An Experimental Evaluation of the Scalability of Real-Time Scheduling Algorithms on Large-Scale Multicore Platforms

Matthew Dellinger, Virginia Tech
Aaron Lindsay, Virginia Tech
Binoy Ravindran, Virginia Tech

We present an experimental analysis of the scalability of 13 multicore real-time scheduling algorithms on a 48-core AMD platform. The algorithms include G-EDF, P-EDF, C-EDF, and G-NP-EDF, besides others. Comparisons are made based on schedulability and tardiness. The algorithms are implemented in a real-time Linux kernel we create called ChronOS. ChronOS extends the Linux kernel's `PREEMPT_RT` patch with a flexible, scalable real-time scheduling framework. Our study shows that it is possible to implement global fixed and dynamic priority real-time scheduling algorithms which will scale to large-scale multicore platforms. Interestingly, and in contrast to the conclusions of prior research, our results reveal that some global scheduling algorithms (e.g. G-NP-EDF) are scalable on 48-core machines. In our implementation, scalability is restricted by lock contention over the global schedule and the cost of inter-processor communication, rather than the global task queue implementation. We show that algorithms implemented with scalability as a first-order goal are able to provide real-time guarantees on our 48-core platform.

Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Real-time, task scheduling, Linux, scalability, multicore processors

ACM Reference Format:
Dellinger, M., Lindsay, A., and Ravindran, B. 2011. An Experimental Evaluation of the Scalability of Real-Time Scheduling Algorithms on Large-Scale Multicore Platforms. ACM J. Exp. Algor. 17, 1, Article 1 (August 2011), 21 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The current trend among chip manufacturers is to improve performance by increasing the core count of processors, rather than increasing clock rates [Asanovic et al. 2009]. This is largely motivated by heat and power constraints [Monchiero 2006]. Currently, dual and quad-core chips are the standard. AMD is already producing a 12-core processor, which we use in a four-processor configuration in this study, while Intel has demonstrated working 48 and 80-core chips [Mattson 2010].

The increased presence of multicore and multiprocessor architectures has generated an increased interest in multiprocessor real-time task scheduling. While a significant amount of effort has been devoted to this field by the academic research community, the focus of this research has largely been on theoretical issues [Carpenter et al. 2004]. More specifically, existing research has largely concentrated on determining efficient schedulability tests for scheduling algorithms (i.e., task utilization conditions under which all task deadlines can be met), or on the development of new algorithms which can provide tighter schedulable utilization upper bounds or simpler tests.

While algorithmic research is both necessary for and beneficial to the advancement of multiprocessor real-time scheduling, it cannot solve every problem. It is not clear how system overheads (and other second order effects) affect the scalability of existing schedulers, in particular on large core count platforms. This is difficult to understand analytically and is therefore correctly scoped out of research that focuses on devising schedulability tests or producing algorithms with improved schedulability. Since task scheduling is generally implemented in the operating system kernel, and a variety of other kernel-level components have previously been shown to scale poorly on large-scale multiprocessor systems, it is doubtful that task scheduling will be immune to the detrimental effects of scaling [Boyd-Wickizer et al. 2010]. Before such algorithms can be deployed in production environment on large-scale multicore platforms, these effects must be understood.

So far, only two studies have approached this issue, and only one of them has specifically focused on it. Calandrino and Anderson performed analysis of the performance of G-EDF and P-EDF on a simulated 64-core MIPS platform and proposed Clustered EDF as a solution [Calandrino et al. 2007]. Brandenburg et. al. experimentally measured the scheduling overheads of a range of tasksets on a 32-core Sun Niagara platform and then determined schedulability using offline schedulability tests [Brandenburg et al. 2008]. However, neither of these efforts actually scheduled a large number of tasksets on a running kernel scheduler, and neither explored RMS or FIFO algorithms, which are commonly used.

Based on these observations, we performed this experimental study of the scalability of thirteen real-time scheduling algorithms on a 48-core AMD “Magny-Cours” platform. This platform uses four physical processors, each having 12 cores. On each of these processors, there are two 6MB L3 caches, each shared between six of the cores. Since this is an x86 platform, it is quite distinct from the RISC platforms used in the two previous studies.

Additionally, we performed the same measurements on a machine with a pair of 8-core AMD Opteron 6128 processors, for a total of 16 cores.

Our focus in this study is primarily on the scalability of various classes of multicore real-time scheduling algorithms. To measure this, we scheduled 288,000 tasksets for each scheduling algorithm, varying in processor utilization between 1 and 48 (1 and 16 for the 16-core machine). For each task set, we measured the schedulability and task tardiness using our scheduling test application. Additionally, we measured various system overheads in order to help us better understand the performance characteristics of our system and scheduling algorithms.

We performed our experiments using the ChronOS Linux kernel which we previously developed [Dellinger et al. 2011]. ChronOS is an extension of the Linux kernel’s `PREEMPT_RT` patch [Rostedt and Hart 2007]. The `PREEMPT_RT` patch enhances the real-time capabilities of the standard Linux kernel. ChronOS provides a framework for flexible multicore real-time scheduling, and supports a wide range of algorithms. A brief overview of its structure is given later. It is important to note that the majority of our conclusions are both implementation and platform dependent, and therefore are only valid in the context of ChronOS and the AMD platform used. We do make several conclusions which are likely generalizable to other systems, but admit that there are likely exceptions.

1.1. Multicore Real-Time Scheduling Algorithms

The most widely researched class of multiprocessor real-time scheduling algorithms are global scheduling algorithms. In global scheduling algorithms, all tasks are placed in a single globally accessible queue, and scheduling decisions are made for the entire system based on the contents of this queue. Tasks are allowed to migrate freely between all processors in the system. One advantage of this approach is that it is capable of providing optimal schedules on a multiprocessor. Another advantage is that tasks can be added to the system at run-time without difficulty. However, since tasks can migrate freely, the system incurs overhead due to migration costs and cache misses [Bertogna et al. 2009]. In our study, we consider five algorithms in this class: Global EDF (G-EDF), Global RMS (G-RMS), Global Non-Preemptive EDF (G-NP-EDF), Global Multiprocessor Utility Accrual (gMUA) [Cho 2006], Non-greedy Global Utility Accrual (NG-GUA) [Garyali 2010], Greedy Global Utility Accrual (G-GUA) [Garyali 2010], Global FIFO (G-FIFO) and SCHED_FIFO, the default real-time scheduling policy in the Linux kernel. SCHED_FIFO is not a typical global scheduling algorithm, since tasks do not reside in a single global queue, but we consider it one since its goal is to provide strict system-wide real-time priority-based scheduling, and since tasks are allowed to migrate freely.

Tasks can also be assigned to processors offline, and then uniprocessor scheduling can be performed on each processor. This approach is known as partitioning. Partitioning has several advantages over global scheduling; first, since all tasks are allocated to processors, the global scheduling problem becomes a set of local uniprocessor scheduling problems. Uniprocessor scheduling has been extensively studied and optimal and efficient algorithms are well known (e.g. EDF, RMS). Second, because the tasks are not allowed to migrate, the system incurs no overhead due to task migrations and cache misses. However, because taskset partitioning is analogous to the bin-packing problem, it is NP-hard in the strong sense [Baruah and Fisher 2005]. Because of this, partitioning cannot provide optimal scheduling on multiprocessors. Additionally, if tasks are added to the system at run-time, it may be necessary to re-partition the entire system. The four algorithms studied in this class are Partitioned EDF (P-EDF), Partitioned RMS (P-RMS), Partitioned LBESA (P-LBESA) [Northcutt 1987], and Partitioned DASA-ND (P-DASA-ND) [Clark 1990].

A variety of schemes have been studied which utilize elements of both partitioning and global scheduling. The most common of these is known as clustered or semi-partitioned scheduling [Bastoni et al. 2011]. In this scheme, tasks are first partitioned onto sets of processors, and global scheduling is then run within each cluster. This allows for some of the benefits of global scheduling, while minimizing the penalties for task migrations. In our study, we consider Clustered EDF (C-EDF) [Calandrino et al. 2007], which resides in this category.

2. CHRONOS LINUX

We use ChronOS Linux as the platform for this study. ChronOS has previously been presented in [Dellinger et al. 2011] and improvements have been made to make the platform more scalable. These improvements have been summarized in [Dellinger 2011]. A general overview is presented here to familiarize the reader with the basic concepts utilized.

