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

Matthew A. Dellinger

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

May 28, 2011
Blacksburg, Virginia

Keywords: Multiprocessors, Real-Time, Scheduling, Utility Accrual, Linux

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

Matthew A. Dellinger

(ABSTRACT)

This thesis studies the problem of experimentally evaluating the scaling behaviors of existing multicore real-time task scheduling algorithms on large-scale multicore platforms. As chip manufacturers rapidly increase the core count of processors, it becomes imperative that multicore real-time scheduling algorithms keep pace. Thus, it must be determined if existing algorithms can scale to these new high core-count platforms. Significant research exists on the theoretical performance of multicore real-time scheduling algorithms, but the vast majority of this research ignores the effects of scalability. It has been demonstrated that multicore real-time scheduling algorithms are feasible for small core-count systems (e.g. 8-core or less), but thus far the majority of the algorithmic research has never been tested on high core-count systems (e.g. 48-core or more).

We present an experimental analysis of the scalability of 16 multicore real-time scheduling algorithms. These algorithms include global, clustered, and partitioned algorithms. We cover a broad range of algorithms, including deadline-based and utility accrual scheduling algorithms. These algorithms are compared under metrics including schedulability, tardiness, deadline satisfaction ratio, and utility accrual ratio. We consider multicore platforms ranging from 8 to 48 cores. The algorithms are implemented in a real-time Linux kernel we create called ChronOS. ChronOS is based on the Linux kernel's `PREEMPT_RT` patch, which provides the underlying operating system kernel with real-time capabilities such as full kernel preemptibility and priority inheritance for kernel locking primitives. ChronOS extends these capabilities with a flexible, scalable real-time scheduling framework.

Our study shows that it is possible to implement global fixed and dynamic priority and simple global utility accrual real-time scheduling algorithms which will scale to large-scale multicore platforms. Interestingly, and in contrast to the conclusion of prior research, our results reveal that some global scheduling algorithms (e.g. G-NP-EDF) is actually scalable on large core counts (e.g. 48). 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 also demonstrate that certain classes of utility accrual algorithms such as the GUA class are inherently not scalable. We show that algorithms implemented with scalability as a first-order implementation goal are able to provide real-time guarantees on our 48-core platform.

This work is supported by US National Science Foundation under Grant 0915895, and NSWC under Grant N00178-09-D-3017-0011.

Dedication

I dedicate this thesis to my wife, Sarah.

Sine subsidio tuo, haec non fuisset.

Acknowledgments

This work would not have been possible without the assistance, support, and collaboration of a great many people. First, I would like to thank my advisor, Dr. Binoy Ravindran, for his support and motivation and for providing me with an opportunity to work on something I enjoy.

I would also like to thank Dr. Paul Plassmann and Dr. Cameron Patterson for serving on my committee.

In addition, I would like to thank my friends at the real-time systems lab, particularly Piyush Garyali and Aaron Lindsay. You have motivated me to learning and excellence.

Finally, I am grateful to my family and friends for all the support they have provided. Without your help, this thesis would not have been possible.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
ATC	Asynchronous Transfer of Control
AUR	Accrued Utility Ratio
ccNUMA	Cache-Coherent Non-Uniform Memory Architecture
C-EDF	Clustered Earliest Deadline First scheduling algorithm
CPU	Central Processing Unit
DASA	Dependent Activity Scheduling Algorithm
DSR	Deadline Satisfaction Ratio
EDF	Earliest Deadline First
G-EDF	Global Earliest Deadline First scheduling algorithm
G-FIFO	Global First In First Out scheduling algorithm
G-GUA	Greedy Global Utility Accrual scheduling algorithm
G-NP-EDF	Global Non-Preemptive Earliest Deadline First scheduling algorithm
gMUA	Global Multiprocessor Utility Accrual scheduling algorithm
G-HVDF	Global Highest Value Density First scheduling algorithm
G-NP-HVDF	Global Non-Preemptive Highest Value Density First scheduling algorithm
G-RMS	Global Rate Monotonic Scheduling algorithm
HVDF	Highest Value Density First scheduling algorithm
IPI	Inter-Processor Interrupt

LBESA	Locke's Best Effort Scheduling Algorithm
LLREF	Largest Local Remaining Execution time First scheduling algorithm
LVD	Local Value Density
MMT	Mean Maximum Tardiness
NG-GUA	Non-Greedy Global Utility Accrual scheduling algorithm
P-DASA-ND	Partitioned Dependent Activity Scheduling Algorithm, No Dependencies
P-EDF	Partitioned Earliest Deadline First
P-HVDF	Partitioned Highest Value Density First
P-LBESA	Partitioned Locke's Best Effort Scheduling Algorithm
P-RMS	Partitioned Rate-Monotonic Scheduling algorithm
PFair	Proportionate Fair scheduling algorithm
POSIX	Portable Operating System Interface [for Unix]
RMS	Rate Monotonic Scheduling algorithm
RTAI	Real-Time Application Interface
RTSJ	Real-Time Standard Java
TUF	Time/Utility Functions
UA	Utility Accrual
WCET	Worst-Case Execution Time

Chapter 1

Introduction

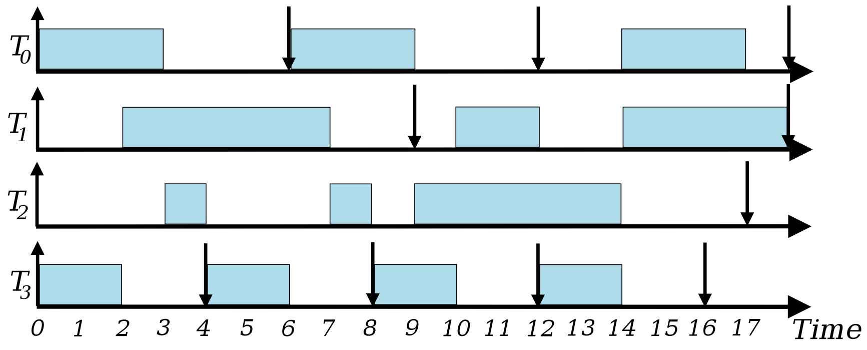
The current trend among chip manufacturers is to improve performance by increasing the core count of processors, rather than increasing clock rates [7]. This is largely motivated by heat and power constraints [89]. Currently, dual and quad-core chips are the standard. AMD is already producing a 12-core processor, which we use in a four-chip configuration in this study, while Intel has demonstrated working 48 and 80-core chips [84]. This trend toward parallelism has even reached the embedded and mobile markets [23]; Qualcomm recently released a new dual-core Snapdragon processor which is already available in a variety of cell phones [54].

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 [34]. 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. Schedulability tests often require knowledge of the worst-case behaviors of tasks, including those on execution times, arrival times, and resource access behaviors.

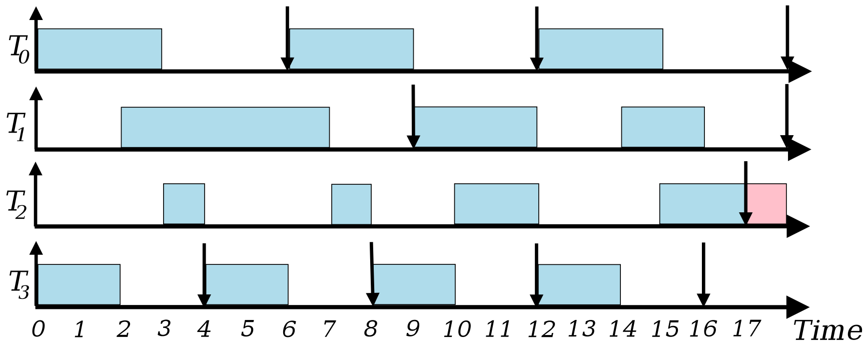
The most widely researched class of multiprocessor real-time scheduling algorithms are global scheduling algorithms. In global scheduling algorithms, all tasks are placed in a single globally accessible queue, and scheduling decisions are made for the entire system based on the contents of this queue. 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 [25]. Example such scheduling algorithms include Global Earliest Deadline First (G-EDF) [24], Global Rate Monotonic Scheduling (G-RMS) [5], and the PFair class [17]. Table 1.1 shows a periodic taskset with deadlines equal to periods, and

Table 1.1: A sample periodic taskset with periods equal to deadlines

Task	Period	WCET
0	6	3
1	9	5
2	17	7
3	4	2



(a) G-EDF schedule on a two processor system

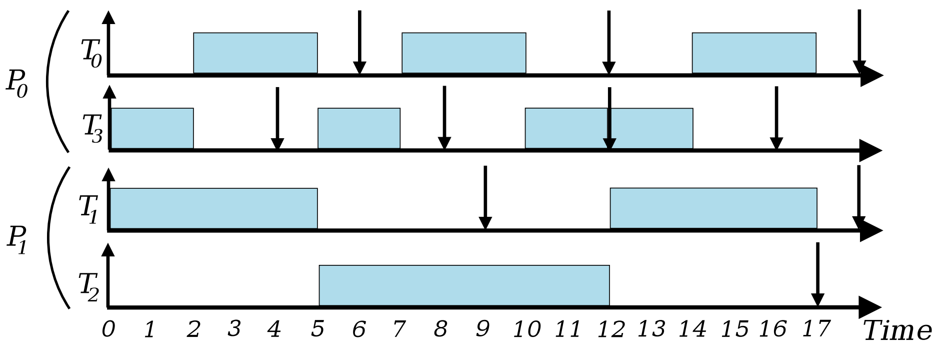


(b) G-RMS schedule on a two processor system

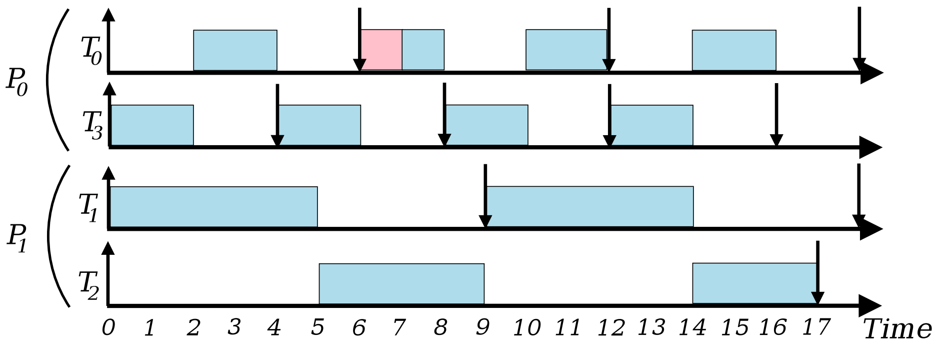
Figure 1.1: Sample schedules for global algorithms on a two processor system. Upward arrows indicate the arrival of a task and downward arrows indicate a deadline of a job and the release of the subsequent job.

Figure 1.1 shows its G-EDF and G-RMS schedules.

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 scheduling problem becomes a set of uniprocessor problems. Uniprocessor scheduling has been extensively studied and optimal and efficient algorithms are well known (e.g. EDF, RMS). Second, since the tasks are not allowed to migrate, the system incurs no overhead due



(a) P-EDF schedule on a two processor system



(b) P-RMS schedule on a two processor system

Figure 1.2: Sample schedules for partitioned algorithms on a two processor system. Upward arrows indicate the arrival of a task and downward arrows indicate a deadline of a job and the release of the subsequent job.

to task migration and cache misses. However, because taskset partitioning is analogous to the bin-packing problem, it is NP-hard in the strong sense [15]. 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. Some examples of algorithms in this class are Partitioned Earliest Deadline First (P-EDF) and Partitioned Rate Monotonic Scheduling (P-RMS). Figure 1.2 shows P-EDF and P-RMS schedules for the taskset shown in Table 1.1 for these scheduling algorithms.

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 [21]. 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.

1.1 Limitations of Current Research

While algorithmic research is both necessary for and beneficial to the advancement of multiprocessor real-time scheduling, the existing body of research is limited in three regards. First, previous research has mainly focused on systems in which it is possible to know or deterministically bound application behaviors such as those on execution times and task arrivals. Although such applications comprise a large and important subset of the real-time problem space [67], applications also exist outside this space. Such applications may have unpredictable task arrivals or non-deterministic execution times, which can cause permanent or transient overloads. Clark [40] and Welch [108] describe two distributed radar tracking systems which often must operate in overload, and suffer from non-deterministic execution times due to data-dependent processing and communication latencies incurred by their distributed structure. Both of their applications desire graceful performance degradation, and so require utility accrual scheduling [52], [39]. Tan and Hsu [103] also present a class of multimedia applications which may require guarantees on deadlines missed while this system is in overload. Extensive research has been performed on scheduling under such conditions on uniprocessors, but only a small amount is known about scheduling such systems on multiprocessors. No existing research has ever explored such a system on a platform with more than 8 cores.

Second, it is not clear how system overheads (and other second order effects) affect the scalability of existing schedulers on large core count platforms. This is difficult to understand analytically and is therefore, correctly, scoped out of research that focus on devising schedulability tests or research that focus on devising 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 to large-scale multiprocessor systems, it is doubtful that task scheduling will be immune to the detrimental effects of scaling [28]. Before such algorithms can be deployed in a 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 platform and concluded that a hybrid solution was needed [32]. Brandenburg et. al. theoretically analyzed the schedulability of a large number of tasksets on a 32-core Sun Niagara platform based on experimental overheads collected [31]. However, neither of these efforts actually scheduled a large number of tasksets on a running kernel scheduler, and neither explored any classes of algorithm other than deadline-based and PFair.

Much of the dearth of experimental research on multicore real-time task scheduling, especially on utility accrual scheduling, is due to the lack of experimental platforms for testing real-time scheduling. In fact, aside from our work, there is only one publicly available platform which provides a framework for advanced multicore real-time scheduling: LITMUS^{RT} [29]. LITMUS^{RT} provides support for many of the classes of scheduling mentioned above, such as G-EDF, P-EDF, and PFair. However, aside from adding real-time scheduling capabilities, it makes no modifications to the Linux kernel to improve determinism or reduce latencies, and thus cannot be considered a real-time operating system. Furthermore, it provides no support for utility accrual scheduling of any kind [52].

1.2 Research Contributions

Based on these lacunae in the contemporary research, we aim to experimentally analyze the scalability of a broad range of global real-time scheduling algorithms, including algorithms that allow execution overruns and runtime uncertainties in a real-time operating system kernel.

To this end, we have created the ChronOS real-time Linux kernel. ChronOS builds upon the existing `PREEMPT_RT` Linux patch, which enhances the real-time capabilities of the standard Linux kernel [106]. ChronOS provides a flexible framework for event-driven multiprocessor real-time scheduling. This allows us to implement a wide range of scheduling algorithms and perform extensive experimentation on them.

To understand the scalability issues encountered, we perform experimentation on ChronOS, focusing on finding and resolving scaling bottlenecks, such as locks. We describe and experimentally demonstrate the bottlenecks discovered and resolve them.

Once ChronOS has been demonstrated to provide scalable event-driven multicore real-time scheduling, we implement 15 multiprocessor real-time scheduling algorithms. Extensive experimentation is performed on these 15 algorithms and the default Linux scheduler to determine the relationship between algorithm design and scalable scheduling.

Our experimentation reveals that it is possible to design global fixed and dynamic priority and simple utility accrual heuristic real-time scheduling algorithms which will scale to large-scale multicore platforms, but also demonstrate that certain classes of algorithms such as the GUA class are inherently not scalable. Additionally, our results reveal that, prior work's conclusion that global scheduling is appropriate only for small core counts (e.g., 8) [31], is only applicable to certain classes of algorithms. In particular, in our implementation, scalability is restricted by lock contention over the creation and distribution of the global schedule and the cost of inter-processor communication, rather than the global queue implementation. We conclude that G-NP-EDF is able to provide high levels of scalability on our 48-core system. We show that algorithms implemented with scalability as a first-order implementation goal are able to provide real-time guarantees on our 48-core platform.

1.3 Thesis Outline

The remainder of this thesis is organized as follows; Chapter 2 provides an overview of previous work related to our goal. Chapter 3 describes the theoretical models we assume as our foundation. Chapter 4 discusses the `PREEMPT_RT` patch, which provides the underlying real-time operating system capabilities to the ChronOS Linux kernel. Chapter 5 presents the architecture and implementation of ChronOS. Chapter 6 describes the scheduling algorithms we implement and measure. In Chapter 7, we analyze the scalability of stop-the-world scheduling architecture within ChronOS. This analysis is used to guide our efforts to increase its scalability. Our experimental results are summarized and analyzed in Chapter 8, and presented in full in Appendices A-F. System overheads are measured in Chapter 9. Chapters 10 and 11 present our conclusions and suggestions for further research in this area.

1.4 Scope of Thesis

It is inevitable given the breadth of the field that this thesis will not comprehensively cover all possible aspects of the problem space. To limit the scope of material covered, we do not address several relevant questions.

The first limitation in scope is an exclusive focus on CPU-intensive application workloads. This is a commonly made assumption when evaluating real-time systems, such as in [13], [52], [31]. However memory-intensive workloads are likely to scale differently than CPU-intensive workloads on large-scale core-count platforms because of the difference in migration and preemption overheads. Both of these actions incur a higher cost for memory-intensive workloads due to cache-miss overheads, meaning their performance will likely suffer more than CPU-intensive workloads on large core counts. We do attempt to provide at least minimal indication of the amount of overhead this type of workload would incur by measuring migration overheads for various working set sizes.

We further limit the scope of our research to consider only independent tasksets—i.e. those without any inter-task precedence constraints or dependencies, such as due to synchronization. As with our workloads, this is an assumption commonly made when analyzing real-time scheduling algorithms ([3], [9], [10]). However in real-world applications, especially those designed to run on multicore applications, inter-task dependencies often arise [27], [39], [76]. In our study, the focus is placed on independent tasks for two reasons. First, our primary focus is on the difference in scalability between various types of scheduling algorithms. Since many locking protocols are agnostic of the scheduling algorithm [27], this can be comprehensively studied without addressing task dependencies. Second, given the large number of algorithms and tasksets used in our experimentation, providing a thorough evaluation of dependent tasksets would require testing a wide range of dependency patterns and scheduling protocols. Given the bulk of the work and its limited relevance to our primary goal, we chose to omit it. To our knowledge, no study has ever looked at multicore real-time locking protocols from the perspective of scalability, so it is difficult to say what effect their inclusion would have on our conclusions.

Third, Brandenburg et. al. have experimentally demonstrated how interrupt handling can effect the performance of global scheduling algorithms [30]. In their experiments, they found that the best real-time performance was achieved by allowing only one processor core to handle interrupts, and excluding this core from executing tasks. However, to simplify our implementation, we allowed all cores to both execute tasks and handle interrupts. We believe that their results for global handling of interrupts, which show limited scalability, are likely due to the use of a vanilla kernel, and that on our system, the use of the Linux kernel's `PREEMPT_RT` patch should offset this.

One method that has been proposed to improve scalability in multicore systems is the use of message passing to convey changes in state, rather than data structures residing in shared memory. This approach has been pioneered at the operating system level by the Barrelfish research operating system [22]. While this approach has been shown to increase the scalability of multicore systems in some cases, it requires a complete redesign of the low-level operating system functionality. For this reason, we limit our scope to traditional monolithic operating system kernels which use locks and shared data structures.

Last, while we cover 16 multicore real-time scheduling algorithms, these are only a small sample of the algorithms developed. In particular, there are a wide range of EDF and RMS variants which have been devised for multicore platforms. We cover six of the most common of these (i.e. G-EDF, G-NP-EDF, P-EDF, C-EDF, G-RMS, and P-RMS) but neglect a several well known variants. Chief among these are EDF-fm, which is a restricted-migration variant of G-EDF for soft real-time systems [3], and the modified EDF algorithm proposed by Srinivasan and Baruah which priorities high-utilization tasks to avoid the Dhall effect [98]. Neither of these algorithms has received as much focus from the academic community as the six we study, but both are capable of outperforming the six we have studied under certain conditions. Another variant is Partitioned Deadline Monotonic Scheduling (PDMS), which is a static priority partitioned algorithm which uses task splitting to increase the lower schedulability utilization bound beyond the 50% typical of partitioned algorithms [71].

Chapter 2

Background

This chapter presents previous work that relates to our objectives. This work is divided up into four sections. First, we examine previous real-time scheduling algorithms classifications and research. Second, we look at various approach to real-time Linux. Third, we present previous multicore real-time scheduling platforms. Fourth, we focus on the two previous studies which relate closely to our work.

2.1 Multicore Real-Time Scheduling

Multicore real-time scheduling algorithms can be grouped in three ways. First, algorithms may be classified based on the extent to which tasks are allowed to migrate between processors. Carpenter has divided the algorithms into three groups: no migration, limited migration, and full migration [34]. No migration implies that once a task has begun executing on a processor, it will not leave that processor. The assignment of tasks to processors is done offline before the taskset is executed by a partitioning algorithm, such as a first-fit or a best-fit algorithm. Any uniprocessor scheduling algorithm can be used in a partitioned manner, although it is necessary to use the correct partitioning algorithm with a given uniprocessor scheduling algorithm to achieve the best performance. In this thesis, we use five uniprocessor algorithms: Rate-Monotonic Scheduling (RMS) [72], Earliest Deadline First (EDF) [35], Locke's Best Effort Scheduling Algorithm (LBESA) [81], Dependent Activity Scheduling Algorithm (DASA) [39], and a simple heuristic algorithm of our own devising, which is discussed later.

The second class is limited migration algorithms. In these algorithms, tasks are allowed to migrate between the system's processors, but only at specific times, most commonly job boundaries. Therefore, at the arrival of each section of a task, the task is placed on a processor, and executed there. Once it has finished a section, it may be moved to a different processor. Anderson et. al. discuss this model in [3]. The Global Non-Preemptible EDF

(G-NP-EDF) algorithm described in [45] is an algorithm which follows this model.

Third, algorithms may allow full migration. In this model, any task is allowed to migrate to any processor at any time. This is the most common model employed. Scheduling algorithms which use it include Global EDF (G-EDF) [24], LLREF [37], PFair [58], and the GUA class of algorithms [52].

There is also a fourth class of algorithms which Carpenter does not address. These might also be termed restricted migration, but are restricted in a different respect than Carpenter's limited migration class. Rather than only being allowed to migrate at certain points, tasks are allowed to migrate at any time, but only between certain processors. As with partitioned scheduling, the tasks are separated offline, but rather than being assigned to a single processor, they are assigned to a set of processors. A multiprocessor scheduling algorithm is then run on these processors. Two examples of such a scheduler are Clustered EDF (C-EDF) [20] and MOCA [69].

Algorithms can also be classified based on the complexity of the priority mechanism they use [3]. These have been divided up into three classes: fixed priority, task-dynamic, and job-dynamic. Fixed priority scheduling assigns each task in the system a single priority for all of its jobs. The most common fixed priority scheduling algorithms is Rate Monotonic Scheduling, or RMS. This has been shown to be an optimal static priority scheduling algorithms on uniprocessors. RMS has been directly extended to multiprocessor scheduling with Global RMS and Partitioned RMS. Unfortunately, unlike the uniprocessor case, no simple utilization bound exists for multiprocessor fixed priority scheduling.

Task-dynamic priority algorithms are those in which different jobs of the same task may have different priorities, but the priority of each job never changes. The most common task-dynamic priority algorithm is Earliest Deadline First (EDF). EDF is an optimal algorithm for uniprocessors, and like RMS has been directly extended to multiprocessors by global scheduling and partitioning, resulting in G-EDF and P-EDF.

Job-dynamic priority algorithms are those in which the priority of a job may change during its execution. There is no single scheduling algorithm which is commonly associated with the job-dynamic approach, but a wide variety of algorithms utilize it. The PFair and LLREF classes of algorithms both utilize this approach, as do all algorithms which rely on the value-density concept defined in Chapter 3, such as the GUA class.

Last, algorithms may be classified by their performance goals. This allows us to broadly group algorithms into three classes. First, there are traditional algorithms, such as EDF and RMS variants. These algorithms attempt to meet all deadlines in underloads, but cannot always do so. They assume that the system will never be in overload, and their performance significantly degrades if such a condition occurs [19]. This is the largest class of algorithms, and a wide range of algorithms have been designed to meet specific cases.

A class of algorithms also exists which is able to meet all deadlines for all tasksets on an m processor system, as long as the total utilization U is less than m. These algorithms can be divided into two subclasses — the PFair class, which is based on scheduling at specific intervals, called quanta, and the LLREF class, which works in an event driven fashion. While both of these are theoretically optimal, both require significant computation to generate the schedule [45], [37].

Finally, a third class of algorithms exists which is designed to schedule in overload. When a system is in overload, the scheduling algorithm must discard tasks. This should be done in such a way that "good" task are kept, and "poor" tasks are allowed to fail. LBESA and DASA [39] provide this kind of performance by heuristically discarding tasks. The GUA class of algorithms and gMUA [36] are extensions of LBESA and DASA for multiprocessors. There are also non-heuristic solutions to this problem; D^{over} [68] and MOCA [69] are schedulers which provide the optimal competitive ratio for uniprocessor and multiprocessors respectively.

2.2 Real-Time Linux

Since one of our main contributions is a real-time Linux kernel based on the `PREEMPT_RT` patch, attention must be paid to the field of real-time Linux. Given the maturity and widespread use of Linux, it is not surprising that it has received substantial interest from both the academic and industrial communities. To date, there have been a wide range of real-time Linux distributions. These may be divided into two groups.

