 2011.

[20] I. Miller, M. Campbell, D. Huttenlocher, F. Kline, A. Nathan, S. Lupashin, J. Catlin, B. Schimpf, P. Moran, N. Zych, et al. Team Cornell's skynet: Robust perception and planning in an urban environment. *Journal of Field Robotics*, 25(8), 2008.

[21] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *32nd RTSS*, 2011.

[22] A. Saifullah, D. Ferry, K. Agrawal, C. Lu, and C. Gill. Real-time scheduling of parallel tasks under a general dag model. Technical Report WUCSE-2012-14, Washington University, St. Louis, USA, 2012.

[23] L. Sha and S. Sathaye. A systematic approach to designing distributed real-time systems. *Computer*, 26(9), 1993.

[24] W. Taymans, S. Baker, A. Wingo, R. Bultje, and S. Kost. *GStreamer Application Development Manual (1.2.2)*, 2013.

[25] C. Urmson, C. Baker, J. Dolan, P. Rybski, B. Salesky, W. Whittaker, D. Ferguson, and M. Darms. Autonomous driving in traffic: Boss and the urban challenge. *AI Mag.*, 30(2), 2009.

[26] J. Wei, J. Snider, J. Kim, J. Dolan, R. Rajkumar, and B. Litkouhi. Towards a viable autonomous driving research platform. In *Intelligent Vehicles Symposium*, 2013.

[27] Whitepaper. Chimera: The NVIDIA computational photography architecture. Technical report, NVIDIA, 2013.