ChronOS is extended from the Linux kernel's PREEMPT_RT patch, which enhances the real-time capabilities of the Linux kernel by increasing the preemptibility of the kernel. This is done by making critical sections preemptible, placing most interrupt handlers in process context, and implementing priority inheritance for in-kernel locking primitives. This results in significantly lower worst-case latencies [Dellinger 2011].

ChronOS supports the three kinds of scheduling previously described — global, clustered, and partitioned. In ChronOS, global scheduling is implemented on top of what we term "architectures", which represent sets of function pointers and data structures that manage the global queue and handle executing tasks selected for scheduling. In ChronOS, we have two primary architectures — concurrent and stop-the-world.

The concurrent architecture is so named because it allows applications to execute concurrently with the scheduler. Each processor makes a scheduling decision for itself based on the global queue, but does not interrupt the other processors unless it needs to migrate a task. In ChronOS, this architecture is used for simple algorithms, non-preemptive algorithms, and online partitioning systems. In this study, G-FIFO and G-NP-EDF are implemented on this architecture.

The stop-the-world architecture schedules in a more traditional way. It stops every processor in the system at each scheduling event, creates a schedule for all processors, and then distributes this new schedule to all processors. This allows for the implementation of significantly more complex algorithms. In this study, G-RMS, G-EDF, and C-EDF are all build on this architecture. C-EDF is performed by performing G-EDF in several distinct scheduling domains.

Because we are studying scalability, it is important to present a few details of the underlying implementations. For both architectures, the global queue is implemented as a standard Linux doubly-linked list. We are aware that this implementation has been shown to be less than ideal in [Brandenburg et al. 2008], but our measurements show that the implementation of this list is not a bottleneck on our system. This is likely due to the difference in architecture between the platform in [Brandenburg et al. 2008] and ours. Second, for both architectures, this linked list is protected by a single Linux spinlock, which is an implementation of the ticket lock algorithm. Since the global queue is the only globally accessed data structure in the concurrent architecture, this lock is the only point of contention. For the stop-the-world architecture, there is a second shared data structure — the global schedule. This must be accessed by every processor almost every scheduling event. There are a few optimizations made to decrease the contention on this lock, but they are omitted here given the space constraints. Due to the high contention, this lock is implemented as an MCS lock [Mellor-crummey and Scott 1991]. This drastically improves system-level performance over a Linux spinlock.

3. EXPERIMENTAL EVALUATION

To evaluate the performance of ChronOS, we conducted experiments on a 48-core AMD Opteron machine with a clock speed of 1.7 GHz. We measure the schedulability and mean-max tardiness of a large number of tasksets. Overheads are then measured to gain understanding into observed behaviors.

3.1. Methodology

None of the algorithms described except the forms of RMS assume any specific task arrival model. Tasks may arrive at any time in any pattern. However, to simplify experimentation and allow comparison between all algorithms, we use only periodic tasks for our experiments. This is also advantageous because computing theoretical bounds for many of the algorithms is either difficult or impossible when aperiodic tasks are allowed into the system. All deadlines are made equal to periods for the same reasons.

In order to measure the scheduling behavior of the system, we designed a test application which takes as input a taskset file, scheduling algorithm, and experiment execution time. This application makes use of the ChronOS APIs to schedule its tasks and supply their timing constraints to the kernel. The taskset file provides a period, WCET, and processor affinity for each task. Our application uses the "thread-per-task" model, which creates a single OS thread per task in the system. Each task is run though a number of jobs equal to the ceiling of experiment's runtime divided by the period of the task. Each job of the task burns the processor for the task's WCET and then the thread sleeps until the beginning of the next job. The processor is burned by incrementing a counter a precomputed number of times. The number of times this counter must be incremented to burn 1μs of time is called the system slope. This means that our workloads are almost completely processor-intensive.

3.2. Baker Tasksets

We use a variation of the method described by Baker to create a set of tasksets which evaluate the scheduler's performance under a variety of circumstances [Baker 2005a]. In this method, we generate a large number of tasks according to a set of task weightings and statistical distributions. In this variation, which was first described in [Brandenburg et al. 2008], we use three different weightings of two statistical distributions — a uniform and a bimodal distribution. This gives us a total of six taskset distributions. As was done in [Brandenburg et al. 2008], we distribute all periods uniformly between $10ms$ and $100ms$. Tasks in the three uniform distributions were distributed over [0.001, 0.1], [0.1, 0.4], and [0.5, 0.9]. Tasks in the three bimodal distributions were distributed uniformly over [0.001, 0.5) or [0.5, 0.9] with probabilities of 8/9 and 1/9, 6/9 and 3/9, and 4/9 and 5/9. These six distributions are referred to as light uniform (BLU), medium uniform (BMU), heavy uniform (BHU), light bimodal (BLB), medium bimodal (BMB), and heavy bimodal (BHB).

Tasksets were generated for each integer utilization point on the interval (1, 48). For each taskset, tasks were added until the utilization demand exceeded the desired utilization, and then the last task was removed. 1000 tasksets were generated for each utilization point in each distribution, resulting in a final count of 288,000 tasksets.

For each algorithm, a total of 1,031,707,071 jobs of 38,779,473 tasks were scheduled. This is by no means the largest sample size ever used to analyze real-time schedulers; [Brandenburg and Anderson 2009] and [Brandenburg et al. 2008] present results using 5.5 and 8.5 million tasksets. However, both of these works rely on using a small number of tasksets to characterize the overheads in a system, and then perform offline schedulability tests on the full set of tasksets which have been inflated with these overheads. In contrast, we *schedule* the experimental workload generated and measure schedulability and tardiness. Our experiment is, to our knowledge, the largest sample size ever experimentally tested.

3.3. Partitioning and Clustering

Tasksets were partitioned offline using one of two algorithms: a first fit method similar to the method devised by Baruah and Fisher [Baruah and Fisher 2005], and a simple least-utilization algorithm. The first-fit algorithm was used for P-EDF. It is based on sufficient EDF schedulability criteria for sporadic tasks. Since we are only using periodic tasks, we can safely ignore the request-bound function constraint of the algorithm, and use only the utilization constraint. Tasks are assigned to the first processor with a utilization low enough that the sum of the utilizations of all tasks previously assigned to the processor and the current task is less than 1.

We make two changes to the original algorithm. First, some tasksets which are feasible under algorithms like PFair [Srinivasan and Anderson 2006] and LLREF [Cho et al. 2006] cannot be partitioned by the algorithm in [Baruah and Fisher 2005]. For example, consider a dual-core machine and a taskset with three tasks, each having a utilization of 0.6. As the tasks have a cumulative utilization of 1.8, the taskset is feasible under both PFair and LLREF, but cannot be partitioned by Baruah's algorithm. We commonly see such cases in our heavy-uniform distributions. In such cases, we employ a best-effort partitioning approach. If a task does not fit on any processor, the task is assigned to the processor with the lowest total utilization.

Second, Baruah and Fisher assume that the algorithm is applied to the tasks in non-decreasing deadline order. However, since periods are equal to deadlines, for our lighter distributions, this ordering often results in a single processor having a large number of tasks with short periods, while another processor has a large number of tasks with large periods. To deal with this, we assign tasks in order from highest utilization to lowest utilization (i.e. $U_i \geq U_{i+1}$).

Since task execution times are not inflated to deal with system overheads, assigning a load of 1 to a given processor would place it slightly into overload. To deal with this, we replace the fit bound of 1 with 0.95, leaving a small amount of room for overhead.

The least-utilization algorithm was used for P-RMS. Under this algorithm, tasks were considered in decreasing utilization order, and each task was assigned to the core with the least total utilization. This guarantees that no core will receive a second task until a task has been assigned to each core, and attempts to evenly distribute the load among all cores.

Clustering is handled by partitioning the taskset, and then grouping all tasks assigned to a set of cores together. For example, in a system with two quad-core processors and two clusters, we partition the taskset and then place all tasks assigned to cores 0 through 3 on cluster A and all tasks assigned to cores 4 through 7 on cluster B. Clusters are selected to correspond to physical processors, creating clusters of 12 cores. Since the first-fit algorithm tends to group all tasks on small set of cores, we use the least-utilization partitioning algorithm for creating clustered tasksets.

3.4. Scheduling Results

Hard real-time schedulability results for the 48-core system are shown in Figure 1. We measured the percentage of the tasksets executed that were successfully scheduled as we varied the utilization from 1 to 48. The first column shows the three weightings in the uniform distributions, while the second column shows the three weightings in the bimodal distributions.