The first group is those based on the nano-kernel approach. This approach is based on the idea that the Linux kernel, as a large complex piece of software, is inherently incapable of providing hard real-time guarantees. To this end, the distributions which follow this approach run the Linux kernel itself as a task on a nano-kernel. Applications needing hard real-time performance can then be run as tasks on this nano-kernel, and therefore are immune from the performance issues of the Linux kernel. Alternately, a separate real-time kernel can be run as a task like Linux, and the real-time applications can then be executed in this environment. When tasks need to communicate with Linux tasks, they do so through lock-free buffers. The most popular distributions in this model are the descendants of the RTLinux distribution, RTAI and Xenomai [14]. Both of these distributions use the Adeos nano-kernel.

However, this model has a large disadvantage; real-time applications must be written to the APIs of the nano-kernel or the real-time kernel, rather than for Linux itself. While both RTAI and Xenomai attempt to provide similar APIs within the nano-kernel environment, they cannot provide the full set of Linux APIs.

This motivates the second model, which is the modification of the Linux kernel itself to tighten the performance provided. This is the approach taken by the `PREEMPT_RT` patch [85], as well as several commercial real-time Linux distributions such as RedHawk Linux [92]. This allows the application to make use of the full range of Linux APIs inside a standard Linux environment, but has the disadvantage of not providing as high a level of hard real-time performance [88]. Despite this drawback, there are a large number of commercial and academic systems based on the `PREEMPT_RT` patch, such as Kansas University Real-Time Linux (KURT) [90] and the Fujitsu Limited and Tokyo stock exchange [105].

2.3 Multiprocessor Real-Time Scheduling Platforms

Given the extensive algorithmic research on real-time scheduling algorithms, it would be logical to assume that there are a wide range of platforms which can be used as experimental frameworks for the implementation and testing of such algorithms. Unfortunately, that is not the case. None of the previously mentioned real-time Linux platforms provide any kind of advanced scheduling. The majority of those authors which present implementations of their work such as [97] and [6] implement their schedulers directly into an operating system kernel or middleware, implement only a single algorithm, and do not provide full source code for their implementations. While this is sufficient to publish a paper, it provides no benefit to future authors who wish to perform implementation-based studies.

There several exceptions to this; both the Alpha [38] and Spring [99] kernels were both extensively documented and continue to influence the design of real-time operating system design. Both existed before the Internet and open-source community and therefore the source-code is not publicly available, and both were designed for highly proprietary hardware, but the detailed implementation descriptions and discussion of the logic behind it allow them to remain relevant.

More recently, Wang and Lin described their RED-Linux platform, which allows time-driven scheduling in a modified Linux kernel [107]. However, it only supports uniprocessor scheduling, and the source-code is not publicly available. Similarly, Li and Ravindran demonstrated a Java-based real-time scheduling middleware, which they formally verify [77]. They implement a variety of algorithms, including EDF, LBESA, and DASA. However, as with RED-Linux, it only supports uniprocessor scheduling and the source code is not publicly available.

Aside from the ChronOS kernel we have developed, there is in fact only one mature platform which provides advanced multiprocessor task scheduling and has freely available source code. This is the Linux Testbed for Multiprocessor Scheduling in Real-Time systems, or Litmus^{RT} [29], [33]. Litmus^{RT} was developed to fill exactly this void. It provides for advanced multiprocessor real-time scheduling algorithms such as G-EDF, C-EDF, P-EDF, and PFair. Having been in development for nearly five years, it is quite mature, and has received significant interest from both the Linux kernel and academic research communities. Unfortunately, it does not utilize the `PREEMPT_RT` patch or make any other changes to Linux itself aside from scheduling. Because of this, it cannot be called a real-time Linux kernel.

2.4 Multiprocessor Real-Time Scheduling Scalability

It is therefore not surprising that the only studies of the performance of multiprocessor real-time scheduling algorithms occurred on Litmus^{RT}. First, Calandrino et. al. used Litmus^{RT} as a test platform for C-EDF, which they propose as a viable middle ground between G-EDF and P-EDF [32]. They demonstrate that on a 64-core platform, the scheduling overheads associated with G-EDF are prohibitive, and the bin-packing problems associated with partitioning limit the performance of P-EDF. They then generate a large number of tasksets, inflate task execution times with the overheads generated, and perform schedulability tests to determine the required number of processors to schedule each taskset under each algorithm. They conclude that, while G-EDF and P-EDF are able to outperform C-EDF by small margins in specific cases, both are also significantly outperformed by C-EDF in others. They therefore conclude that C-EDF provides the most consistent performance across a wide range of tasksets on high core-count multiprocessor systems. However, this study is limited in several regards; first, the overheads used were empirically generated based on a small number of measurements taken from a MIPS architecture simulator, rather than measured on a running platform. Second, only average-case overheads were considered when inflating the task execution times. While both of these approaches are reasonable, they do not necessarily provide an accurate picture of a running hardware system, and cannot be directly compared to other architectures, such as x86. Additionally, only soft real-time schedulability tests were performed.

In the second study, Brandenburg et. al. analyzed the scalability of five multiprocessor real-time scheduling algorithms in LITMUS^{RT} on a 32-core Sun UltraSPARC T1 Niagara platform [31]. The algorithms measured are P-EDF, G-EDF, C-EDF, G-NP-EDF, and two variants of PFair, PD^2 and S-PD^2. Like Calandrino, rather than actual scheduling the tasks in a running kernel, they use overhead measurements to inflate task execution times, and then perform offline schedulability tests. However, unlike Calandrino, they use overheads measured from a running platform and measure both average and worst-case overheads. They demonstrate that on this platform, G-EDF and PD^2 perform poorly for hard real-time loads. They further conclude that C-EDF consistently outperforms G-EDF, and S-PD^2 and P-EDF are consistently the best performers. For soft real-time loads, they conclude that S-PD^2 and C-EDF perform well. Finally, they conclude that the largest hindrance to scalability is the implementation of the global queue.

Chapter 3

Models

3.1 Hardware Model

We consider a homogeneous multicore, multiprocessor system with N cores and M processors. We assume that each of the N cores are identical and uniformly distributed across the M identical processors. In this thesis, the term core will be used to refer to a single computational unit and its associated unshared components, such as low-level cache, while the term processor will be used to refer to one or more computational units residing on the same physical die and potentially sharing on-die components such as high-level cache and memory controllers [66]. A diagram of one such multicore processor (the AMD Magny-Cours 12-core chip used in our study) is shown in Figure 3.1 [1]. This particular chip contains two separate 6-core processor dies on the same physical chip. The two dies are connected together via a cache-coherent high-speed interconnect. Figure 3.2 shows the same chip in a two-chip configuration, and Figure 3.3 shows a four-chip configuration [2]. Like the two dies on each chip, the neighboring dies are connected via a cache-coherent high-speed interconnect. This kind of multiprocessor architecture is known as a cache-coherent non-uniform memory architecture (ccNUMA) platform [100].

Each core is assumed to have a local cache to which only it has access. Each processor may also have a shared cache to which all of its cores have access. Local cache access takes some constant time μ_{lc}, and shared cache access takes μ_{sc}, where $\mu_{sc} > \mu_{lc}$.

All processors are assumed to have equal amounts of local memory, which they can access directly in some time μ_{lm}. Each processor is also able to access every other processor's memory. This interprocessor memory access has a constant time cost of μ_{rm}, where $\mu_{rm} > \mu_{lm}$. Therefore, we say that $\mu_{rm} > \mu_{lm} > \mu_{sc} > \mu_{lc}$. The system's memory is presented to the applications in a single address space.

While ChronOS is designed around such a system, it will function on any x86 platform

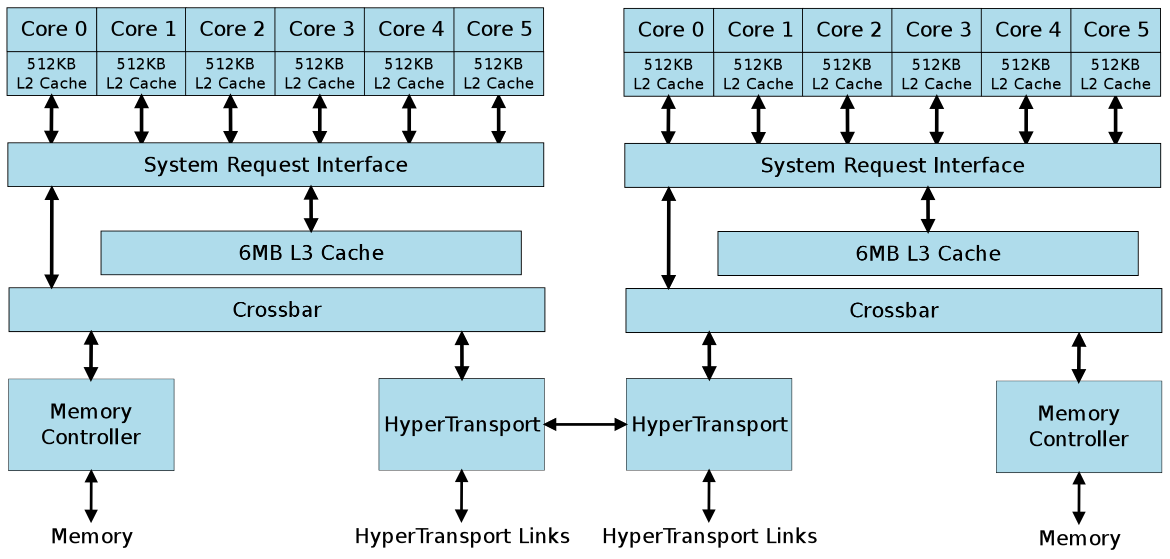


Figure 3.1: AMD Magny-Cours 12-core processor architecture

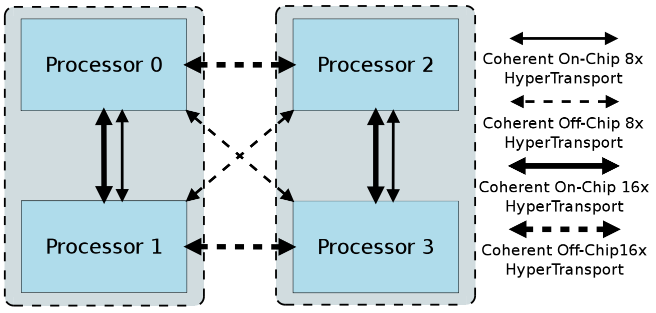

Processor 0
Processor 1
Processor 2
Processor 3
Coherent On-Chip 8x HyperTransport
Coherent Off-Chip 8x HyperTransport
Coherent On-Chip 16x HyperTransport
Coherent Off-Chip16x HyperTransport

Figure 3.2: AMD Magny-Cours 2-Chip Interconnect Topology

supported by the standard Linux kernel. However, if these assumptions are violated, various subsystems of ChronOS will likely perform in a less than ideal manner, and overall performance may seriously decrease.

3.2 Task Model

We consider a programming model in which the work is divided up into a set of tasks. Each of these tasks T_i has an arrival pattern which may be described as periodic, sporadic, or aperiodic. If a task is periodic, it is said to have a period P_i. Sporadic tasks, like aperiodic tasks, have no consistent arrival pattern. However sporadic task have a guaranteed minimum inter-arrival time [63]. Each individual execution of a task T_i is referred to as a phase or job [39]. The J^{th} job of task T_i is written T_i^j. Each job of a task may also have a worst-case or good-faith execution time e_i^j, a deadline d_i^j, and a release time R_i^j, identifying the time at

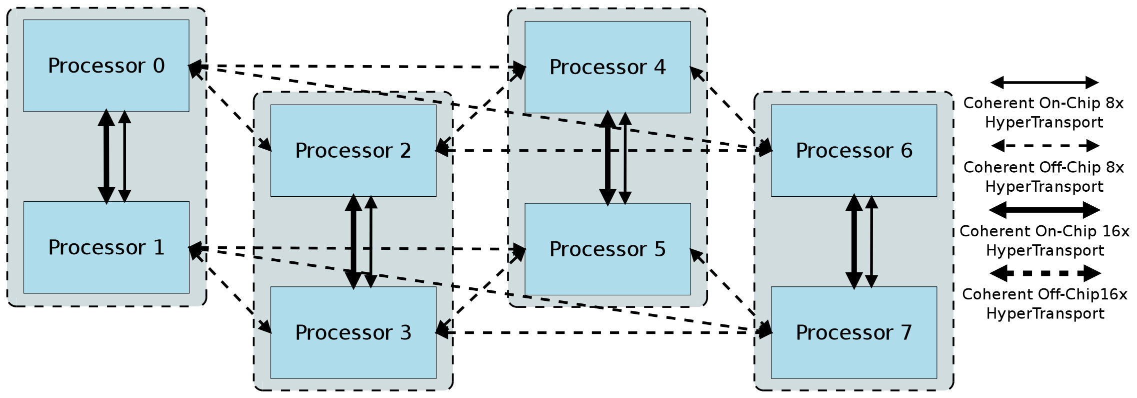

Processor 0
Processor 1
Processor 2
Processor 3
Processor 4
Processor 5
Processor 6
Processor 7
Coherent On-Chip 8x HyperTransport
Coherent Off-Chip 8x HyperTransport
Coherent On-Chip 16x HyperTransport
Coherent Off-Chip16x HyperTransport

Figure 3.3: AMD Magny-Cours 4-Chip Interconnect Topology

which the job becomes eligible for execution. Each job also has a utilization U_i^j, which may be computed by $e_i^j/(d_i^j - R_i^j)$. No arrival pattern is assumed, and in the case of periodic tasks, we do not assume that $P_i = d_i^j - R_i^j$. ChronOS supports the full range of arrival patterns, including fully aperiodic, sporadic, and periodic, and also both event-triggered and time-triggered.

Each job of a task contains one or more scheduling segments, which define a part of the thread which must be executed with real-time constraints. Real-time segments have a single start and end point, which are communicated to the underlying operating system through the `begin_scheduling_segment` and `end_scheduling_segment` operations, based on the Real-Time CORBA APIs [94]. In ChronOS, these operations are named `begin_rt_seg` and `end_rt_seg`. These operations are used to communicate time constraints (e.g. periods, deadlines, TUFs) to the operating system. While some systems such as Alpha and Real-Time CORBA 2.0 allow nested segments (which they refer to as `deadline`/`mark` pairs), we only allowing sequential segments [91], [94]. Unlike any of the previously mentioned systems, we do allow the direct continuation of one segment into another. Therefore, if two segments are adjacent to each other, we follow a `begin_rt_seg` call with a second call, rather than calling `end_rt_seg` followed by `begin_rt_seg`.

The details of `begin_rt_seg` and `end_rt_seg` are further discussed in Chapter 5.

3.3 Threading Model

Each task is represented to the underlying operating system as a thread. For this, there are two common models. Either each job T_i^j of T_i can be executed on the same thread t_i, or a new operating system thread can be created for job. If the first approach is used, at the end of T_i^j, the thread sleeps until the beginning of T_i^{j+1}. If the thread finishes execution of T_i^j after the time at which T_i^{j+1} should begin, T_i^{j+1} becomes available for execution immediately. We call this the "thread-per-task" model. This model is advantageous because, as it uses the same thread for the entire duration of the task, there is no overhead for thread creation/destruction. Also, since the same thread is being used, it is possible to save state information from one job to the next in thread-local variables. This is the most common threading model, being used in systems such as Litmus^{RT}, RTAI, and RTSJ [26], [62], [47].

Alternatively, a new operating system thread can be created at the arrival time of each job T_i^j. In this model, the thread t_i is created, immediately executes T_i^j, and then terminates. We call this model "thread-per-job". This model is often used for aperiodic tasks because it fits well with event-based job arrival. It is generally disadvantageous for periodic tasks, since it requires the creation and destruction of a thread for each job. This model is available in Linux, where the `timer_create()` function will create a new thread for the function to be run if the `SIGEV_THREAD` option is specified.

Additional models are possible, such as a thread-pool model. In this model, a variety of jobs are presented to an interface such as a virtual machine. The jobs are then assigned to a predetermined set of threads by this intermediate layer. Different jobs of the same task may therefore be assigned to different threads. Two examples of such a threading service are Java's ThreadPool class and Mac OS X's Grand Central Dispatch [59], [61].

We assume no particular model, except that each job is associated with a single operating system thread for the duration of a scheduling segment. We likewise provide no specific APIs for threading, preferring to rely on the existing APIs provided in Linux.

3.4 Timeliness Model

We measure the timeliness of each job through the use of a time-utility function (TUF) [65]. This allows us to separately represent the importance and urgency of a job. While this separation is of little interest in a hard real-time system in which all deadlines must be met and therefore the urgency is of prime importance, in a soft real-time system this separation allows the execution of jobs based on the value that will be accrued for the system.

We represent the time/utility function of a job T_i^j as $V_i^j(t)$. Each job has a release time R_i^j and a termination time TM_i^j. The TUF $V_i^j(t)$ is therefore defined on the interval $[R_i^j, TM_i^j]$.

As Figure 3.4 shows, TUFs can take a wide variety of forms. It is even possible to have a TUF defined and positive on the interval $[R_i^j, \infty]$, implying that the job may be completed at any time after its release and still provide some benefit to the system [78].

The concept of a hard deadline is not implicit in TUFs, but can easily be expressed as a downward step function (see Figure 3.4(b)) [52]. Like Clark, Garyali, and others, in this thesis we consider only a downward step TUF in which a utility V_i may be accrued if the job T_i^j is completed before its deadline d_i^j, and no utility is accrued if it is completed afterward [39], [52]. This is in contrast with systems like the Alpha and scheduling algorithms

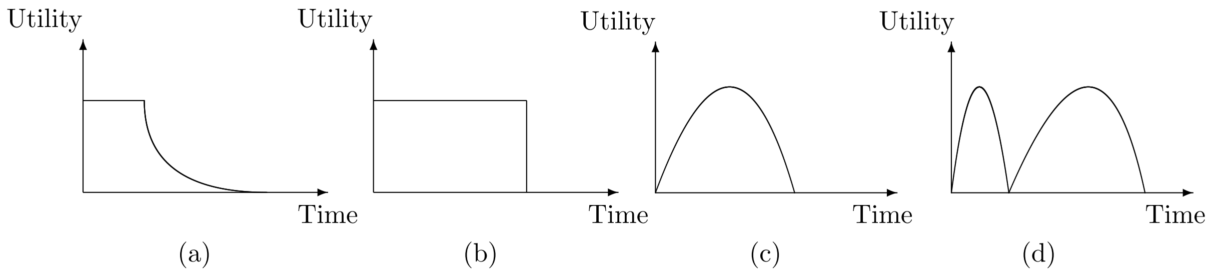


Figure 3.4: Soft timing constraints specified using Jensen's Time-Utility Functions [78]

like GUS and LBESA, which allow a broad range of TUFs [38], [74], [81]. The downward step TUF is a generalized form of the classic deadline scheduling, in which all tasks are considered to accrue a constant utility if finished before their deadline, and no utility if finished afterwards. One extension of this concept is the local value density of the test. The local value density LVD_i^j of a job T_i^j is equal to the utility V_i^j to the job's remaining execution time [52].

If a job fails to finish by its deadline, it is said to be tardy. Therefore, the tardiness θ_i^j of a job T_i^j may be expressed as $TM_i^j - d_i^j$.

In ChronOS, TUFs are specified by passing a utility, execution time, and deadline to the scheduler via `begin_rt_seg`.

3.5 Abort Model

We consider the model in which some tasks, designated abortable tasks, may have their execution aborted. Any job of an abortable task my be aborted at any time. There are a variety of reasons why it may be desirable to abort a job. In this thesis, we primarily consider aborting jobs that are either deadlocked with another job over a resource or have passed their deadline. When the decision to abort a job is made, the job is sent an abort signal by the scheduler, which upon the resumption of execution, will invoke an `abort_handler`. The invocation of the abort handler will not necessarily occur immediately upon the resumption of execution, but will occur within a finite, bounded amount of time l_a. The invoked abort handler will place the task and any other objects it was interacting with into a known good state, including freeing any resources the job acquired.

This abort model differs from previous authors in two points [39], [42], [75]. First, we do not assume immediate invocation of the abort handler upon resumption of execution. Second, we do not assume a single abort_handler for the entire segment [52]. Instead, the thread checks if it has been aborted at specific points, and if it has been, executes the correct abort_handler for that point. Therefore, rather than requiring an abort handler that is capable of undoing all of the actions of the segment, regardless of the point of preemption, our segment can be

```
void abort() {
    ...
    if(b->list.next == a)
        b->list.next = a->list.next;
    if(b->list.next->list.prev == a)
        b->list.next->list.prev = b;
    if(a->list.next == b->list.next)
        a->list.next = h->list.next;
    if(a->list.prev == b)
        a->list.prev = h;
    if(a->list.next->list.prev == h)
        a->list.next->list.prev = h;
    if(h->list.next == a->list.next)
        h->list.next = a;

    ...
    exit();
}

void task() {
    ...
    register_abort_handler(&abort);
    ...
    h->list.next = a->list.next;
    a->list.next->list.prev = h;
    a->list.next = b->list.next;
    a->list.prev = b;
    b->list.next->list.prev = a;
    b->list.next = a;
    ...
}
```

(a)

```
void abort_list_1(...) {
    h->list.next->list.prev = a;
    h->list.next = a;
    exit();
}

void abort_list_2(...) {
    b->list.next = a->next;
    a->next->prev = b;
    a->next = h->next;
    a->prev = h;
    abort_list_1(h, a);
}

void task() {
    ...
    h->list.next = a->list.next;
    a->list.next->list.prev = h;
    if(task_aborted())
        abort_list_1(h, a);
    a->list.next = b->list.next;
    a->list.prev = b;
    b->list.next->list.prev = a;
    b->list.next = a;
    if(task_aborted())
        abort_list_2(h, a, b);
    ...
}
```

(b)

Figure 3.5: Sample code for (a) the single abort handler model and (b) our model

divided up into a series of operations, each of which has its own abort_handler. Consider the following example: an object A is being removed from the head of a circular doubly linked-list and inserted into a second circular doubly linked list after some item B. Assume we have exclusive access to both lists, and therefore no locks or other forms of data protection are necessary. If the task is aborted, we want to unroll the activities of the task, and then exit. Code for this example is shown in Figure 3.5 for each abort model.

If this operation was aborted and an abort_handler of the type considered by previous authors was invoked, the handler would be required to unroll the operations. However, unrolling the operation without any knowledge of where the sequence was interrupted is difficult, because one of the lists may be in any of seven states, two of which are invalid. However, the sequence of operations can clearly be broken up into two separate operations, each of which, if executed uninterrupted, can easily be unrolled.

The evolution of modern programming languages also forces this change in past abort models. Previous authors relied on mechanisms like POSIX signals and cancel that provide an asynchronous transfer of control (ATC) [49]. However, in many modern managed runtime environments such as RTSJ, ATC is strictly forbidden except through specific mechanisms [79]. RTSJ allows tasks to be aborted via the `Interruptible` interface. However, this interface is only available from inside the JVM, which runs counter to our assertion that the OS scheduler should signal the abort. The Java language provides no standard mechanism to receive POSIX signals [104]. It should be noted that in addition to asynchronous cancellation, POSIX does specify `PTHREAD_CANCEL_DEFERRED`, which defers cancellation until a designated cancellation point, similar to our model [60].

Furthermore, unlike previous authors, we do not assume the expression of real-time constraints for the abort_handler [39], [95]. This is done because each real-time segment may have many different abort_handlers, and it is assumed that the cost of expressing time constraints for each of these to the underlying operating system is inordinately large, compared to the execution cost of the abort handler. There are a number of cases in which this assumption may be invalid, such as a job which has manipulated an actuator in a physical system and must reset the actuator when aborted. For these cases, ChronOS does provide an `add_abort_handler` operation that sets the deadline, execution time, and utility for a job's abort handler. However, the request and release of resources do not automatically modify the parameters of the job's abort handler, as Clark does [39]. In this thesis, we assume all abort handlers are "short", and therefore we do not use this operation.

Our abort model is not without precedent, as it is similar to the model used by Li et. al [78]. They allow threads to provide execution times for abort_handlers, but do not allow the abort_handler to arbitrarily preempt job execution and allow the job to specify allowable abort points.

The implementation of our abort mechanism and the APIs needed are discussed in Chapter 5.

3.6 Resource Model

While this thesis gives little focus to resources, many of the algorithms described and measured provide resource-aware scheduling. From a general standpoint, a resource can be anything that the programmer must explicitly request and release which can only have a finite number of users at a time. Like Clark, we restrict our consideration to single-user resources [39]. This implicitly means that what constitutes a resource depends on the language being used; in C and C++, memory is a multi-user resource since the programmer must explicitly both allocate and deallocate it. In Java however, the use of thread-local allocation buffers and garbage collection often allows the programmer to ignore allocation and deallocation, and therefore memory is not a resource [87]. The prototypical resource is a mutex, which must be requested with a `lock` operation, must be released with an `unlock` operation, and can only have one owner at a time. Interfaces to many physical systems, such as disks and networks, can be represented as resources in this model.

We assume that a resource M_k may be requested by any job T_i^j via a `request_resource` operation, and may be released by a `release_resource` operation. T_i^j is `blocked` on M_k until its request is granted. Once the request is granted, T_i^j is said to be the owner of M_k, and all other jobs requesting M_k will block. Each job may be the owner of an unlimited number of resources, but may only be blocked on one resource at a time. This is called the single-unit resource model [52]. Jobs are allowed to hold resources in both a nested and overlapped pattern.