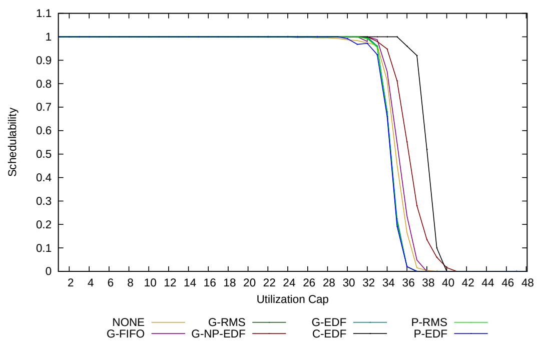
There are several behaviors worth noting in our results. First, P-EDF outperforms all other algorithms except in the heavy uniform case, where both of the partitioned algorithms perform the worst. This occurs because partitioning each taskset into feasible sub-tasksets becomes impossible for this distribution due to the high weight of many of the tasks, meaning that some of the tasks must be migrated in any feasible schedule. This is a well-known problem with partitioning [Baker 2005b]. As expected, P-RMS does not perform as well as P-EDF. However, ignoring the heavy uniform case, both perform well with respect to the other algorithms.

Second, C-EDF performs well in all cases. In no case does it fail to schedule all tasksets below a load of 26. It provides the best performance in the heavy uniform case, and the second best performance behind P-EDF in the heavy bimodal and light uniform cases.

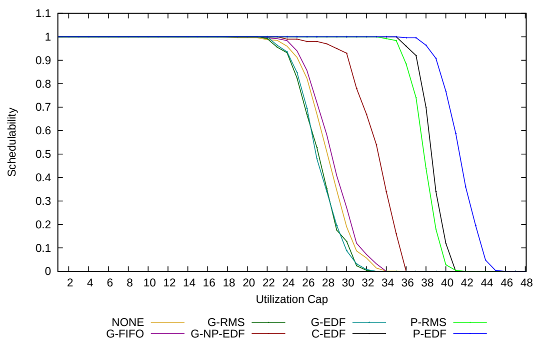
Third, G-NP-EDF routinely performs reasonably well, never failing to schedule all tasksets below a load of 18. Its theoretical performance is not as strong as C-EDF, P-EDF, or G-EDF, and as expected of a non-preemptive algorithm, its performance is weakest on the bimodal distributions. Based on its performance in the light uniform case, which is the most demanding from the perspective of overheads, we conclude that it scales well.

Fourth, G-FIFO outperforms `SCHED_FIFO` in every case, even though `SCHED_FIFO` approximates G-FIFO. This is a highly meaningful result, because `SCHED_FIFO` mostly relies on per-CPU data structures to prevent inter-processor memory contention and cache-thrashing. This approach should, in theory, provide a higher degree of scalability.

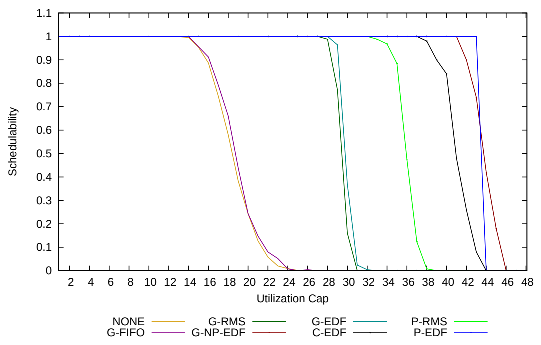
Last, G-EDF and G-RMS perform poorly compared to the other algorithms in the BHU, BHB, BMB, and BLB cases. However, in the light and medium uniform



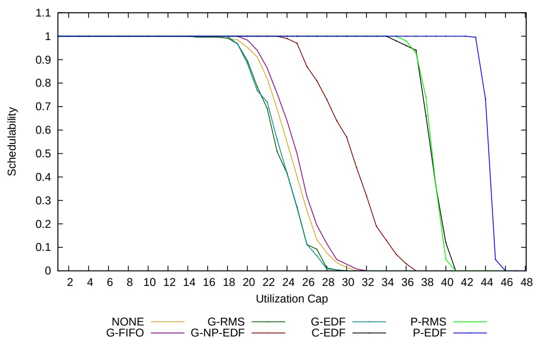
(a)



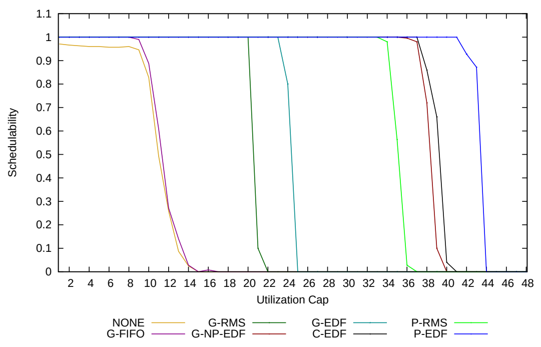
(b)



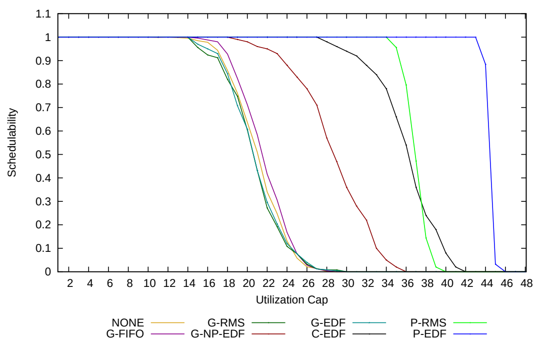
(c)



(d)



(e)



(f)

Fig. 1. Hard real-time schedulability results for (a) heavy uniform, (b) heavy bimodal, (c) medium uniform, (d) medium bimodal, (e) light uniform, and (f) light bimodal per-task weight distributions.

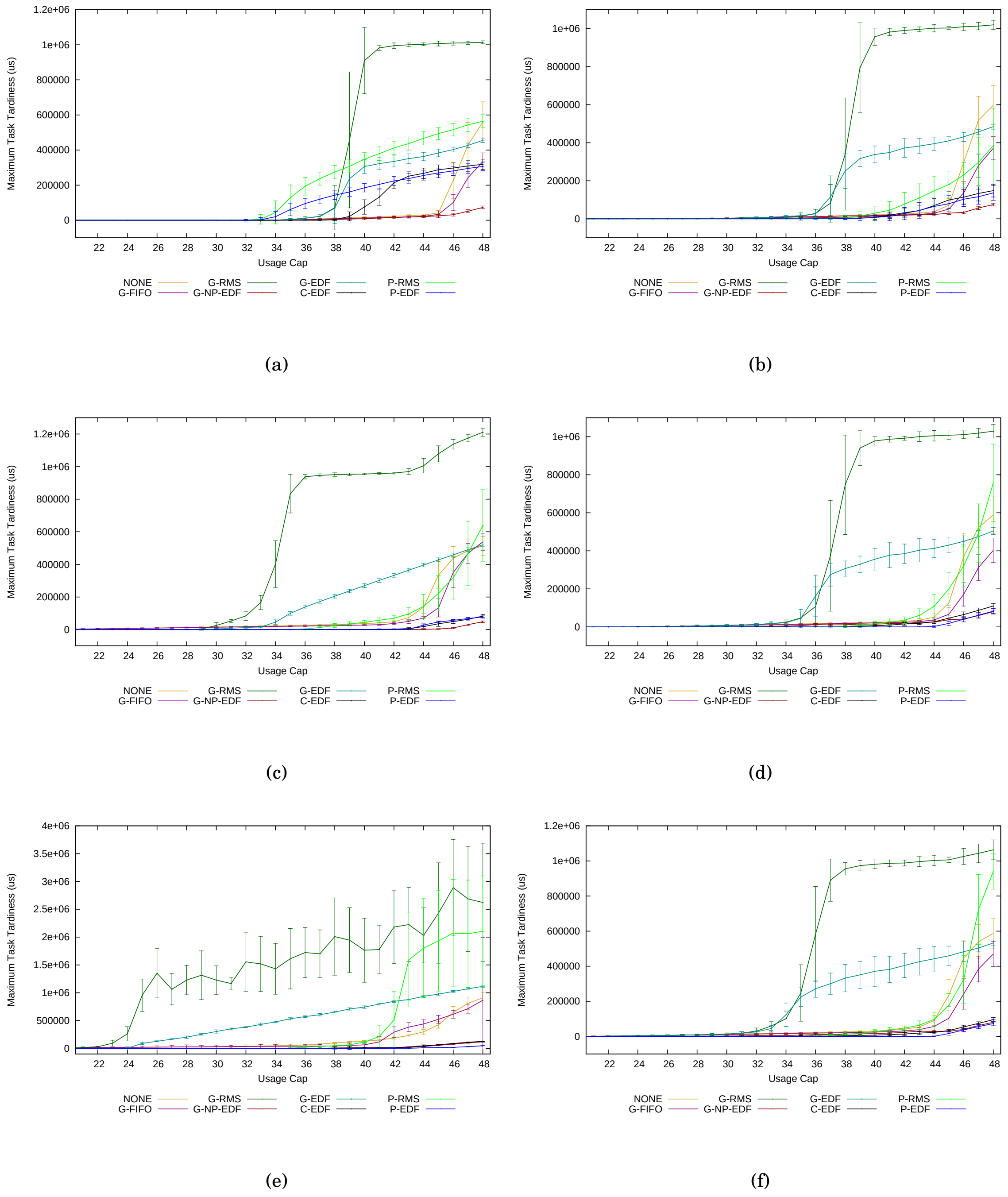
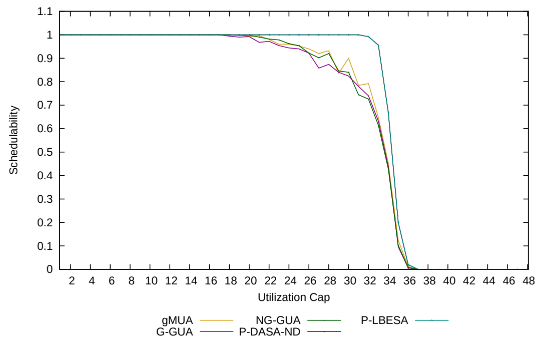
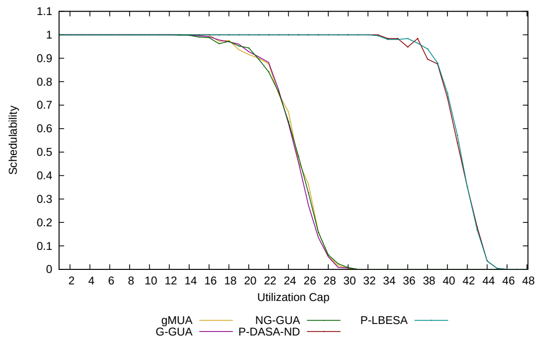


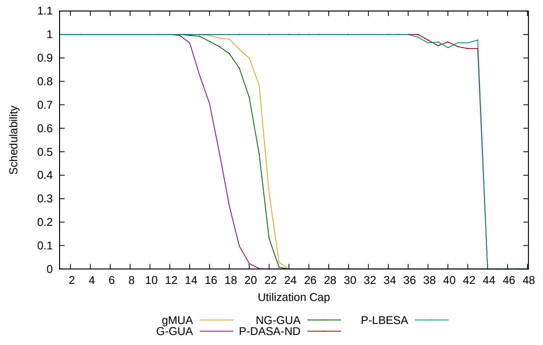
Fig. 2. Mean maximum tardiness results for (a) heavy uniform, (b) heavy bimodal, (c) medium uniform, (d) medium bimodal, (e) light uniform, and (f) light bimodal per-task weight distributions. (Please note that the x-axis begins at a load of 20.)



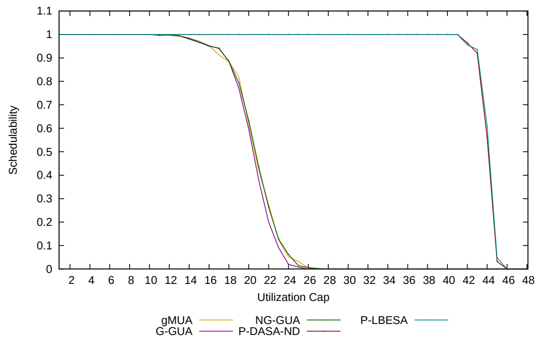
(a)



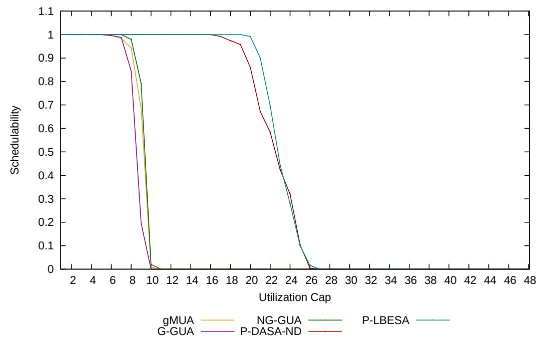
(b)



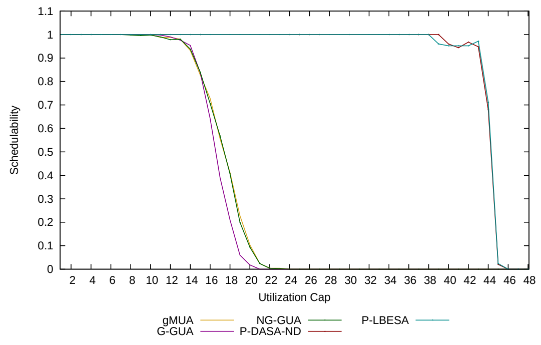
(c)



(d)



(e)



(f)

Fig. 3. Hard real-time schedulability results for TUF schedulers for (a) heavy uniform, (b) heavy bimodal, (c) medium uniform, (d) medium bimodal, (e) light uniform, and (f) light bimodal per-task weight distributions.

distributions, the performance of both algorithms improves significantly over that of G-FIFO. The performance of G-EDF in the medium uniform case is above its theoretical bound as computed by the GFB method [Goossens et al. 2003]. In all cases, G-EDF outperforms G-RMS, which is to be expected.

Mean-maximum tardiness (MMT) results are presented in Figure 2. MMT was computed by recording the maximum tardiness in each taskset and taking the average across all tasksets. Bounded tardiness is often considered a sufficient test for soft real-time schedulability. Rather than comparing our results to theoretically computed tardiness bounds, which would take significant effort to accurately derive for our platform, we look at each algorithm to determine the point where tardiness begins to rise rapidly. For the RMS-based algorithms, this value is quite distinct due to their fixed-priority nature, while for the other algorithms it is not so distinct. Since G-FIFO (and therefore `SCHED_FIFO`), G-EDF, G-NP-EDF, C-EDF, and P-EDF should all bound tardiness in underload, this value also tells us approximately when the system enters overload due to system overheads.

We observe several important trends in these results. First, G-NP-EDF, C-EDF, and P-EDF consistently provide the lowest tardiness in the system. The only exception to this is the heavy uniform case, where C-EDF and P-EDF enter overload earlier than several of the global algorithms due to the partitioning difficulties with heavy tasksets. Second, as expected, the RMS-based algorithms show high tardiness, since one task in the system receives interference from all other tasks. Third, both G-FIFO and `SCHED_FIFO` provide low tardiness until the system enters overload due to scheduling overheads, at which point the tardiness of both rapidly increases. Once this happens, G-FIFO consistently provides lower tardiness than `SCHED_FIFO`. As expected, G-EDF bounds tardiness up to a point, and then in most cases shows a jump to around 300ms followed by a linear increase. However, this jump occurs significantly lower than 48 for all cases, implying that G-EDF is not scaling as well as G-NP-EDF or G-FIFO.

Table I. Load at which tardiness-bounding algorithms enter overload

Algorithm	BHB	BHU	BMB	BMU	BLB	BLU
SCHED_FIFO	44	45	44	43	43	36
G-FIFO	44	45	44	44	44	40
G-EDF	36	37	35	33	33	24
G-NP-EDF	46	46	46	46	46	42
C-EDF	39	39	44	43	44	42
P-EDF	39	31	44	43	44	46

Table I shows the values at which each of the algorithms which bound tardiness in underloads enter overload. From this, we can see two things. First, the scheduling algorithms built on the concurrent architecture (G-FIFO and G-NP-EDF) scale reasonably well. Second, G-EDF's scaling is almost directly proportional to the number of tasks. Third, as expected, partitioned algorithms have difficulties with heavy tasksets, and perform well with lighter ones.

3.4.1. Scheduling Results on 16 cores. We also conducted these experiments on a system with 16-cores. The results are shown in Appendix A in Figures 9 to 11.