Chapter 4

PREEMPT_RT Patch

The standard Linux kernel provides some real-time capabilities, but these are only suitable for soft real-time systems. These features include the `SCHED_FIFO` and `SCHED_RR` real-time scheduling policies, high resolution timers, and `POSIX`-defined specifications such as priorities. Both `SCHED_FIFO` and `SCHED_RR` work within the priority system — a task of a higher priority always preempts a task of a lower priority. `SCHED_FIFO` allows the user to grant a thread the processor and guarantee it freedom from preemption by any lower priority task. `SCHED_RR` equally divides the processor between all tasks at a given priority, and similarly provides freedom from preemption by lower-priority tasks. The Linux kernel provides a priority range from 0 to 99.

However, this is insufficient for a hard real-time system. Since interrupts run in a special interrupt context, all tasks, even those running under a real-time scheduling policy, are subject to preemption by interrupts. Furthermore, large sections of the kernel are non-preemptible, meaning that it is impossible to place tight bounds on the latency a task my experience. Both of these factors make it impossible to provide the necessary guarantees for a hard real-time platform.

The `PREEMPT_RT` patch was designed by the authors of the Linux kernel to solve this problem [106]. It enacts several changes to the underlying structure of the Linux kernel designed to reduce the latencies experienced by real-time tasks. It accomplishes this by reducing the amount of kernel code which is non-preemptible. In the following sections, we review the three major changes. The are a large number of other small changes which are left unmentioned here because they do not pertain directly to our work. Additionally, there are a variety of features which began as part of the `PREEMPT_RT` patch, but have now moved into the mainline kernel, such has the `hrtimer` high-resolution timer interface [96].

4.1 Preemptible Critical Sections

The most expansive change made by the `PREEMPT_RT` patch is the use of preemptible critical sections. In the standard Linux kernel, a locking call on a `spinlock_t` or `rwlock_t` spins until it acquires the lock. In this state, preemption and sometimes hardware interrupts are disabled. The `PREEMPT_RT` patch replaced many of these spinlocks with a new type of lock called an `rt_mutex`. Rather than spinning, this new type of lock can block and sleep, allowing preemption and migration of the calling thread. This allows critical sections to be preempted [85].

This new lock is not used throughout the entire kernel. In some cases, such as the scheduler, it is still necessary to provide a non-preemptible critical section. For this reason, the original `spinlock_t` has been preserved. However, its use is limited to only such areas as are absolutely necessary.

4.2 Preemptible Interrupt Handlers

A second area of non-preemptible code that the `PREEMPT_RT` patch resolves is that of interrupt handlers. Traditionally, these are run in an interrupt context above the process context, which means all running processes suffer interference from them. The `PREEMPT_RT` patch moves almost all interrupts into process context by handling them in kernel threads. This means that, like other threads, they have a priority and can be preempted by higher priority threads. By default, all threaded interrupts run at a priority of 50.

A few select interrupts such as the per-CPU timer interrupt and the floating-point co-processor interrupt are left in interrupt context.

Because interrupt handlers now run in process context, the idea of disabling interrupts no longer makes sense. In the stock kernel, if a section of code desired to access data sometimes modified by an interrupt handler, it would have to disable interrupts to prevent preemption and provide data protection. However, if an interrupt runs in process context, a `spinlock_t` is sufficient, since the interrupt will block while acquiring the lock, and control will return back to the holder of the lock [85].

This is both useful and dangerous. It means that interrupts are preemptible and have priorities, which means that it is possible to design an application which receives no interference from them. However, executing an application above the interrupt handler priority will cause the interrupt handlers to receive interference from the application. For some interrupts, such as disk events, this may not be a problem. However for others, such as timer events, this behavior may not be desirable.

4.3 Priority Inheritance

Before the introduction of preemptible critical sections, priority inversion was never a possibility. If a low-priority task acquired a lock, preemption was disabled, which prevents a higher priority task from causing priority inversion by attempting to take the lock. However, once preemptible critical sections are introduced, it is easy to see how priority inversion can occur. To prevent this, priority inheritance has been implemented for `spinlock_t` and `rwlock_t`. This prevents priority inversion from occurring. Additionally, for code sections which hold locks for extended periods of time, preemption points have been added in which the thread will release the lock to a higher-priority waiter and then reacquire it when the higher-priority thread has finished [85].

4.4 Experimental Results

In order to demonstrate the improvement in latency experienced by user space applications, we performed experiments using two commonly used real-time benchmarking applications. Cyclictest and Signaltest are both widely used and provided through repositories for many Linux distributions. Our test platform was based on a quad-core AMD Phenom 9650 and ran Ubuntu 10.04. Ubuntu provides both applications through the `rt-tests` package.

Both applications measure the average and worst case latencies experienced. Cyclictest creates a set of periodic tasks which wake, work and sleep. Signaltest creates a set of tasks which send and receive signals. In each program, latencies are recorded each loop, and statistics are collected [43].

In order to properly test our system under various load conditions, we need a deterministic disturbance generator. A commonly used method is a parallel compile of the Linux kernel using the -j flag of `make` [43]. This generates a significant memory, processor, and disk load. Added to this, we use four instances of the `ping` command with the -f flag set to ping the localhost. One instance was tied to each of the four cores in our machine using the `taskset` command. This creates a significant and consistent interrupt load.

For each test, we test twelve possible conditions. First, we use two different kernels: the vanilla 2.6.31.12 kernel and the same kernel with the `PREEMPT_RT` patch applied. Additionally, since interrupts are threaded in the `PREEMPT_RT` kernel, we test that kernel at two different priorities — one above the interrupt level and one below. The standard kernel is only tested at one priority. Each test was run with the system under no load and under full load, and for two different numbers of threads (5 and 10). The results are shown in Tables 4.1-4.4.

From these results it is clear that the `PREEMPT_RT` patch is acting as intended. In no case does the minimum, maximum, or average latency increase on the `PREEMPT_RT` kernel. Also, as expected, latencies are significantly lower when the tasks are run at a priority of 80, placing

Table 4.1: Cyclictest results for 5 threads, 100000 loops, μs

		No load			Full load		
Kernel	Priority	Avg.	Min.	Max.	Avg.	Min.	Max
2.6.31.12	20	10	2	4882	8	2	10988
2.6.31.12-rt21	20	9	1	723	5	2	256
2.6.31.12-rt21	80	8	1	659	4	1	25

Table 4.2: Cyclictest results for 10 threads, 100000 loops, μs

		No load			Full load		
Kernel	Priority	Avg.	Min.	Max.	Avg.	Min.	Max
2.6.31.12	20	11	1	7957	10	1	11130
2.6.31.12-rt21	20	8	1	24	5	1	1492
2.6.31.12-rt21	80	9	1	147	4	1	29

Table 4.3: Signaltest results for 5 threads, 100000 loops, μs

		No load			Full load		
Kernel	Priority	Avg.	Min.	Max.	Avg.	Min.	Max
2.6.31.12	20	46	43	239	43	29	1258
2.6.31.12-rt21	30	20	33	66	40	14	149
2.6.31.12-rt21	80	32	30	55	38	29	77

Table 4.4: Signaltest results for 10 threads, 100000 loops, μs

		No load			Full load		
Kernel	Priority	Avg.	Min.	Max.	Avg.	Min.	Max
2.6.31.12	20	95	88	297	85	60	3725
2.6.31.12-rt21	20	74	62	112	85	27	190
2.6.31.12-rt21	80	74	65	95	83	65	135

them over the interrupt threads. The most important change is the maximum latency; for a hard real-time system, this value must be bounded. Under the standard kernel, latencies in the millisecond range appear common. However, in the `PREEMPT_RT` kernel, latencies never exceed 1.5 milliseconds, and rarely exceed 100 microseconds. It is also interesting to note that while running at priority 80 the latency only once exceeded 147 microseconds. Further experimentation showed that this peak was not due to the task being blocked by some low-level activity, but rather due to the fact that tasks in Cyclictest were sleeping. This caused cores to move in and out of idle states, which caused the latency shown. This is shown in our results by the fact that maximum latencies for Cyclictest were consistently lower at full load than at no load.

Chapter 5

ChronOS Real-Time Linux

5.1 Objective

The objective of ChronOS is to provide a real-time operating system with a flexible, modular, extensible framework for multicore real-time task scheduling. To accomplish this, we augment the 2.6.31.14 Linux kernel with the `PREEMPT_RT` patch, and then extend it with our own scheduling framework and APIs. The `PREEMPT_RT` patch provides the necessary real-time operating system capabilities. Our scheduling framework, called ChronOS, builds on this to provide a wide range of advanced real-time task scheduling and resource management policies. The resulting platform is suitable for academic research as both the `PREEMPT_RT` patch and all ChronOS code are released under the GNU general public license version 2 [50].

This chapter describes the changes ChronOS makes to the Linux kernel to provide the framework, the capabilities of the framework, and the user space APIs used to interact with it.

5.2 Linux Scheduling

Linux scheduling is divided up into two classes — real-time scheduling and time-sharing scheduling. There are 140 priorities spanning both classes. Priorities [0, 99] are real-time priorities while [100, 140] are time-sharing priorities, generally referred to as “nice” values. The real-time priorities are reversed in user space to make higher priorities higher. Nice values are also adjusted so that instead of specifying a value in [100, 140], a user specifies a value in [-20, 20]. Time-sharing scheduling, referred to as `SCHED_NORMAL` or `SCHED_OTHER` is the default policy, and is implemented in Linux by the Completely Fair Scheduler, or CFS. Since ChronOS effects no changes in CFS, we henceforth ignore it.

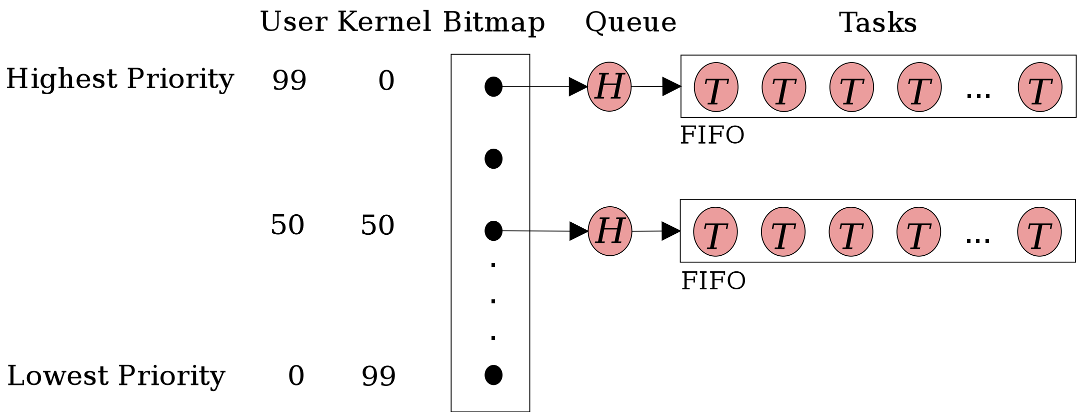


Figure 5.1: Linux priority bitmap and task queues

Linux supports two real-time scheduling policies — `SCHED_FIFO` and `SCHED_RR`. These have briefly been described in Chapter 4, and the behavior of `SCHED_FIFO` in a multicore environment is described in Chapter 6, but a implementation-level description of the workings of `SCHED_FIFO` on a single core is given here, since the implementation of ChronOS builds directly on `SCHED_FIFO`.

Each CPU in Linux has a `runqueue`, which is the scheduling abstraction for that entire CPU. On each `runqueue`, `SCHED_FIFO` is implemented as a bitmap and an associated array of queues. This is shown in Figure 5.1. The bit corresponding to the priority of the task is marked whenever a new task is added to the `runqueue`. `SCHED_FIFO` can then schedule in $O(1)$ time by finding the first marked bit in the bitmap, and executing the first task in the queue associated with that bit.

ChronOS directly extends `SCHED_FIFO`. In order to do this, we add several data structures, which are here described. In order to facilitate scheduling of ChronOS tasks without interfering with the existing priority system, we implement a second task queue for each priority, called the `chronos_queue`. It is sometimes abbreviated in figures as the CRT-RQ. This queue contains a possibly empty subset of the tasks in the standard task queue. This is shown as in Figure 5.2. The `chronos_queue` operates in parallel with the Linux queue. When a task enters the system, changes priority, or changes `runqueue`, it is placed on the FIFO ordered Linux task queue for its `runqueue` and priority. If, at some time during its execution, this task performs a `begin_rt_seg` operation, it is also placed on the `chronos_queue` for its priority and its scheduling policy is changed from `SCHED_FIFO` to `SCHED_CHRONOS`. Depending on the scheduling algorithm, tasks may be stored on the `chronos_queue` in a variety of orders, including FIFO and deadline-ordered. Once a thread is inside a real-time segment, it cannot change its priority. If the task is migrated and therefore changes `runqueue`, it is removed from the `chronos_queue` of the first `runqueue` and inserted on the `chronos_queue` of the second. When the thread executes an `end_rt_seg` operation, it is removed from the `chronos_queue`. ChronOS scheduling is performed within these `chronos_queues`.

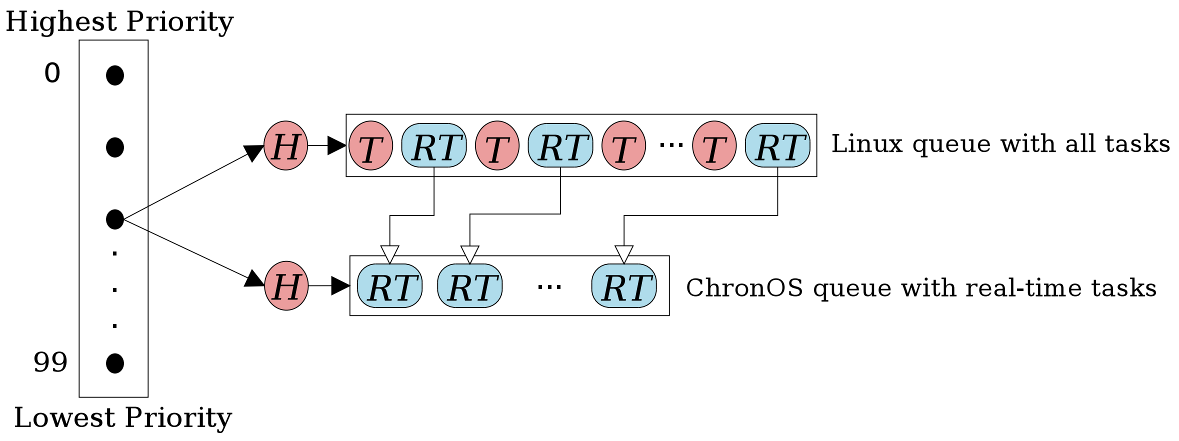

Highest Priority
0
99
Lowest Priority
H
T
RT
Linux queue with all tasks
ChronOS queue with real-time tasks

Figure 5.2: ChronOS task queues

5.3 Data Structures and Operations

5.3.1 Data Structures

We also need to store timing constraints for each task, as well as other objects needed for scheduling, such as the pointers for the `chronos_queue`. To do this, we add an object containing this data to the task descriptor of each task in Linux. This object is shown in Figure 5.3. The four timespec structures represent the deadline, deadline adjusted for dependencies, period, and remaining execution time of the task. `exec_time` is the task's WCET which, together with `max_util` describes the task's downward-step TUF. The `local_ivd` and `global_ivd` represent the local and global value densities of the task. These are inverted when stored because the utility will almost always be a smaller value than the remaining execution time (in microseconds), and therefore the fraction could not be accurately represented as an integer. `seg_start_us` stores the thread's total execution time at the time of the `begin_rt_seg` operation. This is necessary for accurately computing the remaining execution time of a thread with multiple segments. The `task_list` objects are used to store the thread on the local and global task queues, while the `list` objects are provided to allow scheduling algorithms to build their own lists. `requested_resource` holds the address of any resource requested, while `dep` stores the owner of that resource, if any. If timing constraints are provided for the segment, these are stored in `abortinfo`. The `flags` fields stores state information about the task, such as whether there are any pending operations on it, whether it has been aborted, or if it has deadlocked with another task over a resource. The `cpu` field stores the last processor core the segment was executed on. While Linux provides a similar value, it does not differentiate between normal thread execution and real-time segment execution. Therefore, by setting this value to an invalid number at the creation of a segment, we can tell whether a segment has begun execution, and thereby gauge its eligibility for migration. The details of the `rt_graph` structure are described by Garyali and are therefore omitted here [52].

```
struct rt_info {
    /* Timing Constraints */
    struct timespec deadline;       /* Monotonic time */
    struct timespec temp_deadline;  /* Monotonic time */
    struct timespec period;         /* Relative time */
    struct timespec left;           /* Relative time */
    unsigned long exec_time;        /* WCET, us */
    int max_util;
    long local_ivd;
    long global_ivd;
    unsigned int seg_start_us;

    /* List objects for various queues */
    struct list_head task_list[2];
    struct list_head list[SCHED_LISTS];

    /* DAG used by x-GUA class of algorithms */
    struct rt_graph graph;

    /* Lock information */
    struct mutex_head *requested_resource;
    struct rt_info *dep;

    /* Abort information */
    struct abort_info abortinfo;

    /* Task state information */
    unsigned char flags;
    int cpu;
};
```

Figure 5.3: Data structure appended to the task descriptor in ChronOS

ChronOS also creates an `rt_data` structure, which is used any time it is necessary to pass timing constraints between functions. This data structure is available in both the user space and kernel space, and is used to pass timing constraints to the kernel in `begin_rt_seg`. This structure is shown below.

```
struct rt_data {
    int tid;
    int prio;
    unsigned long exec_time;
    int max_util;
    struct timespec *deadline;
    struct timespec *period;
};
```

Figure 5.4: Data structure for holding timing constraints in ChronOS

5.3.2 Operations

ChronOS provides eight operations to the user space in the form of a dynamically linked shared object library. These operations are listed below with the associated function arguments listed for each.

`begin_rt_seg()`

This function is used to begin a real-time segment and provide timing constraints for a task. Several variants of this operation are provided. If `tid` is 0, the operation is performed on the calling thread. The real-time segment will have a priority of `prio`.

```
long begin_rt_seg_basic(int tid, int prio, struct timespec* deadline,
                 struct timespec* period);
long begin_rt_seg(int tid, int prio, struct timespec* deadline,
                 struct timespec* period, unsigned long exec_time,
                 int max_util);
```

`end_rt_seg()`

This function is used to end a real-time segment. If `tid` is 0, the operation will be performed on the calling thread. The thread will have a priority of `prio` after the end of the real-time segment.

```
long end_rt_seg(int tid, int prio);
```

`add_abort_handler()`

This function is used to provide timing constraints for an abort handler used by a real-time segment. If `tid` is 0, the operation will be performed on the calling thread.

```
long add_abort_handler(int tid, struct timespec *deadline,
                       unsigned long exec_time, int max_util);
```

`chronos_mutex_init()`

This function is used to declare a new scheduler-managed mutex.

```
long chronos_mutex_init(chronos_mutex_t *m);
```

`chronos_mutex_lock()`

This function is used to lock a scheduler-managed mutex to the system. If the calling thread is already the owner of the lock, the call returns success.

```
long chronos_mutex_lock(chronos_mutex_t *m);
```

`chronos_mutex_unlock()`

This function is used to unlock a scheduler-managed mutex. If the calling thread is not the owner of the lock, the call returns failure.

```
long chronos_mutex_unlock(chronos_mutex_t *m);
```

`chronos_mutex_destroy()`

This function is used to remove an existing scheduler-managed mutex from the system.

```
long chronos_mutex_destroy(chronos_mutex_t *m);
```

`setscheduler()`

This function is used to set the scheduling algorithm for a specified set of cores. `cpus` is a bitmask of the cores to be set. If the scheduler specified is a global scheduling algorithm, the scheduling domain is created at `prio` priority, otherwise it is ignored.

```
long set_scheduler(int scheduler, int prio, unsigned long cpus)
```

The first seven operations are presented to the user space in the form of two multiplexed Linux system calls, named `do_rt_seg` and `do_chronos_mutex`. The last is its own system call with the same name as the function. Of these operations, `setscheduler`, `init_chronos_mutex`, and `destroy_chronos_mutex` should never be called while any tasks are in scheduling segments.

Since ChronOS is an event-based platform, the real-time scheduler is invoked in after certain events. Four of these events correspond with operations mentioned above:

- The beginning of a real-time segment (`begin_rt_seg`)
- The end of a real-time segment (`end_rt_seg`)
- The request of a scheduler-managed resource (`lock_chronos_mutex`)
- The release of a scheduler managed resource (`unlock_chronos_mutex`)

However, because we are running in Linux and therefore subject to its priority scheme, there is another event that which we qualify as a scheduling event. This event is when higher priority Linux real-time task leaves the system, making ChronOS tasks the highest priority tasks in the system.

5.4 Scheduler Modules

All scheduling algorithms in ChronOS are implemented as Linux kernel modules for flexibility. Each module has several attributes, including a name, an ID number, a sorting key specifying its queue ordering method, and a scheduling algorithm. Modules are divided up into two classes: local or uniprocessor scheduling algorithm and global scheduling algorithms. All scheduling algorithms are added to the algorithm list when their module is loaded.

Local scheduling algorithm have only the information listed above. When a `setscheduler` call is made that requests a local scheduler, the system call searches through the list and finds the algorithm, and then sets it as the scheduler on each `runqueue`. This means that all ChronOS tasks on that `runqueue` will be scheduled with this algorithm, regardless of their priority, and all tasks in real-time segments will be inserted at the correct position on the `chronos_queue` based on the algorithm's sorting key. It is possible to set different local schedulers for different cores.

A global scheduling algorithm contains two additional pieces of information. First, it contains the ID of a local scheduling algorithm. This is because all global schedulers in ChronOS are implemented as hierarchical schedulers. Second, it contains a pointer to a global scheduling

```
struct global_sched_domain {
    /* The global scheduler */
    struct rt_sched_global *scheduler;
    /* The global task list */
    struct list_head global_task_list;
    /* The CPUs in this domain */
    cpumask_t global_sched_mask;
    /* Global scheduling priority in this domain */
    int prio;
    /* Task list lock */
    raw_spinlock_t global_task_list_lock;
    /* Scheduling lock */
    mcs_lock_t global_sched_lock;
    /* Current task count */
    atomic_t tasks;
};
```

Figure 5.5: Global domain structure in ChronOS

architecture. Each architecture is a set of functions and data structures used to provide a correctly sorted global queue to the scheduler and implement the scheduling decisions made. There are two scheduling architectures available in ChronOS, which will be described later.

Global scheduling operates in domains. A domain is a set of cores grouped together under the same instance of a global scheduler. As such, each domain must have its own copy of all data objects necessary to perform global scheduling. Figure 5.5 shows the global domain data structure.

Each domain contains a pointer to the scheduler used in that domain, a global task list for the tasks in that domain and an associated lock, a mask of the cores in the domain, a lock for the scheduler, and count of the tasks currently in the domain. Additionally, each domain exists only at a single priority. This is necessary to prevent the existing Linux priority system from interfering with global scheduling.

Since each domain may contain any non-empty subset of the system's cores, it is possible to have a variety of domains coexisting on a system. It is assumed that the application creating these domains has already specified the affinities of the tasks correctly using existing Linux operations such as `sched_setaffinity`.

When a `setscheduler` call is made that requests a global scheduler, the call first searches through the scheduler list and finds both the global and local scheduler specified. It then creates an empty domain and sets the global scheduler as the scheduler for that domain. Each core specified for the new domain is then removed from its current domain, if any, and placed in the new domain. The local scheduler is then set as such for each `runqueue`.

5.5 Uniprocessor Scheduling

The simplest form of scheduling available in ChronOS is uniprocessor scheduling. Uniprocessor scheduling on ChronOS is illustrated in Figure 5.6 using EDF. The application first calls `sched_setscheduler`, to set EDF as the scheduler for this processor. ChronOS checks through the list of schedulers and finds EDF and makes it as the scheduler for the processor. As tasks begin segments, they are placed in their priority's `chronos_queue` at their deadline position. They are also inserted at the tail of their priority's Linux task queue. When a scheduling event occurs, EDF selects the first (earliest deadline) task in the `chronos_queue`. This task is then moved to the head of the Linux task queue. `SCHED_FIFO` is then called, and selects the first task in the Linux task queue for execution.

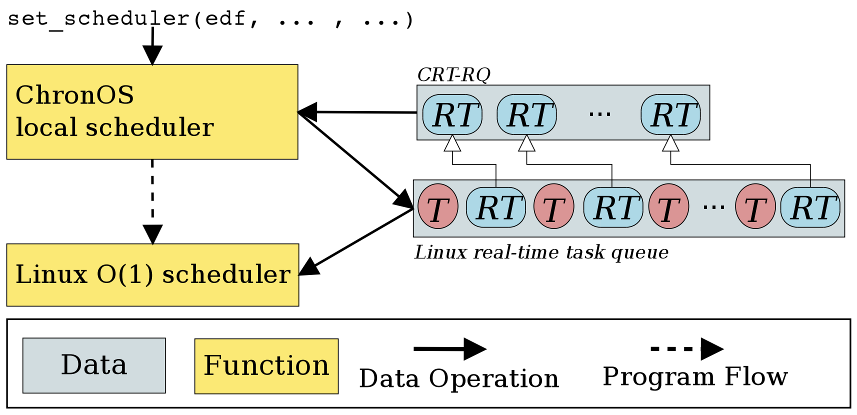


Figure 5.6: Uniprocessor scheduling in ChronOS

This has an important ramification. The execution order of normal `SCHED_FIFO` tasks is unaltered, since their relative ordering is unchanged. While `SCHED_CHRONOS` tasks may be moved in front of `SCHED_FIFO` tasks, these tasks are never moved in relation to each other. Therefore, we can say that `SCHED_CHRONOS` tasks effectively reside between their own priority and the priority above them; if a `SCHED_CHRONOS` tasks is placed at priority 50, it will execute only after all `SCHED_FIFO` tasks at priority 51, but before all `SCHED_FIFO` tasks at priority 50. We can therefore claim that we do not alter the existing POSIX-specified scheduling behavior in Linux.