Most algorithms perform as expected; G-RMS, G-EDF, P-EDF, and P-RMS all achieve their theoretical bounds bounds for all cases. C-EDF outperforms G-EDF in all cases, and both outperform all other global algorithms for the BHB, BHU, BMB, and BLB task sets. G-FIFO and G-NP-HVDF show widely varying performance, and are only able to meet all deadlines consistently under low loads. P-EDF shows performance which keeping with the difficulties of the bin-packing problem involved; the load at which it is able to successfully schedule all tasksets is proportional to the average task weight, and ranges from 10 for the BHU case to 15 for the BLU case. Also, as expected, P-EDF performs much better under the bimodal cases than the global deadline-based algorithms, but worse for the BHU case. For the BHB, BMB, and BLB cases P-EDF schedules all tasksets up to loads of 12, 13, and 14 respectively, while G-EDF only manages 8, 8, and 7. However, G-EDF is able to schedule all tasksets up to a load of 11 for the BHU case, compared to 10 for P-EDF. These difference are not due to scaling problems, but rather are the expected behaviors of the algorithms. G-NP-EDF demonstrates high but unpredictable performance; it is significantly outperformed by G-EDF, C-EDF, and G-RMS the BHB, BHU, and BMB cases. However, it outperforms G-RMS and G-EDF at some loads in the BLB case, outperforms both for all loads and C-EDF for some loads in the BMU case, and outperforms all three for the BLU case.

3.4.2. TUF Scheduling Results. In addition to the more traditional global and partitioned schedulers, we studied time/utility function-based schedulers as well. The studied schedulers included gMUA, G-GUA, NG-GUA, P-DASA-ND, and P-LBESA. As before, the 48-core schedulability plots are included inline in Figure 3, while the 16-core plots are shown in Appendix A.

On the 16-core platform, NG-GUA and gMUA provide similar performance. However, they suffer significant performance degradation due to their high overheads on large systems, and only perform near G-EDF on heavier tasksets. Under the BLU tasksets, they suffer a catastrophic failure, and meet deadlines only up to around half the load of G-EDF. Furthermore, gMUA consistently outperforms NG-GUA by a small margin; in most cases, it appears to be able to decay around a load of 2 later than NG-GUA. This is consistent with the effects of the overhead difference between the two algorithms.

None of the global utility accrual algorithms are able to provide full schedulability at a load greater than 19 for the 48-core experiments. For the BLU case, none can provide it at a load higher than 7. In this case, NG-GUA and gMUA miss their theoretical bound by over 500%. The large number of tasks in the BLU case also affects P-LBESA and P-DASA-ND, as neither is able to provide full schedulability for a load over 19. On the surface, this result is surprising, because the average task weight is no larger than that of the BLU case for the 16-core platform, and so the average number of tasks per core should be the same. However, since a first-fit partition is used, it is likely that a large number tasks allowed many extremely lightweight tasks to be assigned to the first several cores in some cases. This degradation must be attributed to the overhead of the algorithms, since no such effect occurs under P-EDF, which uses the same tasksets. P-DASA-ND fails to schedule tasksets at a lower load than P-LBESA because P-LBESA takes the optimistic approach of placing all tasks in the schedule, and then removing them until the schedule is feasible while P-DASA-ND takes the pessimistic approach of adding tasks to an empty schedule until it becomes infeasible. When in underload, there is always a feasible schedule, and so P-LBESA's approach will result in significantly lower overheads.

3.5. Overheads

To thoroughly understand these results, we must understand various sources of overhead in the system. To accomplish this, we measure a variety of overheads to determine their effects.

All of our measurements are taken by using the x86 `rdtsc` instruction. This instruction reads the processor's time-stamp counter, and is a common feature on all x86 processors manufactured in the last decade. When paired with an `mfence` instruction

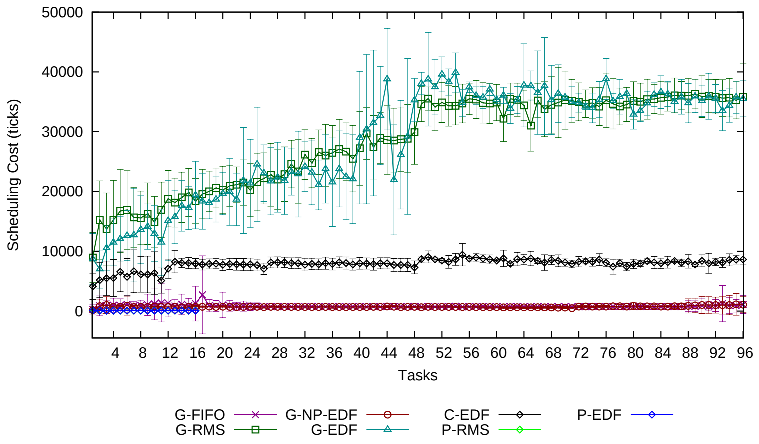


Fig. 4. Scheduling overheads. (Note that G-FIFO and G-NP-EDF are on top of each other, as are P-RMS and P-EDF.)

to prevent re-ordering, it provides single-cycle resolution and allows for the fine-grain measurements we need.

3.5.1. Scheduling Overheads. The most obvious source of overhead in real-time scheduling is the time cost of performing the scheduling itself. The scheduler must be invoked at every scheduling event, so its cost is of prime importance.

To measure scheduling overhead, we instrument the scheduler to record a timestamp before and after our scheduling algorithm is called. Additionally, we record the number of tasks in the scheduler. For each global scheduler, we graph the scheduling overhead with respect to the number of tasks. To generate the data, we ran one full-load taskset for each taskset distribution. On average, each data point is the result of 158 readings. The scheduling overheads are shown in Figure 4. Note that for partitioned algorithms, we only measure up to 16 tasks because the partitioning means that no core will schedule as many tasks as with global scheduling.

The results make several things clear. First, G-FIFO, G-NP-EDF, P-EDF, and P-RMS all schedule in $O(1)$ time, as expected. G-EDF, C-EDF, and G-RMS all schedule in $O(m)$ time, and their performance when there are less than m tasks in the system is linear. This is also in line with their expected performance. Both G-EDF and G-RMS exhibit high scheduling overheads due to the cost of accessing m task descriptors, most of which are not cache-hot. This overhead drastically decreases with C-EDF, which executes the same code as G-EDF, but does so on only a quarter of the processors, and accesses only local task descriptors.

3.5.2. Migration Overheads. Another source of system overhead is due to cache misses after a task is preempted or migrated. When one task is preempted and another task begins execution, some of the first task's data may be removed from the processor's cache. When the first task resumes execution, accessing this data will incur a cache miss. Similarly, when a task is migrated between two processors, it is likely that its data is not cache-hot on the task's new processor. Furthermore, some of the data will likely be cache-hot on the tasks's previous processor, which means that if the task changes data, a cache-invalidate message must be sent to the previous processor. Additionally, if the processors between which the task is migrated do not share memory, fetching the data into the second processor's cache may require additional overhead.

Since our test application performs most of its execution in a simple burn loop, its working set is quite small, and therefore it cannot be instrumented to capture these overheads. Instead, we create a separate test to measure them. This test works as follows; first, a working set of some i pages is allocated. The thread running the test is then locked to a core, and the buffer is initialized. The thread then writes data into j evenly spaced addresses within each page of the buffer, and records the time it took to perform all the writes. This is done 1000 times. The thread then initializes the buffer from some core P_A, and then migrates itself to some other core P_B, so that its data is cache-cold. Once executing on P_B, the thread then performs the same set of writes as before, and again measures the time taken. This is also done 1000 times. The difference between the times is the cost of the cache misses. Measurements are performed for working sets of 1, 2, 4, 8, 16, 32, and 64 pages.

Figure 5 shows the cost of four different migration paths. First, we measure the cost of migrating between two cores which share L3 cache, but not L2 or L1. Second, on our platform, each processor shares memory, but has two separate L3 caches, each for half of the processor's cores. Therefore, we test migrating between cores on the same processor which do not share L3 cache. Third, we measure the cost to migrate to a different processor, thereby loosing direct access to memory. The third and fourth paths are migrating from processor 0 to processor 1 and from processor 0 to processor 2. Migrations between processors 0 and 3 are not shown, because they were measured to have the same overhead as migrating between 0 and 1. On our system, a page is 4096 bytes, and a cache line is 64 bytes. Therefore, the maximum number of writes per page we can use without duplicating writes on a given cache line is 64. Additionally, AMD implements sequential cache line prefetching. In order to avoid inaccuracies from this, we measure with 4 and 16 writes-per-page, or every 4 and 16 cache lines. All of our tests use CPU 0 as the first CPU.

From these results, we can clearly see several things. First, there is a measurable difference in the various migration paths. In fact, the overhead to migrate a task between two physically distinct processors is around four times the cost to migrate within an L3. While this does not significantly impact our scheduling results, it could be a significant performance hit to a highly memory-intensive application. Second, we see

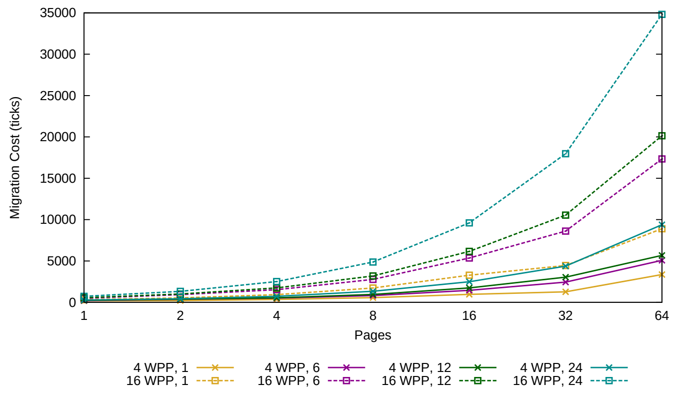
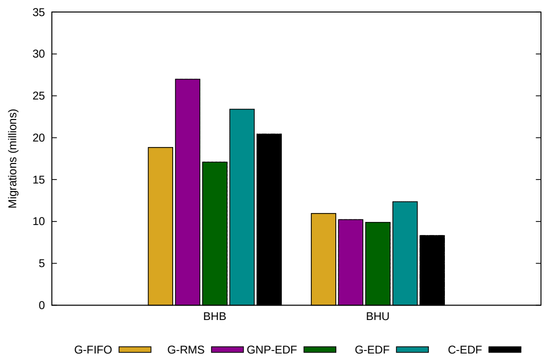
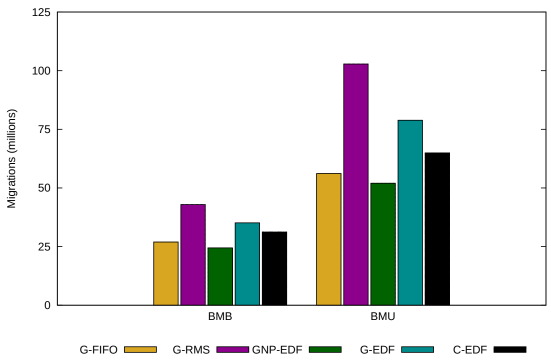


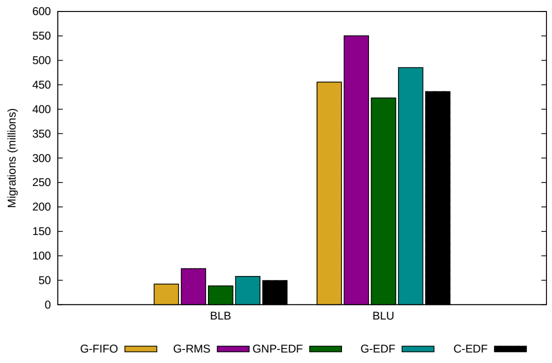
Fig. 5. Cache-miss costs of four migration paths



(a)



(b)



(c)

Fig. 6. Migration counts for (a) heavy (b) medium and (c) light tasksets

that, as expected, memory access times are significantly shorter when accessing the calling processor's memory. Calling processor 1's memory from processor 2 or 4 incurs around a 15% overhead, while accessing processor 1's memory from processor 3 doubled the cost. We ran several further tests, and found that this effect also happened when fetching between processors 2 and 4.

The second aspect of task migration which needs to be measured is the cost of performing the actual migration in the scheduler. Table II shows the average cost of migrations for each scheduling architecture. At least 4000 data points were collected for each measurement. Since the same migration function is used by all the schedulers, migration costs are nearly identical under all algorithms which share an architecture.

Table II. Migration cost for both scheduling architectures(ticks)

Platform	Concurrent	Stop-The-World
Average	4516	14866
St. Dev.	1423	8401
Worst	48459	38867

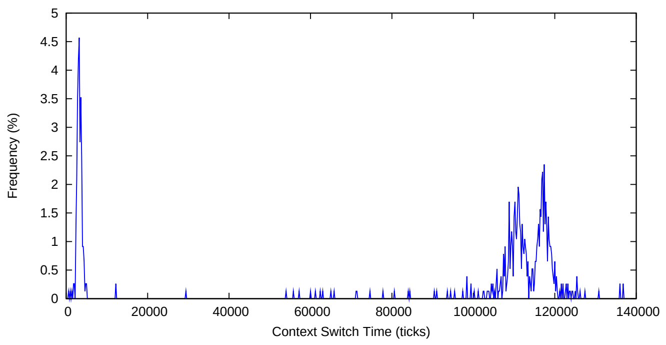


Fig. 7. Context switch times

The difference between the two architectures is mainly due to lock contention for the per-core `runqueue`. When core A wants to pull a task from core B, it must first lock B's `runqueue`. If B is currently in the scheduler, it will have its own `runqueue` locked, so A must wait until B finishes scheduling. In the stop-the-world architecture, it is likely that the core being pulled from is currently executing in the scheduler, so a core must often block. However, the number of migrations is minimized, so it is unlikely that migrations will interfere with each other. In the concurrent architecture, it is unlikely that the target core is in the scheduler, but it is also possible that several migrations are interfering with each other. Hence, the average time is lower for the concurrent architecture, but the worst time is significantly higher.

The third factor needed to understand migration patterns is the number of migrations each scheduling algorithm performs. These numbers are shown in Figure 6.

3.5.3. Context Switch Overheads. This cache miss overhead manifests itself not only in user space task execution times and migration overheads, but also in the time required to context switch to a new task. Figure 7 shows a histogram plot of the context switch time measured. Our platform shows three peaks, representing local migration and two of the possible migration paths previously discussed. Clearly, the cache misses associated with context switching to a migrated task are quite costly.

3.5.4. System Call Overheads. There are two system calls which are highly important to ChronOS: `begin_rt_seg()` and `end_rt_seg()`. Each of these calls must be made by each job, and therefore, the sum of their execution times represents the minimum possible segment length. Figure 8 shows the overheads of various system calls, including `begin_rt_seg()` and `end_rt_seg()`. Traditional system calls `gettid()` and `clock_getres()` are relatively short, and therefore provide reasonable baseline estimates of the overhead of a system call. Both ChronOS system calls are quite long, but not inordinately so when compared to `sched_setaffinity()` and `sched_setscheduler()`, both of which also potentially invoke scheduling changes. In fact, both ChronOS system calls perform the same underlying operations as `sched_setscheduler()`, and therefore their high cost is completely reasonable.

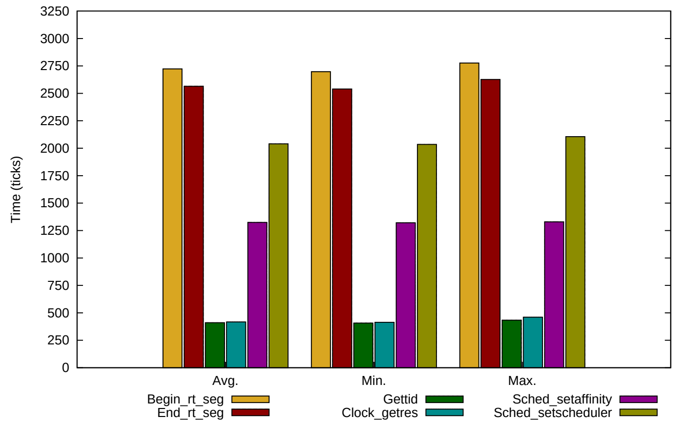


Fig. 8. Execution costs of `begin_rt_seg()`, `end_rt_seg()`, and several other common system calls

4. CONCLUSIONS

We have presented an experimental evaluation of the scalability of thirteen multicore real-time scheduling algorithms on a 48-core platform. While this is not the first study to address scalability, our work is the first to *schedule* the experimental workload generated and measure schedulability and tardiness, and the first to use a x86 platform.

We make several conclusions. First, G-NP-EDF can be implemented in a highly scalable manner, and provides extremely good performance in the soft real-time case. Second, when it is possible to produce a feasible schedule using partitioning or clustering, P-EDF and C-EDF generally outperform other algorithms in the hard real-time case. We find that the scaling bottleneck for G-EDF and G-RMS is the high execution cost of the scheduler and the cost of inter-processor synchronization necessary to distribute the global schedule. The execution cost of the schedulers is largely due to the cost of accessing task descriptors on remote processors. Likely because of our use of an x86 platform and in contrast to [Brandenburg et al. 2008], we find that even a simple linked list implementation of the global queue does not become a bottleneck on 48 cores.