5.5.1 Priorities

Since scheduling is still performed within fixed priority bands, is it necessary to place any set of tasks which should be scheduled together under a specific scheduling policy at the same priority. However, it is possible to place multiple sets of `SCHED_CHRONOS` tasks at multiple priority levels on the same processor when using uniprocessor scheduling. The sets will be scheduled in priority order, and each set will be scheduled under the same policy. However, it is not possible to place different sets of tasks at different priorities on the same processor and schedule them with different algorithms.

Both `begin_rt_seg` and `end_rt_seg` take priorities as input variables, and the priority of the thread is switched to this value during the call. This allows the following three execution patterns; first, a task can end a segment and increase its priority. This allows it to execute some action that requires freedom from preemption by other tasks in the set of tasks at the previous segment's priority level. A simple example is a periodic task waiting for its next period. If the task does not increase its priority and other `SCHED_CHRONOS` tasks exist at the priority, it will be preempted immediately upon completion of the `end_rt_seg` call, and will not begin sleeping for a potentially unbounded amount of time. Furthermore, even if it is allowed to begin sleeping immediately, it will still be a `SCHED_FIFO` task when it awakes, and therefore will not be able to begin its next segment at the correct time. This task would then reduce its priority in the `begin_rt_seg` for its next segment.

Second, a task can finish a real-time segment and reduce its priority. This is used when a task needs to perform some action which should not interfere with any of the other tasks at the segment's priority level.

Third, a segment could, instead of calling `end_rt_seg`, call `begin_rt_seg`, thereby starting a new segment. This is useful when two segments are directly adjacent to each other. While the same effect could be achieved by calling `end_rt_seg` with a priority higher than the segment's priority, and then `begin_rt_seg` at the original segment's priority, calling `begin_rt_seg` directly incurs significantly lower overhead.

5.5.2 Partitioned Scheduling

Partitioned scheduling is a straightforward extension of uniprocessor scheduling. Under uniprocessor scheduling, we rely on OS-supplied functionality to assign all tasks to the core specified. Partitioned scheduling can be performed on ChronOS by partitioned the tasks offline, assigning them to their cores using the same OS-supplied functionality, and then setting a uniprocessor scheduling algorithm as the algorithm for each core.

5.6 Global Scheduling

Almost all non-partitioned multicore scheduling algorithms are global scheduling algorithms. In a global scheduling approach, rather than placing tasks in per-core queues, tasks are placed in a single global queue and scheduling decisions are made based on this queue. Some

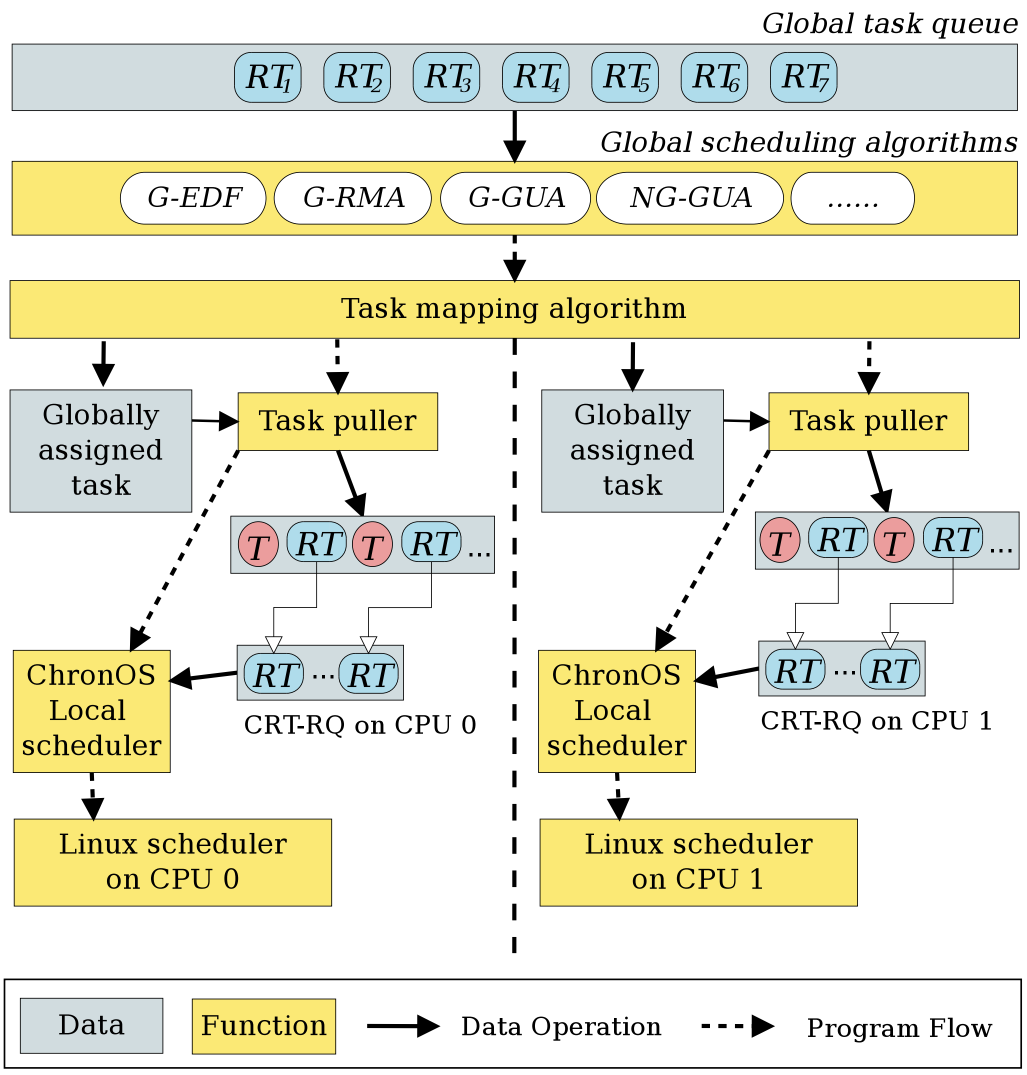


Figure 5.7: Global scheduling in ChronOS

global scheduling algorithms such as G-EDF, PFair, G-GUA, NG-GUA, and LLREF make scheduling decisions for each core at each scheduling event. Others, such as G-NP-EDF, can be implemented in such a way that each core makes a scheduling decision only for itself based on the global queue. The global scheduling approach used in ChronOS is shown in Figure 5.7 and described below.

First, a call to `setscheduler` is made by the application. This call specifies a global scheduling algorithm and a set of cores, in this case two. ChronOS searches through the scheduler list, finds the global scheduling algorithm and its associated local algorithm, and then creates a new scheduling domain covering the two cores. The global algorithm is set as the scheduler for the domain, and the local scheduling algorithms are set as the uniprocessor schedulers for each individual core. As tasks enter the system, they are placed both in a core's local queue and in the global queue. When a scheduling event occurs, the scheduler selects a set of tasks from the global queue for execution. These tasks are then assigned to tasks by a mapping algorithm. The task assigned to each core is placed in the "globally assigned task" field. The execution is then passed to the individual cores. On each cores, if the selected task does not already reside on the core, it is migrated there by the task puller. The uniprocessor scheduling algorithm is then called on each core, and execution continues as described in the previous section.

5.6.1 Priorities

Unlike uniprocessor scheduling, global scheduling places all tasks in a single queue. Since all tasks are scheduled together, in order to maintain fixed priority scheduling all the tasks being scheduled must be at the same priority. Therefore, each scheduling domain is given a priority, and all segments started on the domain are executed at this priority. This means that at any point, a core can check if it should schedule globally by checking if if the current highest priority task on its `runqueue` is equal to or less than the domain's priority and if there are any tasks in the global queue.

5.6.2 Prescheduling

Under some scheduling algorithms, it is possible under some conditions for the correct task for a core to be selected without looking at the global queue. The simplest example of this under non-preemptible scheduling. If segments are not allowed to be preempted by other segments at the same or lesser priorities, then before looking at the global queue, the local queue can first be checked for segments which have begun their execution. If such a segment is found, it should be scheduled, regardless of the state of the global queue. Such as selection process is called prescheduling.

5.6.3 Inter-Processor Communication

At some points during global scheduling, it becomes necessary to communicate between cores to make sure scheduling decisions are enacted. For this, we use Inter-Processor Interrupts (IPI). In Linux, the scheduler is called at the end of every interrupt. The scheduling IPIs we send between cores are dummy interrupts, which do nothing but force an scheduling event to occur on the remote core.

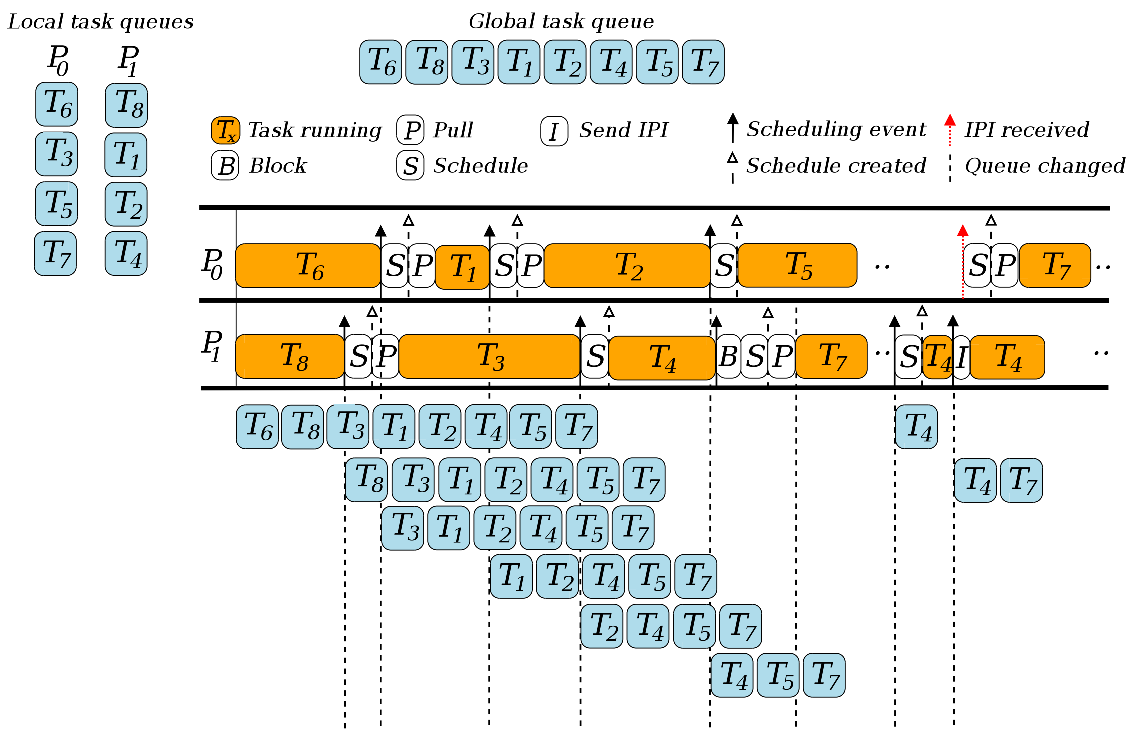


Figure 5.8: The application concurrent scheduling architecture

5.6.4 Scheduling Architectures

Rather than forcing the user to specify the details of managing the global queue, mapping tasks to cores, and invoking the global scheduler, ChronOS provides the ability to define a scheduling architecture which takes care of these details. ChronOS provides two scheduling architectures by default. These are described in the following two sections.

Application Concurrent Architecture

The simplest scheduling architecture available in ChronOS is the application concurrent scheduling architecture. The application concurrent scheduling model assumes that each core is scheduling only for itself and therefore that only one task will be selected at each scheduling event. This task should therefore be mapped to calling core. The name "application concurrent" is derived from the fact that global scheduling occurs without interrupting the application on remote cores. This architecture is illustrated in Figure 5.8. For the sake of demonstration, we will say that the scheduling algorithm being used is global first-in first-out (G-FIFO). G-FIFO executes the task at the head of the queue.

At the beginning of the sample trace, T_6 and T_8 are executing on processors P_0 and P_1 respectively. When T_8 finishes, P_1 schedules and picks the next task in the global queue,

T_3. Since this task is on P_0, it is migrated with a pull operation. This scheduling decision does not affect the execution of T_6 on P_0. Similarly, when T_6 finishes on P_0, it selects T_1 as the next task and migrates it without affecting P_1. This is repeated when T_1 finishes and T_2 is selected. When T_3 finishes execution, P_1 selects T_4 as the next task to execute. Since T_4 is already on P_1, no migration is necessary. T_2 and T_4 finish their execution at almost the same instant. In the concurrent architecture, each processor wanting to add a task from the global queue, remove a task from the global queue, or schedule must lock the task list lock. Since P_0 finishes T_2 slightly before P_1 finishes T_4, it acquires the lock first. P_1 therefore blocks waiting for the lock. P_0 selects T_5. P_1 then acquires the lock, selects T_7, and migrates it from P_0. Once T_5 and T_7 have finished, there are no real-time tasks in the system. At some time later, a second job of T_4 arrives in the system, and is scheduled on T_1, the last processor it executed on. A short time after that, a second job of T_7 arrives on P_1. Since P_0 is unoccupied, it should execute T_7. However, since T_7 arrives on P_1, no scheduling will occur on P_0. We solve this by forcing a scheduling event on P_0 by sending it a scheduling IPI. P_0 receives the IPI, schedules, selects T_7, migrates it from P_1, and executes it.

In this example, we have ignored any additional scheduling events that occur during the execution of a task, such as events due to the presence of interrupt tasks. In such an event, the prescheduling function would select the previously executing task after the higher priority task leaves the system.

Stop-The-World Architecture

While the application concurrent provides a simple framework for global scheduling, there are a large number of schedulers which cannot be implemented under it. These include G-EDF, G-GUA, and NG-GUA. In these algorithms, the schedule for the entire domain is computed at each scheduling event. Furthermore, some algorithms like G-GUA and NG-GUA provide global scheduling for resource-dependent tasks. For these schedulers, we need a new scheduling architecture. To accomplish this, we implement the stop-the-world architecture. This architecture is illustrated in Figure 5.9.

In this two processor example, we will assume that the scheduling algorithm selects the task with the highest number of dependencies, and breaks ties based on some undefined criteria. The figure shows the dependency relationship between tasks, the state of the global queue, and the current processor localities of tasks. Task T_4 is waiting for a resource held by T_2, which is waiting on T_1. Similarly, T_6 is blocked on T_5. No tasks are dependent on T_3. Tasks T_6, T_1, and T_4 are currently on processor P_0, and T_2, T_3, and T_5 are on P_1.

At the beginning of our trace, P_0 is executing T_1 and P_1 is executing T_5. When P_1 finishes executing T_5, it sends an IPI to P_0 to inform it that a scheduling decision is being made. In the stop-the-world architecture, there is a global scheduling lock used to guard the schedule. If a processor attempts to schedule and finds that the lock is already held, it blocks until the lock is free and then instead of scheduling, looks for the task the scheduling processor

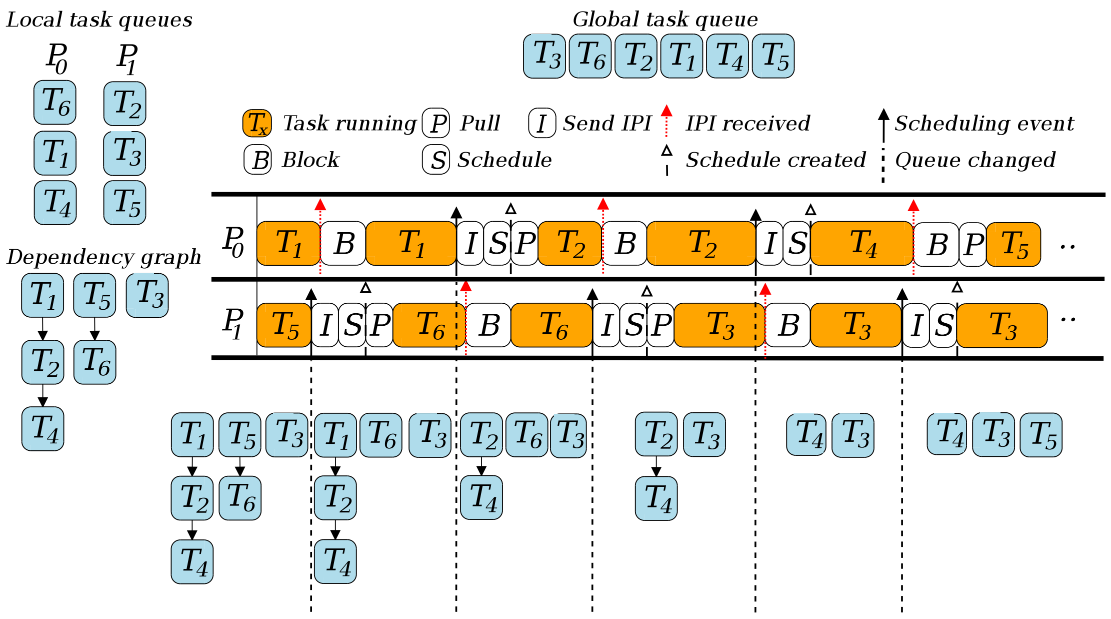


Figure 5.9: The stop-the-world global scheduling architecture

has selected for it. In this case, P_0 blocks until P_1 has computed the new schedule for the system. In this case, the only change is that P_1 has selected T_6 for execution. P_1 migrates T_6 from P_0 and begins executing it. Similarly, when P_0 finishes executing T_1, it sends an IPI to P_1, schedules for both processors, sends a task assignment to the waiting P_1, migrates T_2, and begins execution. A similar sequence of events occurs when P_1 finishes executing T_6. When P_0 finishes executing T_2, a similar process again occurs. This time, the task selected for P_0 is T_4, which is already on the processor, so no migration occurs. Finally, during the execution of T_3 on P_1, a second instance of T_5 arrives. This causes a scheduling event on P_1, which causes a IPI to be sent to P_0. The tie-breaking rule selects T_3 and T_5 as the best tasks for execution. Since T_3 is already executing on P_1, it is left there. P_1 receives T_5 as its task assignment and stops executing T_4, instead migrating and executing T_5.

Under schemes like G-EDF, no task is ever exempt from preemption, and so there is little use for a prescheduling algorithm. We do however use prescheduling in conjunction with abortive stop-the-world algorithms such as G-GUA and NG-GUA to remove aborted tasks from the system.

5.6.5 Mapping Tasks

All of the scheduler we consider are mapping-agnostic. This means that they select at most m tasks to execute, but do not actually care which task executes on which processor or core. For example, G-EDF simply select the m lowest deadline tasks to execute. It makes no statements about which task should execute on which core, but all cores are assumed to be identical.

While this drastically simplifies scheduling, it creates a new problem, referred to as task mapping. The mapping problem is the problem of assigning m tasks to m cores in an optimal way. We define an optimal mapping algorithm as an algorithm which can make an assignment that minimizes the overhead incurred by the system from factors such as task migrations and cache misses. Optimal can therefore describe a wide range of things, depending on the hardware platform. For example, on a simple dual-core processor with a shared cache, an optimal mapping algorithm might guarantee that no tasks are migrated unnecessarily. However, on a complex NUMA system, an optimal mapping algorithm would have to not only minimize migrations, but when forced to migrate a task, consider various possible migration patterns based on distance between cores and the specific architecture of the system. For example, in a multicore, multiprocessor system in which all cores on a processor share cache, it is better to migrate a task between cores in the same processor than between processors.

We define three different levels of optimality which a mapping algorithm may provide. These are migration-optimal, execution-optimal, and distance-optimal.

Definition 1 *Consider a scheduling algorithm which returns a k task schedule for an m core system, where $0 < k \leq m$. To assign these k tasks to m cores, some j minimum number of migrations must be made, where $0 \leq j < k$. We say that a mapping algorithm is migration-optimal if it maps all schedules with exactly j migrations.*

Definition 2 *Consider a scheduling algorithm which returns a k task schedule for an m core system, where $0 < k \leq m$. Of these k tasks, j of them will already be executing on a core in the system which they are eligible for mapping to, where $0 \leq j \leq k$. We say that a mapping algorithm is execution-optimal if none of these j tasks are migrated.*

Definition 3 *Consider a scheduling algorithm which returns a k task schedule for an m core system, where $0 < k \leq m$. By Definition 1, there are some j migrations which must be performed. We say a mapping algorithm is distance-optimal if the set of migration paths with the least total cost is chosen out of all available migration paths, even if the minimum cost set of migration paths includes more than j paths.*

A distance-optimal mapping algorithm will guarantee a minimum possible total migration distance, based on some weighting of inter-processor edges derived from cache structure and inter-processor memory access times. This can be constructed as a variant of the shortest-path problem; in this variant, there are k start and end nodes, and a path must be computed from each start node to an end node in such a way that the total distance of all paths is minimized. Paths are allowed to have a 0 length, implying no migration. Up to k start nodes may coincide, but no end nodes will coincide.

None of these three definitions imply either of the other two. It is possible to have a migration-optimal which migrates running tasks, or migrates tasks across longer than necessary paths. Similarly, it is possible to have an algorithm that migrates no running tasks but performs unnecessary migrations or migrates across longer than necessary paths. On many systems, it is likely that distance optimal would imply both migration- and execution-optimality, but it is possible to construct systems in which this is not the case. Consider a system with four cores and exponential migration costs — i.e. the cost of migrating from any processor to its neighbor is 1, while the cost to migrate to a processor two away is 3 and the cost to migrate to a processor 3 away is 9. The cost to migrate a running task is one more than the cost to migrate a non-running task, i.e. 2, 4, and 10. Assume four tasks have been selected for scheduling. Two of these tasks reside on processor 0, while one resides on processor 1 and one resides on processor two. Of these, one of the tasks on processor 0 and the tasks on cores 1 and 2 are currently executing. Therefore, the minimum number of migrations is one, and three of the four tasks are already migrating. This means that in a migration- and execution-optimal algorithm, the extra task on processor 0 would be moved to processor 3. However, this is a distance of 3, and so has a cost of 9. By migrating the non-running task from processor 0 to processor 1, the task from processor 1 to processor 2, and the task from processor 2 to processor 3, it is possible to map the schedule with a total cost of $1 + 2 + 2 = 5$.

We apply a mapping algorithm which is migration- and execution-optimal, but not distance-optimal. The algorithm we use is a modified variant of the three-pass algorithm described by Garyali [52]. Our algorithm makes two passes over the list of tasks returned by the scheduler. In the first pass, for each core, the mapper loops through the task list. If it finds a task that is the currently running task on the core, that task is mapped to the core, removed from the task list, the core is marked as mapped, and the mapper continues to the next core. This guarantees execution optimality. If the mapper finds a task that currently resides on a core, but is not currently executing, this task is marked as the "backup" task. If all tasks have been consider and a backup task was found, it is mapped to the core, removed from the list, and the core is marked as mapped. This guarantees migration optimality.

Once this first pass has been completed for all cores, we are left with a list of cores without a mapping and a list of tasks which have not been mapped. These remaining tasks are then assigned to cores in FIFO manner. Pseudocode for the mapping algorithm is shown in Algorithm 1

An example mapping is shown in Figure 5.10. Here, we have a quad-core system with nine tasks in it. The scheduler has returned four tasks — RT_5, RT_8, RT_1, and RT_4, in that order. The mapper first looks for a task for rq_0, and finds RT_1, its current running task. It then looks for a task for rq_1, and finds RT_5, its current task. When looking for a task for rq_2, no task is found to be the current running task, but RT_8 currently resides on rq_2, so it is selected. Finally, no task is found for rq_3, since RT_4 is the only remaining task in the system, and it is on rq_0. In the second, pass RT_4 is mapped to rq_3.

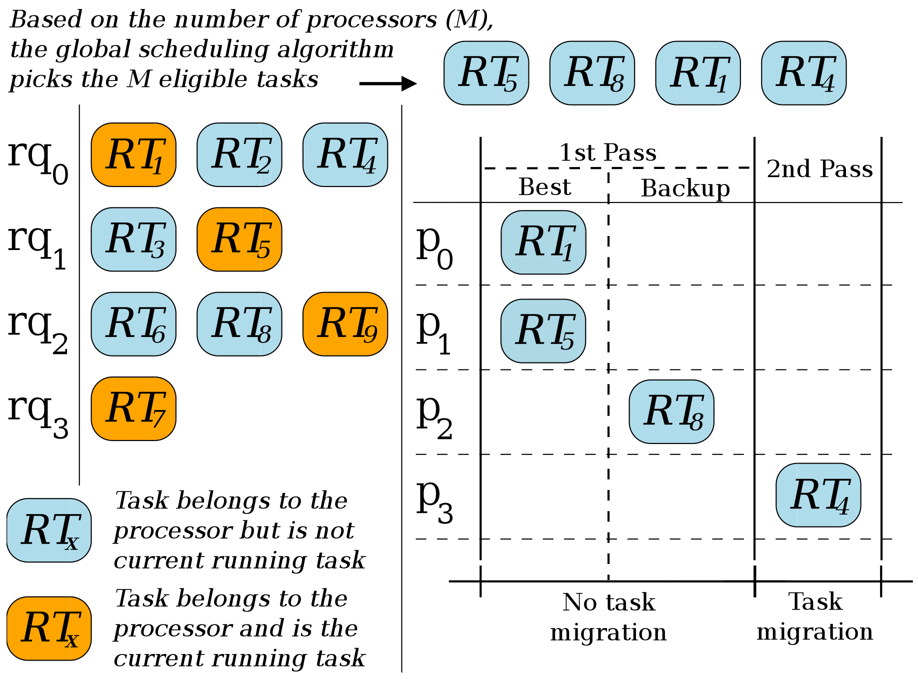

Based on the number of processors (M), the global scheduling algorithm picks the M eligible tasks
RT_5 RT_8 RT_1 RT_4
rq_0 RT_1 RT_2 RT_4
rq_1 RT_3 RT_5
rq_2 RT_6 RT_8 RT_9
rq_3 RT_7
RT_x Task belongs to the processor but is not current running task
RT_x Task belongs to the processor and is the current running task
1st Pass
Best
Backup
2nd Pass
p_0 RT_1
p_1 RT_5
p_2 RT_8
p_3 RT_4
No task migration
Task migration

Figure 5.10: Default task mapping algorithm in ChronOS

The asymptotic cost of this mapping algorithm is $O(m^2)$. However, under fixed or task-dynamic priority schedulers, this can be reduced significantly, since we know that at any scheduling event, at most one task will have entered or left the system. If only one task entered or left the system, and the priority relationships of all remaining tasks in the system have not changed, at most one task in the schedule may have changed, and therefore at most one core will not be mapped in the first pass. If we assume the worst case, namely that it is the first core that was not mapped, this implies that we will have to consider m tasks for both the first and second cores, $m - 1$ tasks for the thirds, and so on down to 2 tasks for core m. Therefore, for a fixed or job-fixed priority algorithm, we will have to consider at most $\frac{m^2+m}{2} + (m-1)$ tasks on the first pass, and 1 task on the second pass. Since $\frac{m^2+m}{2} + (m-1) + 1 = \frac{m^2+3m}{2} \leq m^2$ for all m, this is the asymptotic bound for mapping fixed and job-fixed priority schedulers. Based on this, we can also say that for fixed and job-fixed priority schedulers, we will never migrate more than one task at each scheduling event. For task-dynamic priority scheduling algorithms, we can only say that since all tasks must be on a core in the system, we will migrate at most $m - 1$ tasks.

5.6.6 Clustered Scheduling

Clustered scheduling is an approach devised to strike a balance between partitioned and global scheduling. In clustered scheduling, the system is broken up into multiple different

```
Input: TaskList;                                //An unordered list of tasks
Data: $Task_1 ... Task_m$;                      //Task mapped to each core
Data: CpuMask;                                  //A mask containing all cores
foreach Core P do
    Backup = NULL;
    foreach Task i in TaskList do
        if CurrentTask(P) = i then
            $Task_P$ = i;
            break;
        end
        if TaskCpu(i) = P then
            Backup = i;
        end
    end
    if $Task_P$ = NULL then
        $Task_P$ = Backup;
    end
    if $Task_P \neq$ NULL then
        RemoveTask($Task_P$, TaskList);
        UnsetCpu(P, CpuMask);
    end
end
foreach Core P in CpuMask do
    $Task_P$ = HeadOf(TaskList);
    if $Task_P$ = NULL then
        return;
    end
    RemoveTask($Task_P$, TaskList);
end
```

Algorithm 1: A execution-optimal task mapping algorithm

clusters, tasks are partitioned offline into clusters, and global scheduling is performed within each cluster. In ChronOS, clustered scheduling is performed by creating different scheduling domains with the same global scheduling algorithm. The methods for doing this have been described earlier in this chapter. As with partitioned scheduling, we rely on the user space application to set the core affinity through OS-supplied functions. However, rather than assigning the tasks to a specific core, the tasks are assigned to the same group of cores as the cluster.

Typically, it is desirable to build clusters along hardware-defined lines, such as memory access paths or shared cache [32]. This approach is taken by LITMUS^{RT}, which at the time of writing provides two levels of clustering: L2 cache and L3 cache. While this simplifies scheduler selection from the user space, it does limit the possibilities. In ChronOS, since clustered scheduling is performed by the same system as global scheduling domains, it is possible to use clusters of any size, and even use disjoint clusters. For example, on an 8-core system it is possible to place cores 1, 2, 5, and 8 in a cluster and cores 3, 4, 6, and 7 in another cluster. This places the decision of selecting a “good” clusters on the user, but also allows more flexibility.

5.7 Aborting Tasks

In previous versions of ChronOS, tasks were aborted using POSIX signals [52]. However, since the Linux kernel is not designed to send signals from within the scheduler, this method was never intended as a permanent solution, and has since been replaced.

Tasks are now aborted via shared memory. We create a character device housing a large memory buffer, which can be mmapped by user space applications. This buffer has as many bytes as there are allowed threads. When a thread with a thread id of k is aborted, the kth byte in the buffer is set to one. The user space application must therefore check this byte at some frequency to determine if it has been aborted. This provides a fast, lock-free method for aborting tasks.

Chapter 6

Scheduling Algorithms

In this thesis, we compare the scalability and performance of 16 algorithms which have been implemented in ChronOS. For simplicity, we break these algorithms down into three categories: traditional global real-time scheduling algorithms, utility-accrual global real-time scheduling algorithms, and partitioned real-time scheduling algorithms. We compare the algorithms based on their theoretical performance, implemented asymptotic cost of scheduling, queue insertion and task mapping, and other characteristics. In our analysis, we shall always consider an m core system with n tasks in it. Table 6.3 summarizes various costs and grouping of each algorithm. The algorithms used for partitioning and clustering are described in Chapter 8.

6.1 Linux SCHED_FIFO

The default real-time scheduling policy in Linux is `SCHED_FIFO`. Under `SCHED_FIFO`, a scheduled task continues to run until it either voluntarily yields the processor or is preempted by a real-time task with a higher priority. Rather than using a single queue, `SCHED_FIFO` provides system-wide strict real-time priority scheduling (SWSRTPS) through a push-pull system. To accomplish this, it must manage three specific cases:

- A task waking on the same runqueue as a higher priority task
- A task being preempted by an awakening higher priority task
- A task lowering its own priority below the priority of another task on the runqueue

Each of these three cases can, in some situations, require the migration of tasks to guarantee SWSRTPS. In the first two cases, this is done by pushing the lower priority task onto another runqueue, if there exists a runqueue executing a lower priority task. In the last case, the runqueue will check to see if there exists a waiting task on another runqueue with a higher priority than its own highest priority task. If such a task exists, it will pull the task to itself and start executing it. This push-pull mechanism is only used within the scope of the runqueue's scheduling domain, which is a subset of the system's runqueues. In the case of our systems, we allow the default kernel policy to select the appropriate global domains. In our cases, this policy made the global domains equivalent to physical processors.

This behavior allows `SCHED_FIFO` to provide $O(m)$ scheduling, since each runqueue can find its own highest priority task in $O(1)$ and needs to at most check m other runqueues for a push or a pull operation [51]. Since each core schedules for itself, mapping is $O(1)$.

6.2 Global FIFO

Global FIFO (G-FIFO) provides, as the name implies, a global first-in-first out policy. While this should be logically equivalent to `SCHED_FIFO` almost all the time, our G-FIFO algorithm is implemented using the concurrent architecture. In our algorithm, each task is, on arrival, inserted at the end of the global queue. After this, a scheduling event is invoked. During this event, the scheduler performs two tasks. First, it checks if it currently has a task that it has already begun to execute. If this is the case, this task is executed, since it must have arrived before all tasks currently on the queue. If no task is found, it then locks the queue, removes the first task, and begins to execute it.

Scheduling, task insertion, and mapping are therefore $O(1)$. In theory, this scheduling algorithm should perform nearly identically to `SCHED_FIFO`. It is included here as a useful measurement of the overhead of a single-queue implementation.

Leontyev and Anderson have shown that G-FIFO can provide bounded deadline tardiness [73]. It therefore can be considered a soft real-time scheduling algorithm.

6.3 Global RMS

Global rate monotonic scheduling is a simple extension of rate monotonic scheduling to m cores. Our algorithm is implemented on the stop-the-world architecture. Each arriving task is inserted as its period-ordered place in the global queue. At each scheduling event, the m first tasks are selected and mapped to cores.

As our global queue implementation is a doubly-linked list, insertion therefore takes $O(n)$ time. However, because the list is ordered, scheduling takes only $O(m)$ time. Since m tasks must be considered for mapping to m cores, mapping is $O(m^2)$.

Global rate monotonic scheduling has received significant effort from the research community, and at least three theoretical schedulability tests have been devised. Anderson et. al. have determined that a set of tasks in which the maximum per-task utilization is at most $m/(3m-2)$ is schedulable as long as the total utilization for the taskset is below $m^2/(3m-1)$ [5]. A second test was determined by Baruah and Goossens; they conclude that if the individual utilization of a set of tasks does not exceed 1/3, the taskset will be schedulable as long as the total utilization is not greater than $m/3$ [18]. A third and significantly more complex test was developed by Baker for sporadic tasks with arbitrary deadlines [11]. Under this test, a taskset is considered schedulable if for each task T_k, $m < k \leq n$, there exists a positive value μ such that

$$\sum_{i=1}^{k-1} \beta_{\mu,k}(i) \ \leq \ \mu \ \leq \ m\left(1 - \frac{e_k}{min(P_k, d_k)}\right) \tag{6.1}$$

Where $\lambda = \frac{m-\mu}{m-1}$ and $\beta_{\mu,k}(i)$ is defined as

$$\beta_{\mu,k}(i) \ = \ \begin{cases} \frac{e_i}{P_i}(1 \ + \ \frac{P_i - e_i}{d_k}) & \text{for } \lambda \geq \frac{e_i}{P_i} \\ \frac{e_i}{P_i}(1 \ + \ \frac{P_i - e_i}{d_k}) \ + \ \frac{d_i}{d_k}(\frac{e_i}{P_i} - \lambda) & \text{for } \lambda < \frac{e_i}{P_i} \end{cases} \tag{6.2}$$

For more specific cases, such as deadlines equal to periods, this algorithm can be simplified. Baker provides one such simplification in [10]. G-RMS does not bound tardiness, since by definition longer period tasks will be preempted in favor of lower period tasks, and can therefore be blocked indefinitely.

6.4 Global EDF

Global earliest deadline first (G-EDF) is a direct extension of the single-core earliest deadline first (EDF) policy. As with G-RMS, G-EDF is implemented with the stop-the-world architecture. Each arriving task is inserted in a deadline-ordered list. At each scheduling event, the first m tasks in the global queue are selected and mapped to the cores.

As the implementations of G-EDF is only one line different than that of G-RMS, insertions are also $O(n)$ and scheduling is $O(m)$. Mapping is performed in the same manner as G-RMS and therefore is $O(m^2)$, since G-EDF is a job-fixed priority algorithm.

A large number of schedulability tests have been developed for G-EDF. Unlike EDF, there is no simple utilization based feasibility test which can be applied to G-EDF. This is because of the "Dhall effect", which states that on an m core system, it is possible to produce an $m+1$ task taskset with a utilization arbitrarily close to 1 which cannot be scheduled [46]. A simple example taskset is shown in Table 6.1 for a four-core platform. Under both G-RMS and G-EDF, tasks 1-4 will be scheduled, which will not leave sufficient time to execute task 5 before its deadline.

Table 6.1: A periodic taskset for a four-core system that demonstrates the Dhall effect

Task	Period	WCET	Deadline
1	100	10	100
2	100	10	100
3	100	10	100
4	100	10	100
5	101	92	101

There are at least five major sufficient schedulability tests for G-EDF [31]. The two simplest of these are shown below, while the rest are omitted because of their length. One of the simplest tests was devised by Srinivasan and Baruah [98]. They find that G-EDF is able to schedule any set of periodic tasks with implicit deadlines in which no task has a utilization exceeding $m/(2m-1)$ and the total system utilization is not greater than $m^2/(2m-1)$.

Goossens et. al. have shown that G-EDF can schedule any taskset in which the total utilization does not exceed $m-(m-1)u_{max}$, where u_{max} is the utilization of the highest utilization task in the taskset [53]. This is commonly known as the GFB test.

Two other tests, referred to as BAK2 and BCL have been summarized by Baker in [12]. The fifth test is presented by Bertogna et. al. in [25].

G-EDF bounds deadline tardiness as long as the total utilization of the taskset does not exceed the number of cores. Therefore, for the soft real-time case, testing that $U \leq m$ is a sufficient test [45]. Devi and Anderson have found tardiness bounds for G-EDF and G-NP-EDF [45]. They find that for G-EDF, the tardiness bound for T_i is

$$\tau \leq \frac{(m-1)e_{max} - e_{min}}{m-(m-2)U_{max}} + e_i \tag{6.3}$$

A less pessimistic but higher complexity form of this bound is presented by Devi in [44]. He proves that a task T_i of a k task taskset will incur a maximum tardiness of

$$\tau \leq \frac{\sum_{n=1}^{\Delta} e_n - e_{min}}{m-\sum_{n=1}^{\Delta-1} U_n} + e_i \tag{6.4}$$

Where $\Delta = \lfloor \sum_{n=1}^{n\leq k} U_i \rfloor$. This bound was later improved by Erickson [48]. If we want the worst-case bound for any task in a taskset, we can substitute e_i with e_{max}. Both forms of this bound have one issue; as per-task weights increase in the first form, the bounds become highly pessimistic because $m-(m-2)*U_{max}$ approaches 2. Similarly, for the second form, if a few heavy tasks dominate the taskset, the weight of these tasks pushes $m-\sum_{n=1}^{n\leq m-1} U_n$

to 0. This is a known dilemma, and the authors state that the bounds are reasonable unless both m and the average per-task utilization are high.

6.5 Global Non-Preemptible EDF

Global non-preemptible EDF (G-NP-EDF) is a variation of G-EDF which disallows the preemption of a executing task by a task with a lower deadline. It is therefore a limited migration algorithm; tasks are allowed to migrate freely at job boundaries, but once a job begins to execute on a specific core, it is not allowed to migrate again until its completion. In ChronOS, G-NP-EDF is implemented via the concurrent scheduling architecture. On arrival, tasks are inserted on the deadline-ordered global queue. At each scheduling event, the runqueue first checks if it is already executing a task, and if so continues. If it is not, it locks the queue, dequeues the first task, and then migrates it to its own runqueue and begins execution. Insertion is therefore $O(n)$, while scheduling and mapping are $O(1)$.

Since G-NP-EDF is a direct extension of G-EDF and for a given taskset the initial schedules of both algorithms will be identical, it therefore follows that G-NP-EDF is subject to the "Dhall effect". However, G-NP-EDF suffers from another condition under which a taskset might not be schedulable which does not affect G-EDF. Table 6.2 shows a taskset which exposes this condition.

Table 6.2: A periodic taskset for a four-core system which is unschedulable under G-NP-EDF but schedulable under G-EDF

Task	Period	WCET	Deadline
1	100	51	100
2	100	51	100
3	100	51	100
4	100	51	100
5	1000	100	10000

The initial schedule generated by both G-EDF and G-NP-EDF will schedule tasks 1-4 on the first four cores. After that, both algorithms will schedule task 5 on core 1. Under G-EDF, task 5 will be preempted by task 1 every 100 time units, and will finish its execution at time 202, having been preempted twice. However, under G-NP-EDF, task 5 will not be preempted, meaning the second job of task 1 will not begin execution until time 151. Since task 1's execution time is 51, it therefore does not finish its execution before its deadline.

Exact feasibility of any non-preemptive scheduling algorithm has been previously shown to be NP-hard in the strong sense even on a uniprocessor system [64]. Only one sufficient schedulability test is known for G-NP-EDF. Baruah [16] has shown that for a factor

$V(T_i, T) = \frac{e(T_i)}{P(T_i) - e_{max}(T)}$, a taskset is feasible under G-NP-EDF if

$$\sum_{i=0}^{i \leq n} V(T_i, T) \leq m - (m-1)V_{max}(T) \tag{6.5}$$

This is similar to the GFB test for G-EDF, but not equivalent.

G-NP-EDF is also known to provide bounded deadline tardiness as long as the total utilization of the taskset does not exceed the m. Therefore, for the soft real-time case, testing that $U \leq m$ is a sufficient test [45]. Devi and Anderson have shown that for a task T_i, the tardiness bound under G-NP-EDF is

$$\tau \leq \frac{\sum_{n=1}^{\Delta+1} e_n + \sum_{n=1}^{m-\Delta-1} \beta_n - e_{min}}{m - \sum_{n=1}^{\Delta} U_n} + e_i \tag{6.6}$$

Where $\Delta = \left\lfloor \sum_{n=1}^{n \leq k} U_i \right\rfloor$ and β_i is the ith largest non-preemptible section. As with the G-EDF bound, there is a more pessimistic bound with a constant time computation cost [44]. This bound is

$$\tau \leq \frac{m \cdot e_{max} - e_{min}}{m - (m-1)U_{max}} + e_i \tag{6.7}$$

6.6 Global HVDF

Global highest value density first (G-HVDF) is the simplest fully preemptible heuristic utility accrual algorithm presented. It was first mentioned by Garyali in [52], and is first described here. G-HVDF is a direct result of the value density extension of the TUF concept described in Chapter 3. In G-HVDF, each task is assumed to have a downward step TUF. Each task is then assigned a local value density, which represents the ratio of its utility to its remaining execution time. Since value densities are calculated based on the remaining execution time, rather than the WCET, they must be recalculated every time the scheduler is run. G-HVDF also assumes each thread may provide an abort handler. Such abortable threads are aborted by the scheduler when they have passed their deadline.

G-HVDF is implemented using the stop-the-world architecture. On arrival, tasks are inserted into an unordered list. At each scheduling event, each core first checks if it has any aborted tasks in its runqueue. If aborted tasks are found, they are executed. If no tasks are found, the scheduler recalculates the value densities for all tasks and checks for tasks which have exceeded their deadlines. The m highest value density tasks are then selected and mapped to cores. Insertion is therefore $O(1)$ and scheduling is $O(n)$. While G-HVDF uses the same stop-the-world architecture as G-EDF and G-RMS, it is a job-dynamic priority algorithm and therefore mapping is $O(m^2)$.

No schedulability tests, either sufficient or exact, have ever been found for G-HVDF. Additionally, since the local value density of a task remains unchanged while the task is not executed and can only increase while the task is executing, the behavior of the G-HVDF is not unlike a fixed priority algorithm in some respects. However, since tasks are aborted, deadline tardiness is not a meaningful metric.

6.7 Global Non-Preemptible HVDF

Global non-preemptible highest value density first (G-NP-HVDF) is a non-preemptible version of G-HVDF. As with G-NP-EDF, it is therefore a limited migration algorithm; tasks are allowed to migrate freely at job boundaries, but once a job begins to execute on a specific core, it is not allowed to migrate again until its completion. In ChronOS, G-NP-HVDF is implemented via the concurrent scheduling architecture. On arrival, tasks are inserted on the LVD-ordered global queue. At each scheduling event, the runqueue first checks if it is already executing a task, and if so continues. If it is not, it locks the queue, dequeues the first task, and then migrates it to its own runqueue and begins execution. Insertion is therefore $O(n)$, while scheduling and mapping are $O(1)$.

G-NP-HVDF has two distinct differences from G-HVDF. First, since only the first item on the global queue is ever viewed by the scheduler, only this item may be aborted. This means that it is possible for a task to reside in the queue for an unbounded amount of time after having missed its deadline. Once the task begins execution, it is allowed to execute even after its deadline is past. This means that each task will either never be aborted, or will be aborted immediately after its call to `begin_rt_seg`. Second, since tasks LVDs are only ever consider before they begin execution, G-NP-HVDF may be considered a non-preemptible fixed priority scheduling algorithm.

As with G-HVDF, no sufficient or exact schedulability tests are known for G-NP-HVDF. Because it is a fixed priority algorithm and does not abort tasks immediately if they miss their deadline, under overloads lower priority tasks may incur unbounded deadline tardiness.

However, unlike G-HVDF, it is possible under some conditions to prove a lower utility accrual and deadline satisfaction bound for G-NP-HVDF. This is shown below.

For n periodic tasks $T_1,\ T_2,\ ...\ T_{n-1},\ T_n$ with deadlines equal to periods, ordered by decreasing value density (i.e. $LVD_1 \geq LVD_2\ ...\ LVD_{n-1} \geq LVD_n$), if

$$\max_{i=m+1}^{i\leq n}(e_i) \leq \min_{i=1}^{i\leq m}(P_i - e_i) \tag{6.8}$$

Then tasks T_1 to T_m will meet their deadlines and accrue utility. Since these are the highest LVD tasks in the system, this means at that we will accrue a minimum of $\frac{m}{n}$ of the total possible utility. We sketch the proof of this as follows:

Assume a task T_i where $1 \leq i \leq m$, and the above condition is true. Further assume that at some point during its execution, T_i fails to meet its deadline and therefore does not accrue utility. If there are less than m tasks in the system, G-NP-HVDF will schedule T_i at its arrival, and therefore it will meet its deadline. Therefore there must be more than m tasks in the system, and T_i must receive interference during its execution from another task. We define the critical instant of a task as the instant by which it must be scheduled to complete by its deadline, i.e. $R_i + P_i - e_i$. We know that between the arrival of T_i and its critical instant, at least m scheduling events occur due to task completions by equation 6.8. Furthermore, since

$$\max_{i=m+1}^{i \leq n}(e_i) \;\leq\; \min_{i=1}^{i \leq m}(P_i - e_i) \;\leq\; \min_{i=1}^{i \leq m}(P_i) \tag{6.9}$$

there are at most $m-1$ arrivals of tasks with a higher LVD than T_i. Therefore, T_i must be the highest LVD task available for scheduling at some scheduling event before its critical instant. Since we know at each scheduling event the highest LVD task is selected for execution, T_i must be selected, and therefore cannot fail to meet its deadline.

This proof can further be extended for any $k \leq m$. Therefore, we can say that if

$$\max_{i=k+1}^{i \leq n}(e_i) \;\leq\; \min_{i=1}^{i \leq k}(P_i - e_i) \tag{6.10}$$

Then tasks T_1 to T_k will meet their deadlines and accrue utility. Since these are the highest LVD tasks in the system, this means at that we will accrue a minimum of $\frac{k}{n}$ of the total possible utility.

The most important element of this bound is that it is not based on the utilization of the taskset or of any specific task within it. Therefore, this bound remains valid in overloaded systems, which are the prime candidate for a utility accrual algorithm.

6.8 gMUA

Global multiprocessor utility accrual scheduling algorithm (gMUA) was the first multiprocessor utility accrual algorithm designed [36]. While G-EDF is able to meet all deadlines, it creates the same schedule as G-EDF. After this point, it provides heuristic utility accrual behavior. Since the algorithm is quite complex and has been well documented by its authors, we provide only a general description of it here. Our implementation provides $O(1)$ insertion of new tasks, $O(mn^2)$ scheduling, and $O(m^2)$ mapping.

In gMUA, each task first has its LVD computed, and is placed in a deadline-ordered list. Each task in this list is then considered, and assigned to the core with the lowest total utilization. Each core's list is deadline ordered. Each core's list is then checked for feasibility by computing the slack time for each task with respect to the tasks before it. If any task has a slack time less than 0, the list is declared infeasible and the lowest LVD task in that core's list is removed. This process is repeat until the list is feasible, and is done for each core. Each core then executes the task at the head of its list. It should be noted that while the algorithm assigns tasks to cores, all cores are considered identical, and therefore the mapping algorithm described in Chapter 5 is still used. Pseudocode for gMUA is shown in Algorithm 2.

In underload, all of the schedulability tests previously mentioned for G-EDF may be applied to gMUA. In overload, gMUA is capable of providing statistical assurances for timeliness behavior and utility accrual. However, since its bounds are based on probabilistic execution times, and therefore are not not valid under worst-case execution times, we omit the bound here.

6.9 NG-GUA

Non-greedy global utility accrual is a algorithm which expands gMUA to handler inter-task dependencies [52]. When dependencies are not present, it creates schedules identical to gMUA and has the same asymptotic costs. However, since it allows resources, it incurs additional overhead, and is therefore included here as a comparison. Our implementation is derived directly from Garyali's implementation presented in [52]. We have made several slight improvements which have improved the implementations speed by around 5

6.10 G-GUA

Greedy global utility accrual is a variant of NG-GUA that trades G-EDF performance in underloads for a higher average case utility accrual performance in overloads [52]. As with gMUA, G-GUA is a complex algorithm that has been described elsewhere, and so only a short description of its functionality without dependencies is provided here. Like NG-GUA and gMUA, insertions are $O(1)$, scheduling is $O(mn^2)$ scheduling, and mapping is $O(m^2)$.