We intend to pursue three directions for future work. First, although we did not find implementing the global queue as a linked list to be a bottleneck, if the core count is further increased it will likely become one. To this end, for future studies we intend to replace this with a more advanced data structure. Second, both G-EDF and G-RMS suffer from the high cost and frequency of inter-processor synchronization during global scheduling events. While the use of an MCS lock significantly improved performance over the use of a Linux spinlock, we believe that there are still a number of improvements to be made through the use of some form of read-write lock and careful consideration of when synchronization can be avoided altogether. Third, our results directly contradict the previous study on this subject, which we believe is due to the differences between the SPARC and x86 architectures. We would like to repeat this study on a RISC platform to see which of our conclusions are specific to the x86 architecture, and which may be generalized.

The development of ChronOS Linux is coordinated via git repositories at `http://git.chronoslinux.org`, and the latest releases are published at `http://chronoslinux.org`, along with installation and usage instructions.

REFERENCES

ASANOVIC, K., BODIK, R., ET AL. 2009. A view of the parallel computing landscape. *Commun. ACM 52*, 56–67.

BAKER, T. 2005a. A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical Report TR-051101, Florida State University.

BAKER, T. P. 2005b. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. Tech. rep., Florida State University.

BARUAH, S. AND FISHER, N. 2005. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS '05*. 321–329.

BASTONI, A., BRANDENBURG, B. B., AND ANDERSON, J. H. 2011. Is semi-partitioned scheduling practical? In *ECRTS '11*. to appear.

BERTOGNA, M., CIRINEI, M., AND LIPARI, G. 2009. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. Parallel Distrib. Syst. 20*, 553–566.

BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. 2010. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. 1–8.

BRANDENBURG, B. B. AND ANDERSON, J. H. 2009. On the implementation of global real-time schedulers. In *RTSS '09*. 214–224.

BRANDENBURG, B. B., CALANDRINO, J. M., AND ANDERSON, J. H. 2008. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *RTSS '08*. 157–169.

CALANDRINO, J. M., ANDERSON, J. H., AND BAUMBERGER, D. P. 2007. A hybrid real-time scheduling approach for large-scale multicore platforms. *ECTRS '07 0*, 247–258.

CARPENTER, J., FUNK, S., HOLMAN, P., SRINIVASAN, A., ANDERSON, J., AND BARUAH, S. 2004. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca.

CHO, H. 2006. Utility accrual real-time scheduling and synchronization on single and multiprocessors: Models, algorithms, and tradeoffs. Ph.D. thesis, Virginia Tech.

CHO, H., RAVINDRAN, B., AND JENSEN, E. D. 2006. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *RTSS '06*. 101–110.

CLARK, R. K. 1990. Scheduling dependent real-time activities. Ph.D. thesis, CMU. CMU-CS-90-155.

DELLINGER, M., GARYALI, P., AND RAVINDRAN, B. 2011. ChronOS Linux: A best-effort, real-time multiprocessor Linux kernel. In *ACM DAC '11*.

DELLINGER, M. A. 2011. An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms. M.S. thesis, Virginia Tech. http://www.real-time.ece.vt.edu/dellinger-thesis11.pdf.

GARYALI, P. 2010. On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors. M.S. thesis, Virginia Tech.

GOOSSENS, J., FUNK, S., AND BARUAH, S. 2003. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst. 25*, 187–205.

MATTSON, T. 2010. The future of many core computing: A tale of two processors.

MELLOR-CRUMMEY, J. M. AND SCOTT, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 21–65.

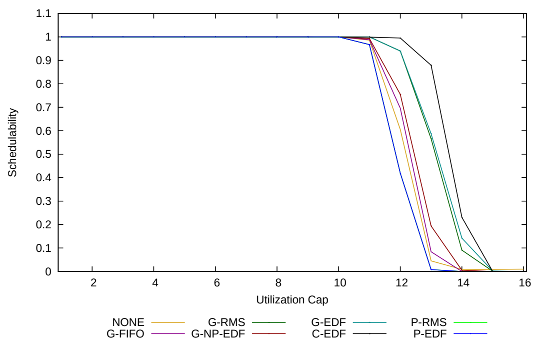
MONCHIERO, M. 2006. Design space exploration for multicore architectures: A power/performance/thermal view. In *In Proc. of 20th ICS*. 177–186.

NORTHCUTT, J. D. 1987. *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Academic Press.

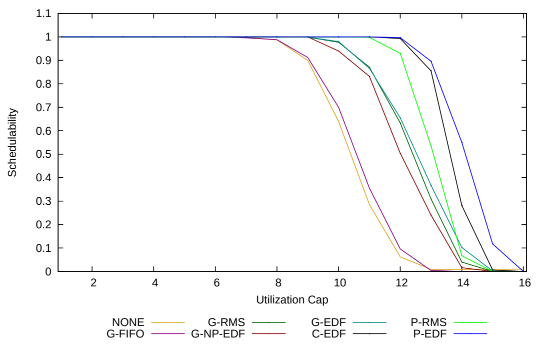
ROSTEDT, S. AND HART, D. V. 2007. Internals of the RT patch. In *Proceedings of the Linux Symposium*. Vol. 2. 161–172.

SRINIVASAN, A. AND ANDERSON, J. H. 2006. Optimal rate-based scheduling on multiprocessors. *J. Comput. Syst. Sci. 72*, 1094–1117.

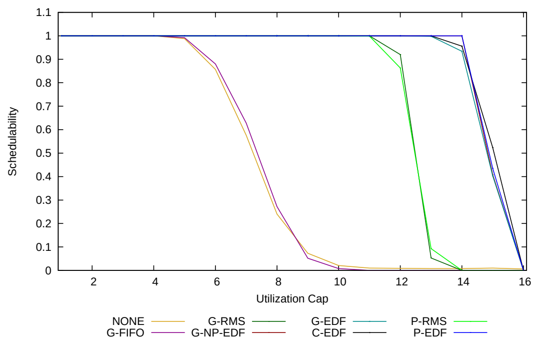
A. 16-CORE RESULTS



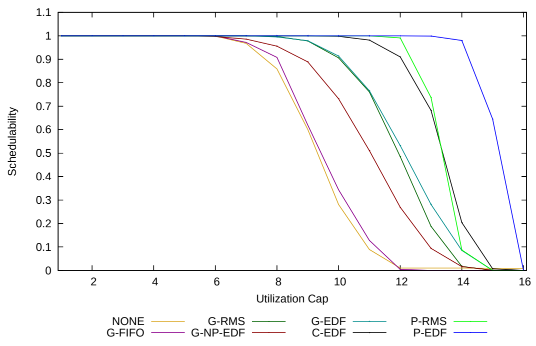
(a)



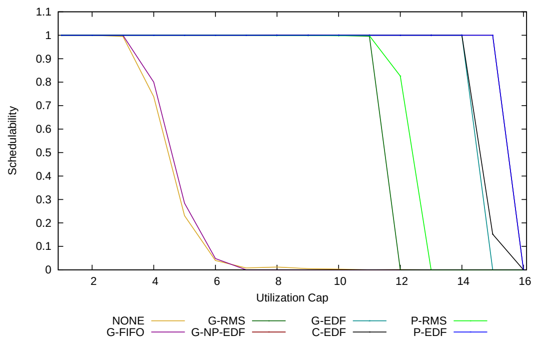
(b)



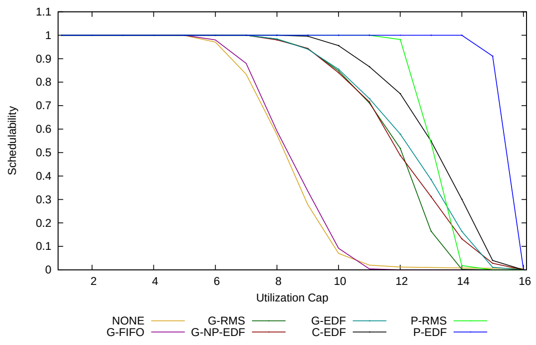
(c)



(d)