GGUA functions as follows; we first compute the LVD of each task and abort any tasks which need aborted. Each task is also placed in a list, which is then sorted by LVD. The scheduler then loops over every task in this list. For each task, all cores are considered, in order from the least to the highest total utilization. The task is inserted at its deadline position on

```
Input: Runqueue;                                   //FIFO-ordered runqueue
Data: $DList_1 ... DList_m$;          //Empty deadline-ordered per-core lists
Data: LVDList;                                    //Empty LVD-ordered list
foreach Task i in Runqueue do
    if $deadline(i) < now$ then
        AbortTask($i$);
    end
    CalculateLVD($i$);
    InsertOnList($i$,$LVDList$);
end
SortByLVD($LVDList$);
foreach Task i in LVDList do
    while $NotFeasible = true$ do
        P = FindProcessor(CpuMask);
        InsertAtDeadline($i$,$DList_P$);
    end
    foreach Core P do
        while ScheduleInfeasible($DList_P$) do
            RemoveLeastLVDTask($DList_P$);
        end
    end
end
return HeadOf($DList_1 ... DList_M$);
```

Algorithm 2: Global multiprocessor utility accrual scheduling algorithm

the core's schedule, and the schedule is then checked for feasibility using the same method described for gMUA. If the schedule is infeasible, the task is removed, and the scheduler tries it on the next core. Therefore, for a task to be omitted from the final schedule, it must be found infeasible on every core. Pseudocode for a simplified version of G-GUA without dependencies is shown in Algorithm 3. The `FindProcessor(CpuMask)` function returns the lowest total utilization core not in `CpuMask`. If it returns `NULL`, there are no more cores. Our implementation is derived directly from Garyali's implementation presented in [52]. We have made several slight improvements which have improved the implementations speed by around 5

6.11 Partitioned RMS

Partitioned rate monotonic scheduling (P-RMS) schedules tasks based on a per-core basis. Tasks are assigned to cores offline. On each core, tasks are scheduled using rate monotonic scheduling, which assigns increasing fixed priorities to tasks in order of decreasing period.

```
Input: Runqueue;                                    //FIFO-ordered runqueue
Data: $DList_1...DList_m$;             //Empty deadline-ordered per-core lists
Data: LVDList;                                      //Empty LVD-ordered list
Data: CpuMask;        //Mask of all CPUs on which a given task infeasible
Data: NotFeasible;             //Flag to check if feasibility check failed
foreach Task i in Runqueue do
    if deadline(i) < now then
        AbortTask(i);
    end
    CalculateLVD(i);
    InsertOnList(i,LVDList);
end
SortByLVD(LVDList);
foreach Task i in LVDList do
    CpuMask = 0;
    NotFeasible = true;
    while NotFeasible = true do
        P = FindProcessor(CpuMask);
        if P = NULL then
            break;
        end
        InsertAtDeadline(i,$DList_P$);
        IncreaseUtilization(i,P);
        if ScheduleInfeasible($DList_P$) then
            RemoveTask(i, $DList_P$);
            DecreaseUtilization(i,P);
            NotFeasible = true;
            AddCpuToMask(P,CpuMask);
        end
        else
            NotFeasible = false;
        end
    end
end
return HeadOf($DList_1...DList_M$);
```

Algorithm 3: A simplified version of G-GUA

Scheduling is preemptive, so the arrival of a task with a shorter period will preempt an already executing task with a longer period. In our implementation, tasks are inserted at their period position in the runqueue. This makes insertion $O(n)$ and scheduling $O(1)$.

RMS itself has been thoroughly studied for over 30 years, and both exact and sufficient schedulability tests have been determined. Only the sufficient tests are shown here. Liu and Layland proved that n periodic tasks are schedulable under RMS if $U \leq n(2^{1/n} - 1)$. This results in a lower utilization bound of 69.3% [80]. It has also been shown that in underload, RMS is an optimal fixed-priority scheduler [72]. Since the set of tasks assigned to each core under P-RMS may be considered a separate taskset, these tests extend to P-RMS. However, since it has been proved that partitioning a taskset is analogous to the bin-packing problem and is therefore NP complete, we also require tests to determine whether a taskset can be partitioned into m individually schedulable tasksets [70]. Oh and Baker have shown that Liu and Layland's utilization bound can be extended to show that any independent periodic taskset with a utilization less than $m(2^{1/2} - 1)$ can be scheduled with P-RMS on a m core system [93]. This bound was extended by Lopez et. al. to

$$U < (m\beta_{LLB} + 1)(2^{1/(\beta_{LLB}+1)} - 1) \tag{6.11}$$

where $\beta_{LLB} = (1/log_2(U_{max} + 1)$. The generic form of this for all utilization becomes $U \leq (m + 1)(2^{1/2} - 1)$ [83].

6.12 Partitioned EDF

Under partitioned earliest deadline first (P-EDF) scheduling, tasks are first assigned to cores offline and then scheduled dynamically. At every scheduling event, the earliest deadline task available on that core is executed. In our implementation, tasks are inserted at their deadline position in the runqueue. This makes insertion $O(n)$ and scheduling $O(1)$.

Like RMS, EDF has been extensively studied. Under EDF, a taskset having a total utilization of less than 1 is both a sufficient and exact tests for schedulability, and is therefore an optimal scheduling algorithm on uniprocessors [35]. EDF is also known to bound worst-case deadline tardiness [101]

While it is therefore implicit that if a taskset is schedulable if it can be partitioned onto m cores in such a way that no core's total utilization is greater than 1, as with RMS the partitioning itself is NP complete. Lopez et. al. have proven a utilization lower utilization bound of

$$U < \frac{m\beta_{EDF} + 1}{\beta_{EDF} + 1} \tag{6.12}$$

where $\beta_{EDF} = 1/U_{max}$. The generalized form of this bound is $U \leq (m+1)/2$ for P-EDF [82].

6.13 Partitioned HVDF

Partitioned highest value density first (P-HVDF) is a relatively simple heuristic utility accrual algorithm based on the LVD concept. On each core, tasks are inserted into a deadline-ordered runqueue on arrival. At each scheduling event, the scheduler checks if it should abort the first task, and if so, performs the abort and executes the newly aborted task. If the lowest deadline task has not passed its deadline, no other task will have, since the list is deadline ordered. The scheduler then computes the LVD of each task in its runqueue and executes the task with the highest LVD. Both insertion and scheduling are therefore $O(n)$.

No schedulability tests or utility accrual bounds are known for HVDF. It is provided here as a comparison to more advanced uniprocessor utility accrual algorithms, such as DASA-ND and LBESA.

6.14 Partitioned LBESA

Locke's best effort scheduling algorithm (LBESA) was designed by Doug Locke and first implemented in the Alpha kernel [91]. It provides optimal behavior in underload by defaulting to EDF, and provides best-effort performance in overload [81]. LBESA utilizes TUFs to provides this functionality. While the original algorithm allowed for TUFs of various forms, we allow only downward step TUFs in our implementation.

The algorithm is designed as follows; all tasks first have their LVD computed. They are then placed in deadline-ordered schedule. This schedule is checked for feasibility by computing the slack time for each task based on the execution times of the tasks before it. If slack time for all tasks is negative, then the system is in underload, and the task at the head of the schedule is selected for execution. If any task has a negative slack time, it may be assumed that the system is in overload. The amount of overload is then compared to a user-defined value representing the maximum allowable overload for the system. If this comparison determines that an undesirable overload has occurred, then the task with the lowest LVD is removed from the schedule. This process is repeated until an overload-free schedule is found.

Our implementation is as follows. Each task in the local runqueue has its LVD calculated. It is then placed into two ordered lists — one deadline ordered and the other LVD ordered. The deadline ordered list is then checked for feasibility. If the list is found to be infeasible, the task at the tail of the LVD-ordered list is then removed from both lists. The actual runqueue is deadline ordered. Insertion is therefore $O(n)$, while scheduling is $O(n^2)$. Additionally, we abort tasks which have exceeded their deadline. This was not described by Locke, but we believe it is a reasonable addition. Pseudocode is shown in Algorithm 4.

```
Input: Runqueue;                          /*Deadline-ordered runqueue*/
Data: DList;                              /*Empty deadline-ordered list*/
Data: LVDList;                            /*Empty LVD-ordered list*/
foreach Task i in Runqueue do
    if deadline(i) < now then
        AbortTask(i);
        return i;
    end
    CalculateLVD(i);
    InsertOnList(i,LVDList);
    InsertAtTail(i, DList);
end
SortByLVD(LVDList);
while ScheduleInfeasible(DList) do
    RemoveLeastLVD(LVDList, DList);
end
return HeadOf(DList);
```

Algorithm 4: Locke's Best Effort Scheduling Algorithm

Because it defaults to EDF, the previously mentioned utilization bound of $U \leq (m+1)/2$ for P-EDF also applies to P-LBESA.

6.15 Partitioned DASA-ND

Dependent activity scheduling algorithm (DASA) was developed by Ray Clark as the scheduling algorithm for the Alpha kernel [39]. It is the successor to LBESA, and like it, exhibits EDF performance in underloads while relying on a heuristic utility accrual approach in overloads. Clark demonstrated that DASA will often create slightly better schedules than LBESA. The largest difference between the two algorithms is that DASA allows inter-task dependencies. Unlike LBESA, DASA only allows downward-step TUFs. Also, DASA allows tasks to be aborted, but Clark does not specify a fixed set of conditions under which a task should be aborted, except for two tasks deadlocking over a resource. He further assumes that abort handlers will provide timing constraints, which we forbid in this thesis.

Since we only study independent tasks in this thesis, we present DASA-ND, a variant of DASA that does not allow dependencies. It is therefore a direct competitor to LBESA. The algorithm functions as follows. Tasks first have their LVD computed. They are then placed in a list, and that list is sorted by LVD, with the highest LVD tasks at the front. Tasks are then considered in LVD order. Each task is placed in a deadline ordered schedule. This schedule is checked for feasibility in the same manner as LBESA. If the schedule is not feasible, the task is removed, and the next task is considered, until all tasks have been considered. The resulting schedule is therefore feasible.

Our implementation adds one optional function. As with LBESA, when a task exceeds it deadline, it is aborted and immediately executed to remove it from the system. First, all tasks in the local runqueue have their LVD calculated and are checked to see if they should abort. If a task is found that must be aborted, it is executed. All tasks are then placed in a list, which is sorted by LVD. Starting at the head of this list, tasks are removed and inserted into a deadline-ordered list. This list is checked for feasibility in the same way as LBESA. Once all tasks have been considered, the first task in the deadline ordered list is passed to the scheduler for execution. The runqueue is deadline ordered, so insertion is $O(n)$, while scheduling is $O(n^2)$. Pseudocode is shown in Algorithm 5.

```
Input: Runqueue;                         /*Deadline-ordered runqueue*/
Data: DList;                             /*Empty deadline-ordered list*/
Data: LVDList;                           /*Empty LVD-ordered list*/
foreach Task i in Runqueue do
    if deadline(i) < now then
        AbortTask(i);
        return i;
    end
    CalculateLVD(i);
    InsertOnList(i,LVDList);
end
SortByLVD(LVDList);
foreach Task i in LVDList do
    InsertAtDeadline(i,DList);
    if ScheduleInfeasible(DList) then
        RemoveFromList(i,DList);
    end
end
return HeadOf(DList);
```

Algorithm 5: Dependent activity scheduling algorithm, no dependencies

As with LBESA, the previously mentioned utilization bound of $U \leq (m+1)/2$ for P-EDF also applies to P-DASA-ND.

6.16 Clustered EDF

Clustered earliest deadline first (C-EDF) or hybrid EDF (H-EDF) is a restricted migration variant of G-EDF [32]. In clustered EDF, tasks are statically partitioned offline onto clusters of cores. Within each cluster, G-EDF scheduling is performed, and tasks are allowed to migrate freely, however no task may migrate out of its cluster. C-EDF was designed as a compromise between G-EDF and P-EDF that could take advantage of the shared caches on modern multicore processors.

In ChronOS, C-EDF is implemented by defining separate G-EDF domains for each cluster, and then setting the CPU affinity of each task to the desired cluster. The method used to determine cluster size and create clustered tasksets is discussed in Chapter 8.

Since C-EDF functions like G-EDF on a sub-taskset, the same schedulability tests mentioned for G-EDF may be applied to each cluster to determine schedulability. C-EDF is known to bound deadline tardiness as long as the total taskset utilization does not exceed m [20].

6.17 Summary

The execution cost, queue insertion cost, mapping cost, architecture, and categorization of each algorithm discussed is summarized in the following table. Note that for mapping cost, $O(m^2)*$ implies that the mapping cost, while still $O(m^2)$, is the lower worst-case cost described in Chapter 5 for fixed and task-dynamic priority algorithms. Also, since `SCHED_FIFO` cannot be analyzed in the same way as the other algorithms, it is omitted.

Table 6.3: Summary of algorithm properties

Algorithm	Grouping	Architecture	Scheduling	Insertion	Mapping
G-FIFO	Traditional	Concurrent	$O(1)$	$O(1)$	$O(1)$
G-RMS	Traditional	STW	$O(m)$	$O(n)$	$O(m^2)*$
G-EDF	Traditional	STW	$O(m)$	$O(n)$	$O(m^2)*$
G-NP-EDF	Traditional	Concurrent	$O(1)$	$O(n)$	$O(1)$
G-NP-HVDF	GUA	Concurrent	$O(1)$	$O(n)$	$O(1)$
G-HVDF	GUA	STW	$O(n)$	$O(1)$	$O(m^2)$
gMUA	GUA	STW	$O(mn^2)$	$O(1)$	$O(m^2)$
NG-GUA	GUA	STW	$O(mn^2)$	$O(1)$	$O(m^2)$
G-GUA	GUA	STW	$O(mn^2)$	$O(1)$	$O(m^2)$
P-RMS	Partitioned	Uniprocessor	$O(1)$	$O(n)$	N/A
P-EDF	Partitioned	Uniprocessor	$O(1)$	$O(n)$	N/A
P-HVDF	Partitioned	Uniprocessor	$O(n)$	$O(1)$	N/A
P-LBESA	Partitioned	Uniprocessor	$O(n^2)$	$O(n)$	N/A
P-DASA-ND	Partitioned	Uniprocessor	$O(n^2)$	$O(n)$	N/A

Chapter 7

Scalability

In an ideal world, if a program can be parallelized into m threads, we should see an m time speedup on an m core machine. However, this is sadly almost never possible in the real world due to the cost of inter-processor communication and synchronization.

The governing law in multiprocessor programming is Amdahl's law, which states that the maximum speedup a program may achieve on an n core system is

$$S = \frac{1}{1 - p + \frac{p}{n}} \tag{7.1}$$

where p is the parallel part of the program [57]. This has an important implication; for a given application with a given amount of parallelism, there is a limit to the speedup that can ever be achieved. In other words, as the number of cores increases, the marginal speedup decreases and the total speedup approaches some finite limit.

The most fundamental issue encountered in multiprocessor programming is the use of shared data objects, and the data protection that must be used with them [56]. At its core, this presents the mutual exclusion problem [102]. The question of scalable global scheduling in ChronOS can be viewed as two distinct instances of this problem. First, we have a global queue which contains all of the tasks in the system. To guarantee that the queue is in a correct state when used, we must provide mutual exclusion for it. Second, we have a schedule, which must be changed at scheduling events. Since this schedule is generated from the global queue, we must be sure that the state of the queue does not change during the generation of the schedule.

The most common solution to mutual exclusion is the use of locks, and we follow this approach in ChronOS. Both the global queue and the global schedule are protected with a single lock on each domain. Any core wishing to add or remove a task from the global queue must acquire its lock, and any core wishing to create a schedule must acquire the schedule's lock. Additionally, since the global schedule is created from the global queue, we must acquire the global queue's lock when trying to schedule. The only other locks accessed by the scheduler are the locks for the per-core `runqueues`. Since these are per-core, each task must only lock its own lock to schedule, and will only acquire another `runqueue's` lock if it needs to migrate a task from it. Therefore, while they are still locks and will effect scalability, no lock should ever need to be acquired by a significant portion of the system's cores at once, except in extremely rare circumstances.

There are two well-known problems that arise when using locks to protect data structures on high core-count systems. The first is that the lock may be highly contended, meaning that a large number of threads may be waiting for access. This means that because the threads are waiting for the lock, only a small amount of progress is being made in the system. This can happen even if the threads are attempting to perform operations that would not collide with each other, since the lock protects the whole data structure. The second problem is that in a modern multiprocessor system, not all memory is necessarily local. Furthermore, data is often stored in various caches throughout the system. This means that if all the cores are accessing the same piece of data, each of them needs a fresh copy any time one of them modifies it. This process generates a significant amount of bus traffic, which potentially delays the next core in line for the lock from acquiring it [56].

To understand analyze whether scalability is an issue in ChronOS, we devise a set of tests to run. G-EDF was selected as the scheduler to be used, because it is the most throughly studied global scheduler presented in this thesis. Our test consists of 10 tasksets at every integer load point from 1 to 48, resulting in a total of 480 tasksets. These tasksets are periodic with implicit deadlines. Periods are uniformly distributed over [10ms, 100ms] and have per-task utilization evenly distributed over [0.1, 0.4]. These values were chosen because they fit nicely into the previously mentioned schedulability tests for G-EDF. Based on the Srinivasan and Baruah's test, we can guarantee that, in theory, G-EDF should be able to schedule all tasksets up to a load of 24, while the GFB test allows us to generate a bound of 29. The results of executing these tasksets on ChronOS is shown in Figure 7.3.

Obviously, ChronOS does not perform as intended. To discover the source of this poor performance, we record the average number of waiters at each of the two previously mentioned locks. Figure 7.1 shows the results as a histogram. The source of the problem is the high contention for the scheduling lock. This is an important results, since the work of Brandenburg et. al. identified contention on the global queue as the largest detriment to scalability [31].

This problem can be resolved two ways; either the locking mechanism can be improved to decrease the contention, or the number of core contending for the lock can be reduced. We take both approaches. Raw spinlocks in Linux are implemented as ticket locks. This means that they offer similar performance to a raw compare-and-swap lock for low contention, but also provide starvation freedom and FIFO ordering. However, they are well known to scale poorly. To resolve this, we replaced the ticket lock being used to guard the schedule with an MCS lock. MCS is a queue-based lock in which each lock requester spins on only its

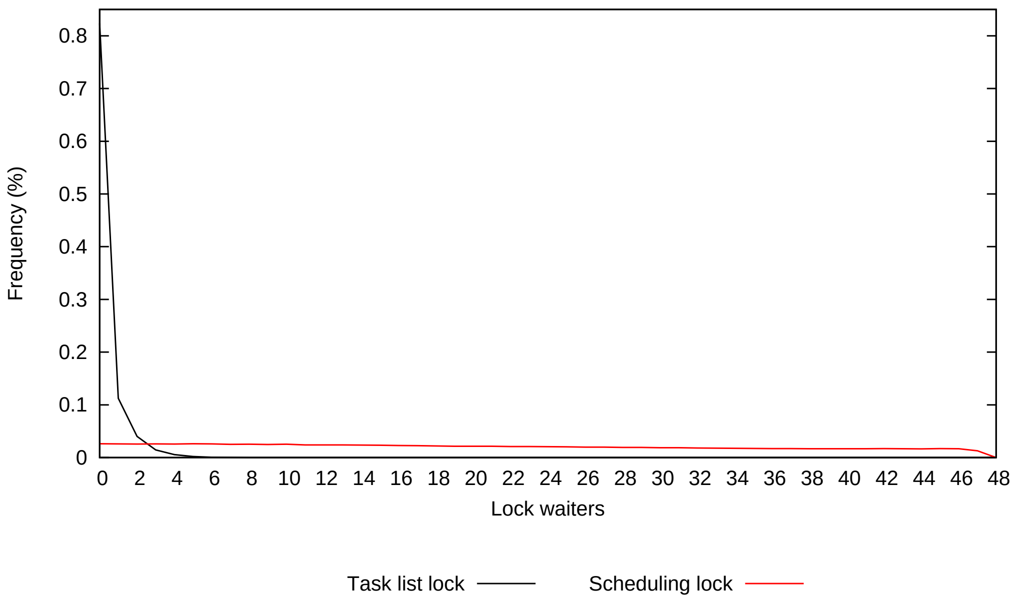


Figure 7.1: Number of waiters for the scheduling and task list locks

local memory [86]. The performance difference between these locks on our platform is shown in Figure 7.2. For this experiment, a variable number of threads were created which each requested and released the lock 100,000 times. The plotted values are the total time required for all threads to acquire and release the lock this many times.

When 48 threads are all contending for the lock, the resulting time for the ticket lock is 13 times higher than for the MCS lock. However, the ticket lock is faster when the contention is low. Based on this, we replaced the scheduling lock with an MCS lock. Figure 7.3 shows the performance relative to the previous results. This change increases the load at which we achieve full schedulability by 200%. However, we are still not performing in keeping with the theoretical performance.

To improve the scalability further, we leverage our knowledge of task mapping in the stop-the-world model to avoid sending IPIs to ever core at every event. First, we know that cores which are currently executing a task at a higher priority than the global scheduling priority will reschedule their current task when they receive an IPI. Therefore, we check the current priority of each core and only send an IPI if the core's priority is less than or equal to the global scheduling priority. Second, if there are more tasks in the system than cores, we know that the task mapping algorithm will not make any unnecessary migrations. Because of this, we know that any core with only one task will be mapped that task. This implies that tasks will only be moved from cores with more than one task, and will only be to cores with no tasks. In other words, if there are less tasks than cores, any core currently executing a globally scheduled task does not need to receive an IPI. Finally, we know that if there are n tasks in the system, we know that we will need to make at most $n-1$ migrations, which

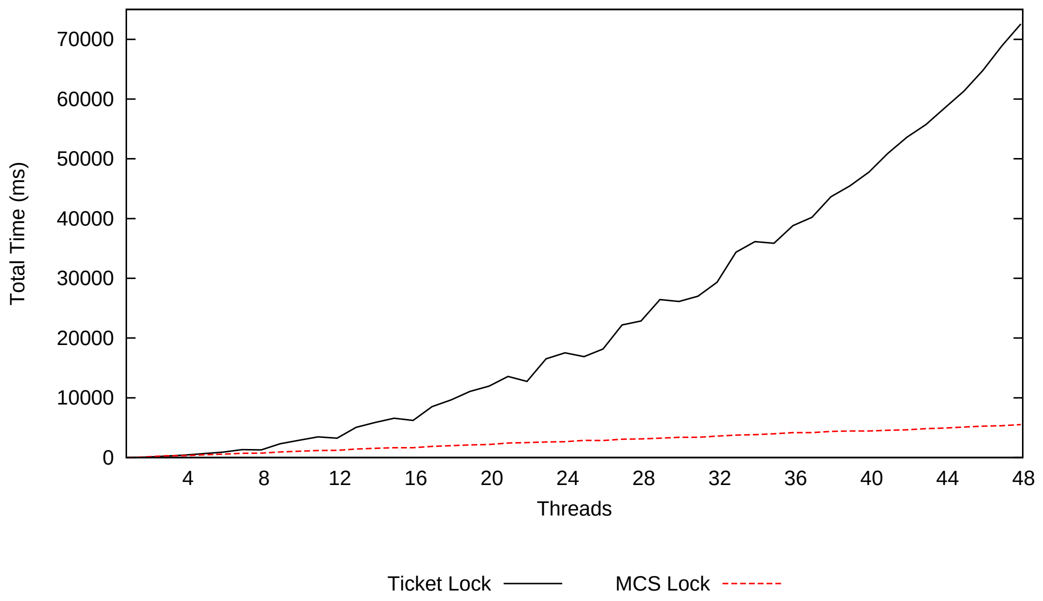


Figure 7.2: Locking times for ticket and MCS locks on a 48-core system

means we will need to send IPIs to at most $n - 1$ cores. Since we know that tasks which must be migrated are assigned to the first available core, we know that we need to signal the first $n - 1$ cores in the system. Combining this with the previous rule, we know if there are less tasks than cores we need to signal only the cores in the first $n - 1$ which are not already executing a globally schedule task.

The results of applying this logic to our system is shown in Figure 7.3. This increases the load at which we are able to schedule all tasks by 12.5% beyond the improvements of the MCS lock. While this is close to the theoretical lower bound, it is still not quite sufficient to meet it. Unfortunately, we cannot decrease the number of IPIs sent further in the stop-the-world architecture, because job-dynamic priority algorithms such as G-GUA may change any number of tasks in the schedule at every scheduling event. However, we can leverage knowledge specific to fixed and dynamic priority algorithms. In these algorithms, each scheduling event can enact only one change on the state of the global queue: a task must be either added or removed. Furthermore, when a task is removed, the scheduling event occurs on the core which removed the task, and because our mapping algorithm will not move running tasks, we can guarantee that the task assignment for this core is the only change that will be made in the system's schedule. Based on this, when a core finishes a task, we can schedule without sending any IPIs. This change is shown in Figure 7.3 as "Reduced IPIs 2". This final change boosts our performance above the utilization bound.

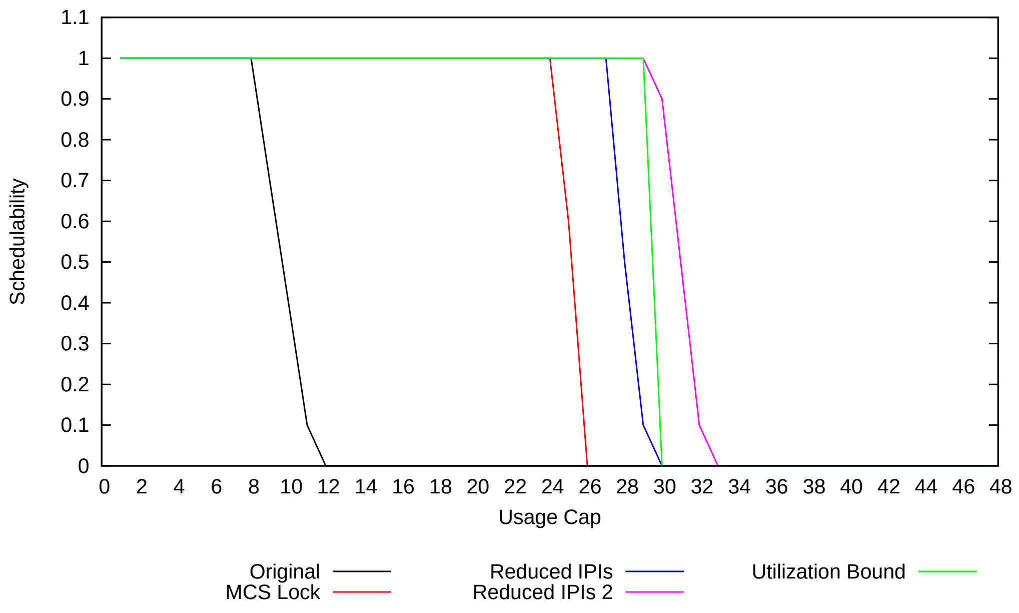


Figure 7.3: ChronOS scheduling performance with various improvements

Chapter 8

Experimental Scheduling Results

8.1 Hardware Platform

We conducted experiments on three different hardware platforms. The detailed specifications of each machine are shown in Table 8.1. The first machine has a pair of quad-core Intel Xeon E5520 processors. These processors support hyper-threading, but this feature has been disabled due to the non-determinism it introduces. The second has a pair of 8-core AMD Opteron 6128 chips. The third is based on four 12-core AMD Opteron 6164 HE chips. Between these three platforms, we have a wide range of core counts and clock speeds, allowing us to extensively test the scalability of our kernel and schedulers.