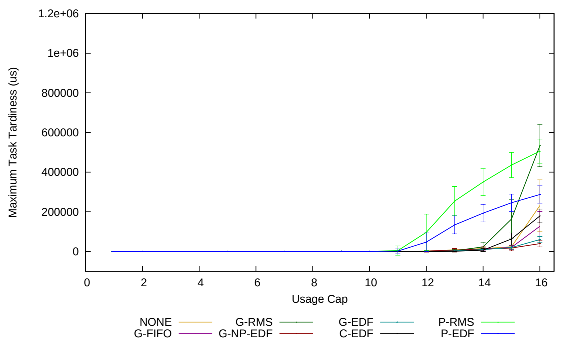


(e)

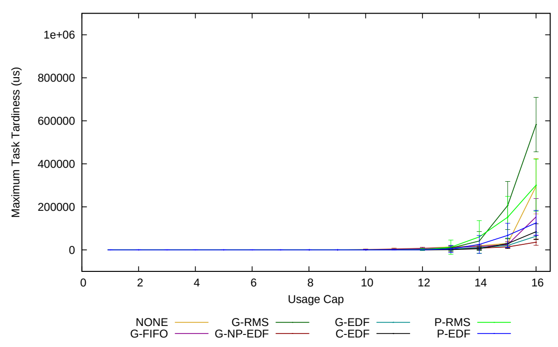


(f)

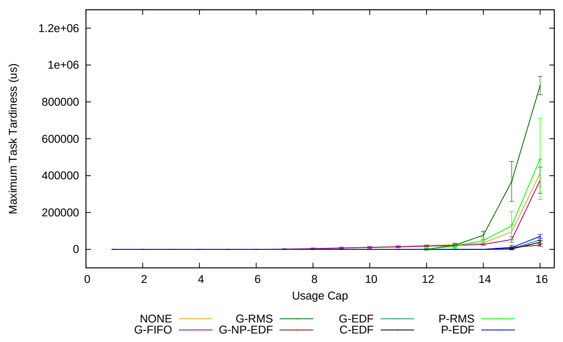
Fig. 9. Hard real-time schedulability results for (a) heavy uniform, (b) heavy bimodal, (c) medium uniform, (d) medium bimodal, (e) light uniform, and (f) light bimodal per-task weight distributions.



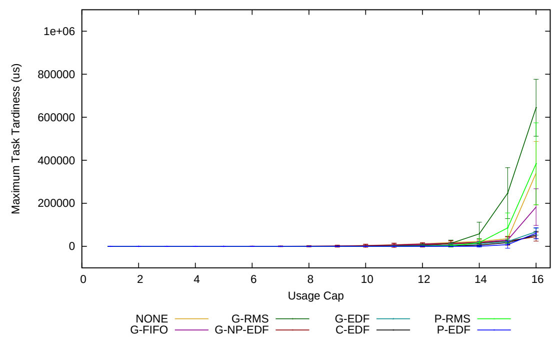
(a)



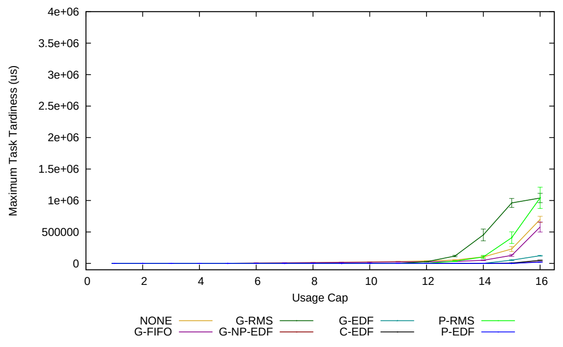
(b)



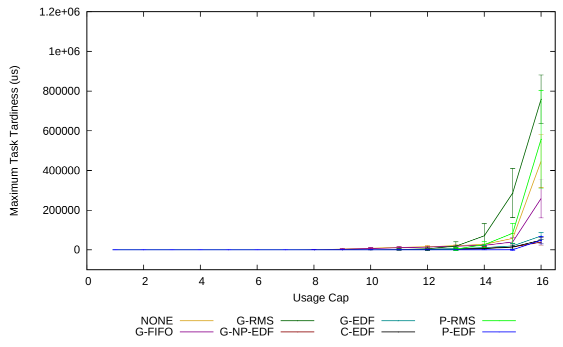
(c)



(d)

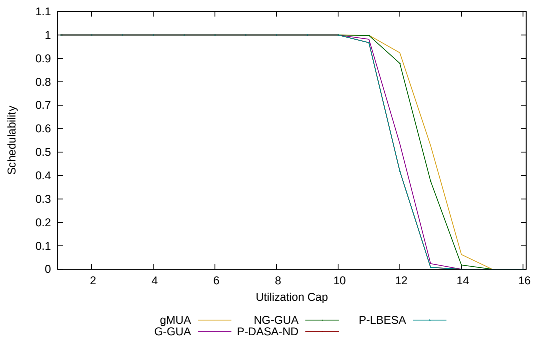


(e)

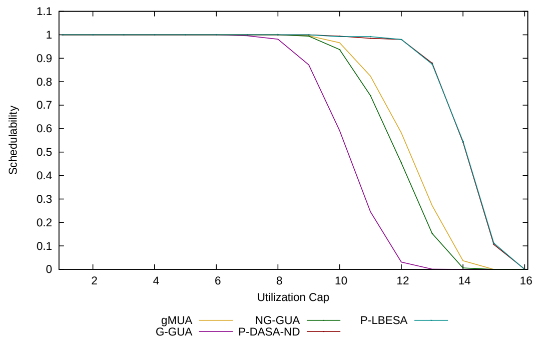


(f)

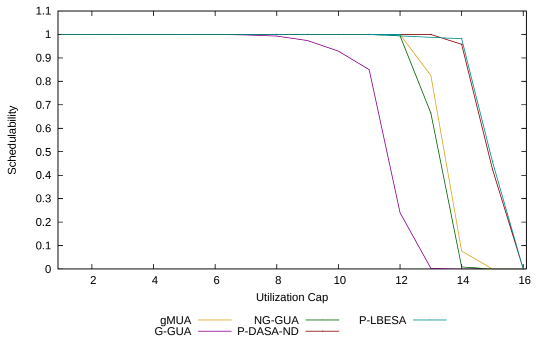
Fig. 10. Mean maximum tardiness results for (a) heavy uniform, (b) heavy bimodal, (c) medium uniform, (d) medium bimodal, (e) light uniform, and (f) light bimodal per-task weight distributions.



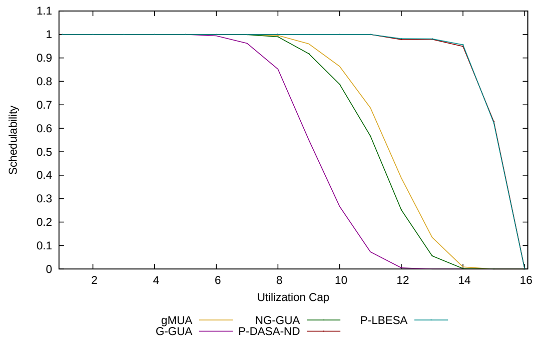
(a)



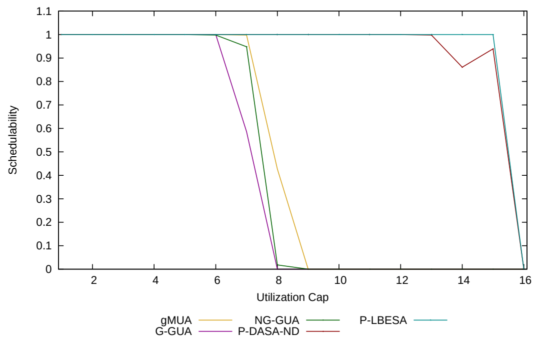
(b)



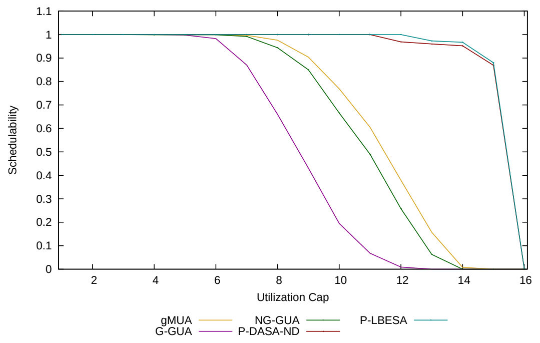
(c)



(d)



(e)



(f)

Fig. 11. Hard real-time schedulability results for TUF schedulers for (a) heavy uniform, (b) heavy bimodal, (c) medium uniform, (d) medium bimodal, (e) light uniform, and (f) light bimodal per-task weight distributions.