Table 8.1: Specifications of hardware platforms

Specification	8-Core	16-Core	48-Core
Processor Model	Intel Xeon E5520	AMD Opteron 6128	AMD Opteron 6164 HE
Processor Count	2	2	4
Core Count	4	8	12
Clock Speed	2267 MHz	2000 MHz	1700 MHz
L2 Cache	4x 256 KB	8x 512 KB	12x 512 KB
L3 Cache	8 MB	2x 6 MB	2x 6 MB
Bus Speed	2933 MHz	3200 MHz	3200 MHz
Memory	8 GB	8 GB	16 GB

8.2 Software Platform

All machines were running Ubuntu 10.04 Server 64-bit with a ChronOS kernel based on the 2.6.31.14 kernel and the 2.6.31.12-rt21 `PREEMPT_RT` patch, which we rebased to the 2.6.31.14 kernel. All user space code was compiled with GCC 4.4.3 and QT4 4.6.2.

8.3 Methodology

All of the algorithms described except the forms of RMS do not 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 are either difficult or impossible when aperiodic tasks are allowed into the system. All deadlines are made equal to periods for the same set of 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, utility, and processor affinity for each task. Our application uses the "thread-per-task" model, which creates a single OS thread for each task, to which this task is permanently affixed. 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 begins a real-time segment, burns the processor for its WCET, and then ends its segment and sleeps until the beginning of the next job. A simplified task loop is shown in Figure 8.1. All tasks are run at a priority of 40, so that they are under priority of the interrupts. This is not strictly necessary, since our application's `wait_for_next_period()` method uses the nanosleep() function, which relies on the per-cpu timer interrupt. Since this interrupt has been left in the dedicated interrupt context in `PREEMPT_RT`, it would be possible for us to run our tasks over the priority of the interrupts. However, doing so assumes that no other interrupts will be needed, which is an assumption that we cannot confidently make about real-world real-time workloads. Moving the test application above priority 50 results in an increase in performance for more complex schedulers like G-GUA, since they no longer have to regenerate a schedule when preempted by an interrupt. For the simpler algorithms, such as G-FIFO, the performance difference is negligible. This approach is also used to make our study more comparable with the two previous studies ([31], [32]) which used a vanilla Linux kernel and were therefore subject to interference from interrupts.

The processor is burned by incrementing a counter a precomputed number of times. The number of times this counter must be incremented to burn $1\mu s$ of time is called the system slope. There are two ways in which we can burn the processor: with or without checking for

```
int max_runs = ceil(runtime/period);
int runs = 0;

while(runs <= max_runs) {
    runs++;
    begin_rt_seg(prio, period, deadline, wcet, utility)
    burn_cpu_abort(wcet);
    end_rt_seg(prio + 1);
    wait_for_next_period();
}
```

Figure 8.1: Simplified test application task loop

aborts. If we check for aborts, we will return from the burning function within a specifiable amount of time after the abort is sent. As is shown in Figure 8.2, checking for aborts does incur a slight amount of overhead. Over long burns, this is generally 3-4%. All tasks in our test application check for aborts, to provide a consistent workload for our schedulers. The slopes on the machines used have been adjusted to account for this overhead, but this does slightly bias our tests against non-aborting algorithms such as G-EDF which do not need this functionality.

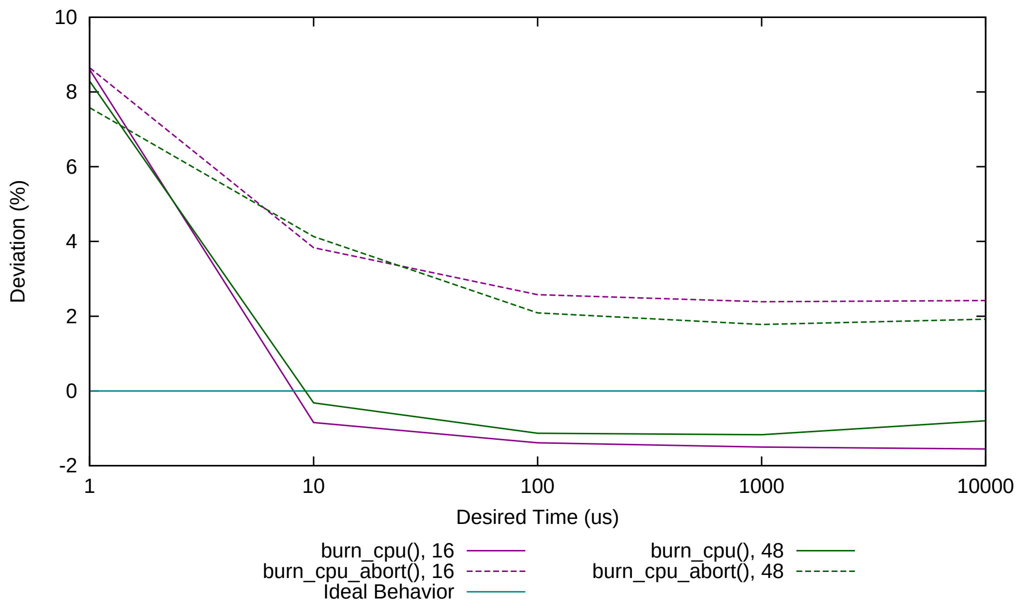


Figure 8.2: Slope accuracy

8.4 Baker Tasksets

Rather than use a single type of taskset to evaluate the algorithms, we use the a variation of the method described by Baker to create a to create a set of tasksets which evaluate the scheduler's performance under a variety of circumstances [9]. 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 by Brandenburg et. al., we use three different weightings of two statistical distributions — a uniform and a bimodal distribution [31]. This gives us a total of six taskset distributions. As was done by Brandenburg et. al., we distribute all periods uniformly between $10ms$ and $100ms$, and set deadlines equal to periods. 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. The average number of tasks in a taskset by utilization and distribution is shown in Figure 8.3.

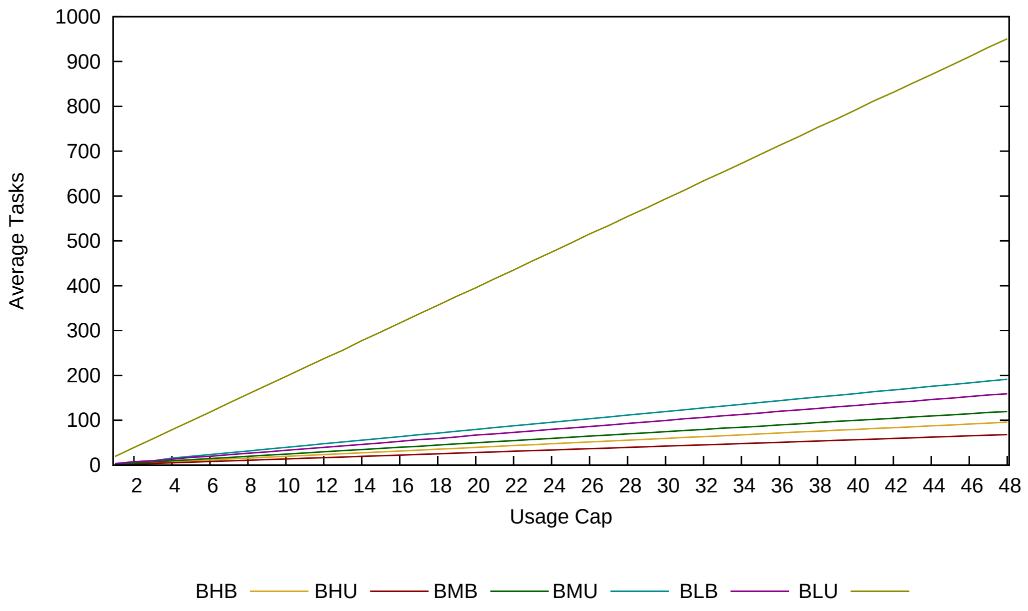


Figure 8.3: Average tasks in a taskset for various per-task weight distributions

For each algorithm, a total of 1,031,707,071 jobs of 38,779,473 tasks were scheduled on the 48-core platform. 118,670,394 jobs of 4,460,485 tasks were scheduled on the 16-core platform, and 47,781,229 jobs of 1,172,208 tasks were scheduled on the 8-core machine. This is by no means the largest sample size ever used to analyze real-time schedulers; Brandenburg et. al. have published two papers using 5.5 and 8.5 million tasksets [30], [31]. However, both of these works rely on using a small number of tasksets to characterize the behavior of a system, and then perform offline schedulability tests on the full set of tasksets. 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.

Utilities were assigned to tasks by taking the modulo of a randomly generated number and the period of the taskset. This results is a slight value-density bias toward lower utilization tasks, and no value-density bias with respect to periods. This leads to correlation values of -0.43 (BLU), -0.54 (BMU), -0.32 (BHU), -0.23 (BLB), -0.33 (BMB), and -0.33 (BHB) between task utilization and value density for the different distributions. Utilities were generated in this manner to bias non-utility accrual schedulers such as variants of RMS toward higher utility accrual numbers and thereby present competition to the utility accrual schedulers. The performance of utility accrual schedulers under various utility distributions has already been researched by Garyali [52].

8.5 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 [15], and a simple least-utilization algorithm. The first-fit algorithm was used for P-EDF, P-LBESA, and P-DASA-ND, all of which follow EDF ordering in underloads. 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 core with a utilization low enough that the sum of the utilizations of all tasks previously assigned to the core and the current task is less than 1. This can be represented as follows. On an m core machine, let $T(\pi_k)$ denote the tasks already assigned to a core k, where $k \leq m$. We will assign T_i to π_k if and only if

$$\left(1 - \sum_{T_j \in T(\pi_k)} U_j\right) \geq U_i \tag{8.1}$$

We make two changes to the original algorithm. First, some tasksets which are feasible under algorithms like PFair and LLREF cannot be partitioned. 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. Additionally, since we test in overloads (which are by definition infeasible) and need some kind of partitioning to provide valid results for best-effort algorithms like DASA and LBESA, we need a solution to partitioning infeasible tasksets. In both such a case, we employ a best-effort partitioning approach. If a task does not fit on any core, the task is assigned to the core with the lowest total sum utilization.

Second, Baruah and Fisher assume that the algorithm is applied to the tasks in non-decreasing deadline order. However, when combined with the first change and applied to our tasksets, this order sometimes produces uneven distributions of tasks or results in cores with utilizations of significantly higher than 1. 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 core 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 the remaining two algorithms — P-RMS and P-HVDF. 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 always selected to correspond to physical processors, and therefore our 8, 16, and 48 core systems use clusters of 4, 8, and 12 cores respectively. 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.

8.6 Performance Measurements

We record four measurements when experimenting with scheduling algorithms — Deadline Satisfaction Ratio (DSR), Accrued Utility Ratio (AUR), Schedulability, and Mean Maximum Tardiness (MMT).

The DSR for a given taskset i at a utilization of U is

$$DSR_U^i = \frac{Deadlines\ met}{Total\ deadlines} \tag{8.2}$$

And therefore, the average DSR for k tasksets with a utilization of U is

$$DSR_U = \frac{\sum_{i=1}^{k} DSR_U^i}{k} \tag{8.3}$$

Similarly, the AUR for a given taskset i at a utilization of U is

$$AUR_U^i = \frac{Utility\ accrued}{Total\ possibleutility} \tag{8.4}$$

And the average AUR for k tasksets with a utilization of U is

$$AUR_U = \frac{\sum_{i=1}^{k} AUR_U^i}{k} \tag{8.5}$$

The schedulability for a given taskset i at a utilization of U is

$$Schedulability_U^i = \begin{cases} 1 & \text{for } DSR_U^i = 1 \\ 0 & \text{for } DSR_U^i < 1 \end{cases} \tag{8.6}$$

Therefore, the schedulability for a given utilization range U with k tasksets is calculated as follows:

$$Schedulability_U = \frac{\sum_{i=1}^{k} Schedulability_U^i}{k} \tag{8.7}$$

Finally, the maximum tardiness is calculated by measuring the worst tardiness θ_U^i experienced by any task in a taskset i at a utilization U. The mean maximum tardiness for a given utilization range U with k tasksets is therefore

$$MMT_U = \frac{\sum_{i=1}^{k} \theta_U^i}{k} \tag{8.8}$$

DSR measures the percentage of the scheduled jobs that successfully met their deadlines. Similarly, AUR measures the amount of utility accrued by these jobs with respect to the total possible utility. These two metrics have been sufficient in the past [52], however, as our results will show, they are no longer sufficient to understand the behavior of algorithms. To complement them, we add schedulability, which represents the percentage of the time an algorithm is able to meet all of a taskset's deadlines, and mean maximum tardiness, which measures the worst tardiness experienced by any task in the taskset. These two metrics help us understand the behavior of algorithms in specific contexts; schedulability is helpful in the context of hard real-time system, where the main constraint is often that all deadlines must be met. In soft real-time system, a commonly used constraint is that tardiness must bounded, and therefore measuring tardiness is a reasonable addition.

8.7 Schedulability Results

The schedulability results of our tests are shown in Appendix A, due to their length. For schedulability results, we mostly ignore the results of G-NP-HVDF, G-HVDF, P-HVDF, and G-GUA, since they make no attempt to meet all deadlines.

When analyzing scalability, we especially focus on the BHU, BMU, and BLU tasksets. This is done for two reasons. First, these are the easiest to analyze under the previously discussed schedulability tests for G-RMS, P-RMS, G-EDF, and P-EDF. Second, these represent the most extreme cases, with BHU being the most difficult from a bin-packing perspective and BMU and BLU having the largest number of tasks.

Tables 8.2 to 8.4 show the computed theoretical schedulability bounds for each of these three algorithms for each of the three distributions on each platform. These are only the results of Anderson and Baruah's schedulability tests for G-RMS, and for Baruah's test and the GFB test for G-EDF. Where two bounds give different results, we take the higher bound. The BHU bounds for G-EDF are significantly lower than their actual expected performance,

Table 8.2: Schedulable utilization bounds for G-RMS, P-RMS, G-EDF, and P-EDF on the 8-core platform

Algorithm	BHU	BMU	BLU
G-RMS	N/A	N/A	2.78
P-RMS	3.81	4.44	5.17
G-EDF	1.70	5.20	7.30
P-EDF	4.68	6.00	7.36

Table 8.3: Schedulable utilization bounds for G-RMS, P-RMS, G-EDF, and P-EDF on the 16-core platform

Algorithm	BHU	BMU	BLU
G-RMS	N/A	N/A	5.45
P-RMS	7.23	8.63	10.26
G-EDF	2.50	10.00	14.50
P-EDF	8.89	11.71	14.64

Table 8.4: Schedulable utilization bounds for G-RMS, P-RMS, G-EDF, and P-EDF on the 48-core platform

Algorithm	BHU	BMU	BLU
G-RMS	N/A	N/A	16.11
P-RMS	20.90	25.39	30.60
G-EDF	5.70	29.20	43.30
P-EDF	25.74	34.57	43.73

since both of the tests used are known to be sub-optimal for heavy tasksets [9]. However, this is not of critical importance for us, since scalability issues are more likely to arise during the lighter tasksets.

8.7.1 8-Core Schedulability Results

On the 8-core platform, there is little of interest to observe. Most of the algorithms behave in accordance with their theoretical performance. G-EDF, G-RMS, P-EDF, and P-RMS all meet the bounds listed above. G-EDF, gMUA, and NG-GUA all demonstrate nearly identical performance, which is expected since they all should theoretically produce the same schedules when the system is underloaded. The lone exception is the BLU distribution, where gMUA and NG-GUA begin loosing deadlines after a load of 6, compared to 7 for G-EDF. C-EDF slightly outperforms G-EDF in most cases; this performance differences ranges from 66% more tasksets scheduled at a load of 7 in the BMB case to no difference at all for the BLU case. G-EDF, C-EDF, P-EDF, G-RMS, P-RMS, P-LBESA, and P-DASA-ND never fail to schedule a taskset below a load of 4. One interesting aspect of G-EDF is that it for the BLB case, rather than the sharp drop we see in the BLU and BMU cases, performance gradually degrades. This is due to the fact that the BLB case is the most likely to contain tasksets that demonstrate the Dhall effect. G-GUA, G-HVDF, and G-NP-HVDF all fail to schedule tasksets at a load of three for some cases, which is expected as they provide only best-effort scheduling. G-FIFO performs similarly to `SCHED_FIFO`; and both fail to schedule all tasksets at a load of 2 in the BLU case and are never able to schedule all tasksets over a load of 5. G-NP-EDF is able to schedule some taskset up to a load of 7 for all cases, but is also never able to provide full schedulability at a load higher than 5. This is consistent with its expected performance as a non-preemptive algorithm. P-EDF performs near G-EDF and C-EDF in most cases; it slightly outperforms them for the BMB and BLB cases, but is outperform by both for the BHB and BHU cases, because of the bin-packing problems associated with partitioning heavy tasksets. P-DASA-ND and P-LBESA perform identically to P-EDF in the BHU and BLU cases. In all other cases, their performance begins degrading one load point before P-EDF, likely due to their increased scheduling overhead. G-RMS and P-RMS are slightly outperformed by G-EDF and P-EDF in all cases, as is expected with static priority algorithms.

8.7.2 16-Core Schedulability Results

The scheduling results are similar to those on the 8-core, but there are a few noticeable differences. 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 in the BHB, BHU, BMB, and BLB cases. `SCHED_FIFO`, G-FIFO, G-GUA, G-HVDF, and G-NP-HVDF show widely varying performance, and are only able to meet all deadlines consistently under low loads. P-EDF, P-LBESA, and P-DASA-ND all demonstrate nearly identical performance, as expected. They also all show performance in keeping with the difficulties of the bin-packing problems involved; the load at which they are able to successfully schedule all taskset is proportional to the average task weight, and ranges from 10 for the BHU case to 15 for the BLU case. Also, as expected, the partitioned deadline-based algorithms perform 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.

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 0.5 later than NG-GUA. This is consistent with the effects of the overhead difference between the two algorithms.

8.7.3 48-Core Schedulability Results

On the 48-core, four results stand out. First, under most conditions, G-NP-EDF is able to successfully schedule the most tasksets of any global scheduling algorithm. Second, G-FIFO and `SCHED_FIFO` outperform G-EDF and G-RMS on four of the six distributions. Third, in all six distributions, G-FIFO outperforms `SCHED_FIFO`. They are quite close for the heavier distributions, but when the number of tasks increases, the performance of `SCHED_FIFO` degrades significantly; for the BLU distribution, it is not able to provide full schedulability even at a load of 1. Fourth, P-EDF and C-EDF clearly provide the highest performance. Neither algorithm fails to provide full schedulability for any case at a load less than 28. P-EDF is especially effective, scheduling all tasksets up to a load of 43 for the BLB and BLU cases.

There are several other results worth noting. P-EDF and P-RMS exceed all of their theoretical bounds. G-EDF exceeds its theoretical bounds for the BHU and BMU cases, but falls 47% short of its bound for the BLU case. G-RMS exceeds its BLU bound, but since its is implemented in the same manner as G-EDF, this likely means that the bound is significantly pessimistic. Despite this, the performance of both algorithms is quite impressive, considering the rigorousness of the test and their implementation. None of the global utility accrual algorithms are able to provide full schedulability at a load greater than 19. 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, as is shown in Chapter 9.

8.8 DSR Results

The deadline satisfaction ratio results of our tests are shown in Appendix B, due to their length.

8.8.1 8-Core DSR Results

We note the following about the DSR results on the 8-core platform. G-EDF and C-EDF both maintain a DSR of over 98% up to a load of 7 for all taskset distributions. P-EDF is not as consistent, meeting 97.5% of deadlines at a load of 8 for the BLU case, but meeting only 72% at a load of 7 for the BHU case. This is because the BLU case is the easiest to partition, while the BHU case is the hardest. G-EDF and C-EDF exhibit traditional deadline-based scheduling behavior, experiencing a rapid loss of deadlines once the system becomes fully loaded. This rapid loss is commonly known as the "domino effect". P-EDF shows this only for the lighter tasksets. For the heavier tasksets, the uneven loading caused by partitioning means that not all cores become overloaded at the same time. Interestingly, although C-EDF outperformed G-EDF in schedulability in most cases, in only outperforms G-EDF in DSR for the BLU case. In all other cases, though both algorithms begin missing a significant number of deadlines at the same point, C-EDF's DSR drops faster. P-RMS and G-RMS both begin missing deadlines earlier than any of the deadline algorithms, but also miss less of them in overload. For all distributions, G-RMS meets between 84% and 87% at a load of 10, the highest of all algorithms measured.

For the utility accrual algorithms, we see that behavior is as expected. NG-GUA and gMUA both are able to meet a higher percentage of deadlines in underload than G-GUA and G-HVDF, but meet less in underload. In four of the first five taskset distributions, G-GUA is able to meet more deadlines than G-HVDF at all loads. The exception is the BHU cases, where their performance is almost identical. In the first five distributions, G-NP-HVDF meets less deadlines than all other UA schedulers at all points. However, this changes for the BLU case. In this case, the large overheads of gMUA, NG-GUA, and G-GUA inhibit their best-effort performance. At a load of 10, none are able to meet even 40% of deadlines. G-HVDF and G-NP-HVDF both meet a significantly larger number: 73% and 79% respectively. Among the partitioned UA algorithms, P-DASA-ND always achieves the highest DSR, followed by P-LBESA and then P-HVDF. As expected, P-DASA-ND and P-LBESA always lie nearly on top of P-EDF until it begins missing deadlines, and then maintain a higher DSR. Except for the BLU case, none of the three ever exceed the performance of G-GUA in overloads.

8.8.2 16-Core DSR Results

On the 16-core platform, we observe that G-EDF, G-NP-EDF, G-FIFO, and G-RMS perform as expected. G-EDF and C-EDF meet over 99% of deadlines until a load of 14 for all distributions. For both algorithms, we see the domino effect begin to occur between a load of 15 and 16 in all distributions. As expected from the schedulability results, C-EDF maintains a higher DSR than G-EDF at a load of 16 for the BMB, BMU, BLB, and BLU cases, but looses deadlines more rapidly for the BHB and BHU cases. G-NP-EDF generally begins dropping deadlines slightly earlier than G-EDF. Like G-EDF, G-NP-EDF maintains a DSR over 99% for the BHB, BMB, BMU, BLB, and BLU distributions at a load of 14. For the BHU distributions, G-NP-EDF meets only 97%. However, because of its non-preemptive nature, the domino effect is not as pronounced. G-FIFO and `SCHED_FIFO` experience a drop in deadlines between loads of 8 and 10 for all distributions, but this decrease is slower, due to the fact that tasks are not preempted. G-RMS maintains the highest DSR of any algorithm, never meeting less than 89% of deadlines under any case.

Among the utility accrual algorithms, we observe that G-HVDF, G-NP-HVDF, and G-GUA consistently begin missing deadlines between a load of 10 and 12, while gMUA and NG-GUA begin missing a significant number of deadlines between 12 and 14. However, gMUA and NG-GUA both loose deadlines far more rapidly once they begin to lose them; neither meets more than 82% at a load of 16 for any distribution. Under the BLU distribution, gMUA, NG-GUA, and G-GUA begin loosing deadlines rapidly between 6 and 8. Rather than providing the best-effort performance they are designed to provide, all three algorithms meet less than 5% of deadlines at full load for the BLU case. In contrast, for the BLU case G-NP-HVDF is able to meet over 90% of deadlines at full load, and looses deadlines the slowest of all the utility accrual algorithms.

We observe that P-EDF, P-LBESA, and P-DASA-ND all perform nearly identically under all distributions until they start loosing deadlines. After they begin loosing deadlines, P-EDF's DSR drops faster than either P-LBESA or P-DASA-ND. For the BHU case, P-EDF's DSR drops in a linear manner, rather than the domino-effect failure pattern typically associated with deadline-based scheduling. This happens because the bin-packing associated with partitioning implies that not all cores will undergo the domino effect at the same time. G-HVDF begins missing deadlines significantly before any other partitioned algorithms (at a load of 2 in the BLU case). P-RMS always begins missing deadlines slightly earlier than P-EDF. However, P-RMS and P-HVDF lose deadlines at a slower pace than the other algorithms. The DSR curve for P-RMS is generally around 10% higher than the curve for P-HVDF for all cases, while their slopes are generally similar.

8.8.3 48-Core DSR Results

The DSR results on the 48-core show several clear trends. G-GUA, NG-GUA, gMUA, and G-HVDF do not provide a graceful performance degradation. All four show a sharp drop in DSR, followed by a period of more level performance. Of the four, G-HVDF consistently provides the highest DSR, followed by gMUA, then NG-GUA, then G-GUA. G-HVDF is never able to meet more than 50% of deadlines at full load. NG-GUA and gMUA never exceed 30% at full load, and G-GUA never surpasses 15%. In contrast, G-NP-HVDF never meets less than 70% of deadlines at a load of 48. The domino effect is clearly visible in G-EDF's DSR behavior; in all cases, it drops from above 90% to below 30% in a load range of three or less. This drop trails the drop in schedulability by a significant margin in some cases; in the BLU case, the curves coincide, while for the BLB case the schedulability drops at a load of 14 and the DSR drops at 31. This gap is due to two factors. First, overheads may cause temporary delays which push only a few tasks beyond their deadlines. Second, This gap is most pronounced for the bimodal cases, implying that we are seeing a few heavy tasks failing to meet their deadline because of interference from lighter tasks. This allows the DSR to remain high even though the taskset is not completely scheduled. For all cases, G-FIFO and `SCHED_FIFO` begin loosing deadlines slowly between a load of 20 and 30, and then begin to rapidly miss deadlines between 40 and 44. C-EDF and G-NP-EDF perform better, meeting over 95% of deadline until a load of between 37 and 44, and then rapidly loosing deadlines after that. In all but the BLU case, G-NP-EDF is able to maintain a DSR over 90% until at least 45. For all cases, G-NP-EDF has a higher DSR than G-EDF.

Among the partitioned algorithms, P-RMS consistently meets the largest number of deadlines at a load of 48, and P-EDF the least. Aside from the BLU case, P-DASA-ND meets a larger number of deadlines than P-LBESA, and both begin missing deadlines at the same point as P-EDF. P-HVDF generally has the same slope as P-RMS in the 40-48 load range, but meets around 10% less deadlines. Both have a slope roughly half as steep as P-LBESA and P-DASA-ND in this range.

8.9 AUR Results

The deadline satisfaction ratio results of our tests are shown in Appendix C, due to their length. Since the AUR for deadline, period, and FIFO based algorithms is directly tied to their DSR, we provide only a limited discussion of those algorithms here. Furthermore, since utility accrual is only important metric in high-load and overloaded systems, we only analyze the results in these areas.

8.9.1 8-Core AUR Results

On the 8-core, G-HVDF provides the highest AUR in overload for all tasksets distributions except BLU, where it accrues 4.7% less utility than G-NP-HVDF. At a load of 10, G-HVDF never accrues less than 86% of the possible utility. For the three heaviest distributions (BHB, BHU, and BMB) G-GUA performs second, accruing between 79% and 85% at a load of 10. For the lightest three distributions (BMU, BLB, and BLU) G-NP-HVDF performs second, accruing between 87% and 95% utility at a load of 10. NG-GUA and gMUA accrue less utility in overload than the other global UA algorithms for the first five distributions. At a load of 10, neither algorithm ever exceeds 78% utility. For the BLU distribution, as has been noted in other sections, G-GUA, NG-GUA, and gMUA do not perform as intended for the BLU case due to their high overhead, and in fact accrue less utility than G-FIFO and `SCHED_FIFO`.

Among the partitioned UA algorithms, G-HVDF always accrues the least utility in underload conditions, but the most utility in overload. P-DASA-ND and P-LBESA both perform similarly, accruing more utility than G-HVDF in underloads, then dropping rapidly between a load of 8 and 9, and then leveling off between 9 and 10. For the first five distributions, P-DASA-ND accrues more utility in all cases, but never exceeds P-LBESA by more than 3.6%. For the BLU case, both perform identically in underloads, but P-LBESA begins to pull ahead in overloads. At a load of 10, it accrues 6.8% more utility than P-DASA-ND. In no case do P-DASA-ND or P-LBESA accrue more utility than G-HVDF during overloads.

8.9.2 16-Core AUR Results

For the 16-core, we observe the following. First, all of the non utility-accrual algorithms (`SCHED_FIFO`, G-FIFO, G-RMS, G-EDF, C-EDF, P-RMS, P-EDF) demonstrate utility accrual performance directly related to their DSR performance. This is to be expected, since they do not consider the utility of tasks when scheduling.

Second, we observe that P-HVDF outperforms P-LBESA by up to 19% and outperforms P-DASA-ND by up to 23% at a load of 16. P-HVDF is never outperformed by P-LBESA at this load. P-DASA-ND only outperforms P-HVDF once at this load; it accrues 1.6% more utility than P-HVDF for the BMB distribution.

For the global utility accrual algorithms, we see that gMUA and NG-GUA always accrue the least utility at high loads. Neither algorithm has an AUR of more than 80% at full load. Interestingly, gMUA generally accrues less utility than NG-GUA, despite their functionality being similar. G-GUA only accrues less utility than NG-GUA at high load for the BMU and BLU distributions. This makes sense, because the overhead of G-GUA is much larger than that of NG-GUA and therefore even though it is theoretically superior, it is unable to actually achieve this performance. In high load situations, G-NP-HVDF and G-HVDF accrue significantly more utility than any of the more complex heuristics; neither algorithm ever accrues less utility than gMUA, NG-GUA, or G-GUA at a load of 16. The relative performance of G-NP-HVDF and G-HVDF is dependent on the weight of the tasksets. For BHU, the heaviest distribution in the system, G-HVDF accrues 8.9% more utility at a load of 16. However, for the BLU case, which is the lightest distribution tested, G-NP-HVDF accrues 25% more utility than G-HVDF. Overall, for all tasksets considered, G-NP-HVDF is able to accrue the most utility.

8.9.3 48-Core AUR Results

Our observations about the AUR results for the 48-core are similar to those made about the DSR results. G-GUA, NG-GUA, gMUA, and G-HVDF fail to provide a graceful decline in performance. They are consistently outperformed by both the partitioned algorithms and G-NP-HVDF. At full load, G-NP-HVDF accrues at least 8% more utility than any other algorithm for all cases except BLU. In the BLU case, P-HVDF accrues 1% more utility. P-HVDF always accrues at least 5% more utility than either P-LBESA or P-DASA-ND at full load. As noted before, P-DASA-ND slightly outperforms P-LBESA for all cases except BLU, where the pessimistic approach of P-DASA-ND hinders it.

8.10 Tardiness Results

The mean maximum tardiness results of our tests are shown in Appendix D, due to their length. Since fully abortive algorithms (G-GUA, NG-GUA, gMUA, G-HVDF, P-HVDF, P-LBESA, P-DASA-ND) abort tasks as soon as they exceed their deadlines, they should never allow tasks to display unbounded tardiness. Our discussion of their tardiness results is therefore limited.

Tardiness bounds are computed for G-NP-EDF and G-EDF using the methods discussed in 6. We know that G-FIFO also bounds tardiness, but since the only method available requires a specific evaluation of each task in each taskset, we do not compute a bound for G-FIFO. The bound's authors experiments show that the bound for G-FIFO is always higher than that for G-NP-EDF and G-EDF. In their experimental results, they found the bound for GFIFO to always be between two and ten times that of the deadline-based algorithms. Based on this, we analyze G-FIFO's experimental results for reasonableness. Furthermore, we must note that overheads have not been considered in the system load, and so on each platform, full load actually occurs somewhere between a load of m and $m - 1$. Application overhead is calculated in 9.

Table 8.5 shows computed tardiness bounds for G-EDF and G-NP-EDF computed using these methods. It should be noted that the bound computed for G-EDF is known to provide unreasonable bounds when both m and average per-task utilization are high, which is likely why the bounds for the BHU and bimodal distributions are so high. It must also be noted that since these bounds do not account for overhead, none of our algorithms ever meet them. Nonetheless, they provide an idea of the behavior we should observe.

Table 8.5: Tardiness bounds for G-EDF and G-NP-EDF (microseconds)

Algorithm	Cores	BHU	BMU	BLU	All Bimodal
G-EDF	8	313076	88214	19445	332269
	16	472352	96730	20267	487029
	48	723333	103175	20827	730893
G-NP-EDF	8	444118	69616	11046	423571
	16	606000	73000	11128	576060
	48	799123	75411	11184	757978

Since we cannot easily compute bounds for all of the tardiness-bounding algorithms (`SCHED_FIFO`, G-FIFO, G-EDF, G-NP-EDF, C-EDF, P-EDF), we also measure tardiness by visually analyzing each graph to identify the point at which tardiness begins to rapidly increase. All of the mentioned algorithms should provide relatively low tardiness up until they enter overload, at which point their tardiness should begin increasing rapidly. The results of this analysis are shown at the end of each section.

8.10.1 8-Core Tardiness Results

On the 8-core, G-RMS and P-RMS always incur the largest MMT. This is expected, since both are fixed priority algorithms. For all distributions, once the system is overloaded the maximum observed tardiness rises above 1 second — the length of the test. This implies that a task was blocked for the entirety of the test. G-NP-HVDF is also a fixed priority algorithm, and like the period-based algorithms, tasks suffer unbounded tardiness. However, this is consistently lower than either of the periodic algorithms because G-NP-HVDF aborts tasks, lowering the system load. After these three algorithms come the two FIFO-based algorithms — G-FIFO and `SCHED_FIFO`. Both perform nearly identically, incurring a maximum mean tardiness of between 600,000 and 900,000 μs at a load of 10 for all distributions. These algorithms should bound tardiness at full load, and do for the four heaviest distribution. For the two lightest distributions (BMU and BLU) we see that tardiness is bounded at a load of 7, but not at a load of 8. For all cases, G-EDF, G-NP-EDF, P-EDF, and C-EDF demonstrate a tardiness of between 200,000 and 400,000 μs at a load of 10. Both G-EDF and G-NP-EDF bound tardiness under all distributions at full load. P-EDF and C-EDF bound tardiness at full load for all cases except the BHU case. In this case, tardiness cannot be bounded at full load because there is no viable partitioning for the tasksets. Table 8.6 shows the load points after which algorithms which bound tardiness are in overload.

Table 8.6: Last underload point for tardiness-bounding algorithms on our 8-core platform

Algorithm	BHB	BHU	BMB	BMU	BLB	BLU
SCHED_FIFO	8	8	8	7	7	7
G-FIFO	8	8	8	7	7	7
G-EDF	8	8	8	8	8	8
G-NP-EDF	8	8	8	8	8	8
C-EDF	7	7	8	8	8	8
P-EDF	7	6	7	7	8	8

8.10.2 16-Core Tardiness Results

We note several things about the tardiness results on the 16-core platform. First, at no time to any of the fully abortive algorithms show a tardiness of more than 20 microseconds. Second, G-RMS, P-RMS, and G-NP-HVDF all demonstrate completely unbounded latencies, with latencies approaching 1 second in some tests. This makes sense, since all three are static priority algorithms, and therefore cannot bound tardiness by definition. Of the three, G-RMS demonstrates the highest latencies measured, with latencies for the BLU case reaching 1.175 seconds. Since our tests are only one second long, this implies that a task has been blocked for the entire duration of the test and part of the clean-up period. G-EDF, C-EDF, and G-NP-EDF all demonstrate bounded latencies. Based on this, we conclude that G-NP-EDF, G-EDF, and C-EDF all demonstrate sufficient performance to state that they provide soft real-time scheduling on this platforms. P-EDF demonstrates tightly bounded latencies in all but the heavy taskset cases. In these cases, the bin-packing problems dictate that certain cores will be overloaded significantly before the total load on the system reaches overload. This implies that neither soft nor hard real-time schedulability is possible. Table 8.7 shows the load points after which algorithms which bound tardiness are in overload.

Table 8.7: Last underload point for tardiness-bounding algorithms on our 16-core platform

Algorithm	BHB	BHU	BMB	BMU	BLB	BLU
SCHED_FIFO	15	15	15	15	15	15
G-FIFO	15	15	15	15	15	15
G-EDF	15	15	15	15	15	14
G-NP-EDF	15	15	15	15	15	15
C-EDF	14	14	15	15	15	15
P-EDF	13	11	15	15	15	15

8.10.3 48-Core Tardiness Results

In the 48-core tardiness results, there are two key facts to note. The first is that for all cases except BLU, G-NP-EDF bounds tardiness at a higher load than any other algorithm. While it does fail to meet its theoretical bounds, its high performance implies that it scales to our 48-core platform. The second fact to note is that for the BMB, BMU, BLB, and BLU cases, C-EDF and P-EDF also bound tardiness at quite high loads. In the other two cases, their performance is limited by the partitioning.

Other results worth nothing are that for the first five cases, G-EDF is able to bound tardiness below a load of 34 for the first five distributions. It is also able to meets its computed tardiness bounds for the BHB, BHU, BMB, and BLB distributions, although these are not tight bounds. Additionally, for the first five cases G-FIFO is able to provide tardiness below 100ms below a load of 44, and always provides a lower MMT than `SCHED_FIFO`. Last, for the BLU case, not even fully aborting algorithms can bound tardiness; G-GUA experiences latencies in excess of 1.4 seconds, implying that tasks are being released faster than they can be aborted. Table 8.8 shows the load points after which algorithms which bound tardiness are in overload.

Table 8.8: Last underload point for tardiness-bounding algorithms on our 48-core platform

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	38	44	43	44	42
P-EDF	39	33	44	43	44	46

8.11 Migrations and Abortion

The migration and abortion results of our tests are shown in Appendix E, due to their length. Since partitioned algorithms (P-RMS, P-EDF, P-HVDF, P-LBESA, P-DASA-ND) make no migrations, they are omitted from the results. P-HVDF, P-LBESA, and P-DASA-ND do abort tasks, so their abortion results are grouped together with those of the utility accrual algorithms.

8.11.1 8-Core Migration Results

In analyzing the migration results on the 8-core platform, we can see the following. First, the three non-preemptive algorithms (G-FIFO, G-NP-EDF, and G-NP-HVDF) always perform the least migrations, and for each distribution, all perform a around the same number of migrations. Second G-RMS always performs around 24% more migrations than G-EDF. C-EDF generally performs around 15% less migrations than G-EDF. This makes sense, because though the actual algorithm logic is similar, the lower number of cores in a cluster implies a higher probability of a selected task already being on the correct core. G-HVDF always performs a number between G-RMS and G-EDF. In the lighter distributions, G-GUA performs significantly more migrations than any other algorithm; however, on the heavier distributions, it performs around the same number as G-HVDF. Conversely, for the lighter distributions, NG-GUA and gMUA perform similar numbers of migrations to G-HVDF, while for the heavier distributions they perform more. While both perform a similar number of migrations to G-EDF in all cases, this is coincidence, since their behavior in overloads (and therefore their migration patterns) are different.

8.11.2 16-Core Migration Results

We observe the following about the migration results on the 16-core platform. First, non-preemptive algorithms such as G-FIFO and G-NP-EDF, perform a relatively low number of migrations. G-FIFO and G-NP-EDF migrate around the same number of tasks, while G-NP-HVDF always migrates less tasks than them because it aborts tasks and aborted tasks are never migrated. Second, G-RMS and G-EDF both consistently perform more migrations than the concurrent architecture schedulers. G-EDF performs between 1% and 8% less migrations than G-RMS for all cases except the BHU case. C-EDF always performs less migrations than G-EDF but more than G-NP-EDF. This is due to its clustered nature, which increases the probability that a task will not need migrated. Because it is clustered, these migrations will also be less expensive on average than the migrations made by global schedulers. NG-GUA and gMUA both perform slightly more migrations than G-RMS and G-EDF, despite aborting tasks. The only exception to this is the BLU case, where NG-GUA and gMUA abort enough segments that they migrate less. G-GUA performs around 40%

less migrations than gMUA and NG-GUA for the two heavy tasksets. However, for the BMU and BLU tasksets it performs up to 77% more migrations.

8.11.3 48-Core Migration Results

On the 48-core, NG-GUA and gMUA no longer perform similar numbers of migrations to G-RMS and G-EDF. For most cases, they perform almost twice as many, while for the BLU case they only perform one third as many. We do note that the relationship between G-RMS and G-EDF, and C-EDF remains unchanged; G-RMS performs around 10% more migrations than G-EDF, which performs around 10% more than C-EDF. Also, as before, G-FIFO and G-NP-EDF perform less than any of the three. G-NP-HVDF once again performs slightly less than G-NP-EDF for all cases.

8.11.4 8-Core Abortion Results

On the 8-core, there are two primary observations we may make about the abortion results. The first is that the abort counts for G-GUA, gMUA, NG-GUA, P-LBESA, and P-DASA-ND are roughly consistent across all three bimodal distributions, despite the fact that the BLB distribution creates nearly twice the tasks of the BHB distribution. In fact, G-GUA and P-DASA-ND both show around a 5% decrease in abort count between the BHB and BLB distributions. This is due to the fact that while the task count increases, the average task weight decreases, making it easier for these algorithms to perform their bin-packing. Hence, they are able to successfully schedule a larger percentage of the tasks. This is born out in the DSR results, where we see NG-GUA, gMUA, P-DASA-ND and P-LBESA achieving higher DSR numbers for BMB than for BHB, and yet still higher numbers for BLB. This is further evidenced by the opposite behavior from the simple heuristics; P-HVDF, G-HVDF, and G-NP-HVDF show increasing abort counts as the task count increases. This also explain why G-GUA aborts less tasks than gMUA and NG-GUA for all five distributions without runaway aborts; since it performs a more rigorous bin-packing method, it is able achieve a higher DSR in overload than NG-GUA and gMUA, which is where the vast majority of aborts occur. While it performs more in underload, this number is dwarfed by the number produced in overload, and so its total abort count is less.

The second behavior is that, for all except the BLU case where gMUA, G-GUA, and NG-GUA perform runaway aborts, P-HVDF performs the highest number of aborts of any algorithm. This is because it is a partitioned algorithm, and because the partitioning was done with no respect to value-densities, but in order of decreasing utilization. This implies that each core has a smattering of both light and heavy utilization tasks in these distributions. Therefore, if several low utilization tasks are assigned to the same core as a single high value-density high utilization task on a fully loaded or overloaded core, most of them may be aborted.

8.11.5 16-Core Abortion Results

Several interesting behaviors may be observed in our abortion results from the 16-core platform. The most important of these is that the abort counts for G-NP-HVDF, G-HVDF, gMUA, and NG-GUA actually decreases for the BHB, BHU, and BMB cases relative to the 8-core results. The reason for this is that we test the 8-core platform further into overload than the 16 core platform. The second observation is that unlike on the 8-core, G-GUA performs more aborts than NG-GUA in most cases. When combined with the DSR results, we see that for most cases, it both meets more deadlines and aborts more tasks than NG-GUA. This difference is due to the fact that G-GUA begins aborting tasks at a lower load than NG-GUA. We also observe that, unlike the 8-core results, gMUA and NG-GUA have significantly different abort counts for the bimodal cases. This is due to the overhead differences between the two algorithms. gMUA's lower overhead means less scheduling events conflict, and so it performs more optimistically. G-NP-HVDF always performs less aborts than any other global algorithm, but performs more aborts than P-DASA-ND and P-LBESA. These two algorithms exhibit an unusual behavior; all the other algorithms shown perform an increasing number of aborts as the number of tasks increase. However, because more tasks implies smaller tasks, which allows for better partitioning, both of these algorithms actually perform half as many aborts for the BLU case as they do for the BHU case.

8.11.6 48-Core Abortion Results

The results for the 48-core platform are more consistent than they were for the 16-core platform. G-GUA performs the most aborts in all cases, aborting 41% of segments in the BHU case and 91% of segments in the BLU case. Since this number includes segments from low loads in which no segments were aborted, we find that for the BLU case at a load of 48, G-GUA is aborting over 99% of the segments, most of which are aborted before they begin their execution. For the first five distributions, NG-GUA and gMUA abort between 60% and 80% of the number of tasks that G-GUA aborts. However, for the BLU case, both algorithms abort over 97% of the segments G-GUA aborts. G-HVDF performs less aborts than gMUA, NG-GUA, and G-GUA for all cases, but as with them, its performance seriously degrades for both the BMU and BLU cases. As with the previous platforms, G-NP-HVDF performs less aborts than any other global algorithm. However, on the 48-core, this different is enlarged; G-NP-HVDF performs at least one order of magnitude less aborts for all distributions.

Of the partitioned algorithms, no algorithm consistently aborts more segments than another. However, there are several predominant trends; first, for all except the BLU distribution, P-DASA-ND aborts less tasks than P-LBESA. The difference in the BLU case is likely due to the fact that P-DASA-ND incurs a higher overhead than P-LBESA. Also, unlike the other partitioned algorithms, P-HVDF aborts an increasing number of segments as the number of tasks increases. The last trend is that all three partitioned algorithms perform less aborts in all conditions than any global algorithm except G-NP-HVDF.

Chapter 9

System Measurements

In order to understand the limitations of ChronOS, we must understand various sources of overhead and inaccuracy in the system. To accomplish this, we measure a variety of overheads and possible sources of inaccuracy 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. This provides single-cycle resolution and allows for the fine-grain measurements we need. As is good practice, we preface all `RDTSC` instructions with a `CPUID` instruction to prevent instruction reordering [41].

9.1 Scheduling Overheads

The most obvious source of overhead in real-time scheduling is the time cost of performing the scheduling itself. Since the scheduler must be invoked at every scheduling event, its cost is of prime importance. This is especially true for algorithms such as G-GUA and NG-GUA that schedule in $O(mn^2)$ worst-case time.

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 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 overhead for all of the scheduling algorithms used in this thesis are shown in Figures 9.1 through 9.9. Note that for partitioned algorithms, we only measure up to 16 tasks for all machines because the partitioning means that for each distribution, each core will receive roughly the same number of tasks in a fully loaded system regardless of core count.

The results make several things clear. First, G-FIFO, G-RMS, P-RMS, G-EDF, C-EDF, P-EDF, G-NP-EDF, and G-NP-HVDF all provide $O(1)$ performance, as they should. Interestingly, the execution times of G-EDF and G-RMS fluctuate largely when there are less tasks than cores. The scheduling cost of G-EDF and G-RMS are largely dependent on the cost of accessing m task descriptors, most of which are on remote processors. This cost combined with the number of global scheduling events explains the poor scalability of these algorithms in some cases.

Also, as expected, despite the fact that C-EDF and G-EDF are in fact executing the same code, C-EDF is consistently faster due to the cache and memory access times associated with fetching task descriptors.

Second, the more advanced utility accrual algorithms are immensely expensive and demonstrate roughly linear performance. The linear performance is because while the asymptotic cost of gMUA, G-GUA, and NG-GUA is $O(mn^2)$ and that of P-LBESA and P-DASA-ND is $O(n^2)$, this only occurs under rare conditions. Despite not reaching their asymptotic bounds, these algorithms still incur enormous overheads. On the 48-core platform, G-GUA took a maximum of 1,541,297 cycles, or 906μs. Furthermore, all the other cores in the system are signaled and therefore will block during this time. Since the average execution time of a task in the BLU distribution is 2.78ms, and the average inter-event time measured for a BLU taskset with a load of 48 is 40μs, scheduling for 906μs is immensely problematic. Due to this, G-GUA only manages to schedule for 3.5% of the scheduling events for BLU tasksets with a load of 48. The costs of gMUA and NG-GUA are both significantly lower than G-GUA,

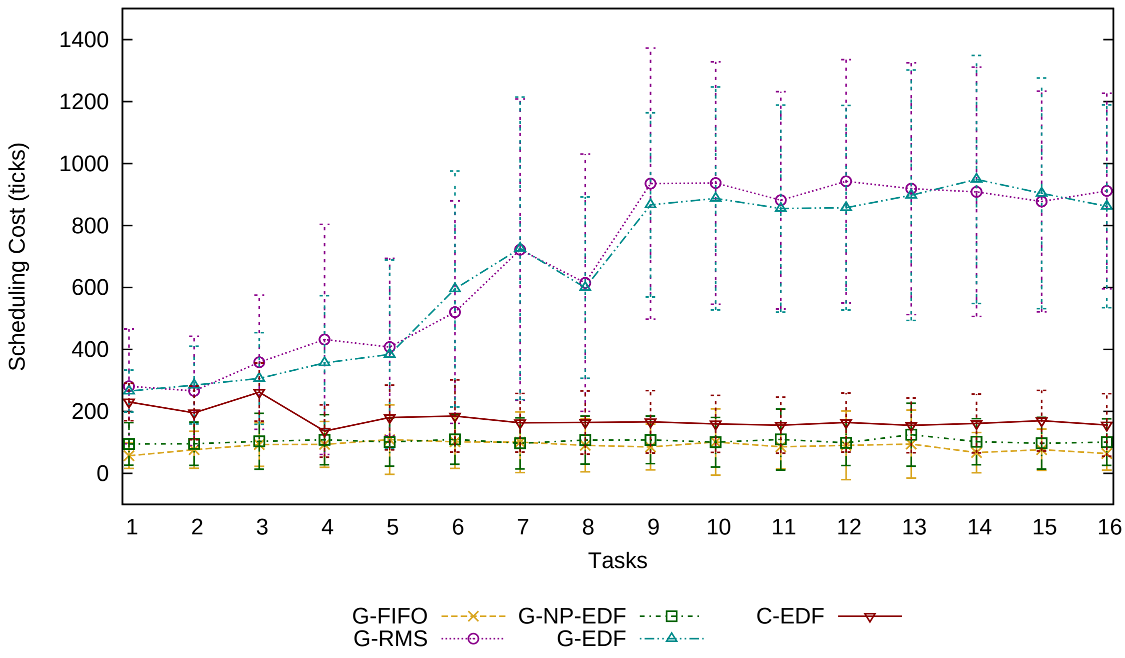


Figure 9.1: Scheduling cost of the traditional global scheduling algorithms on the 8-core platform

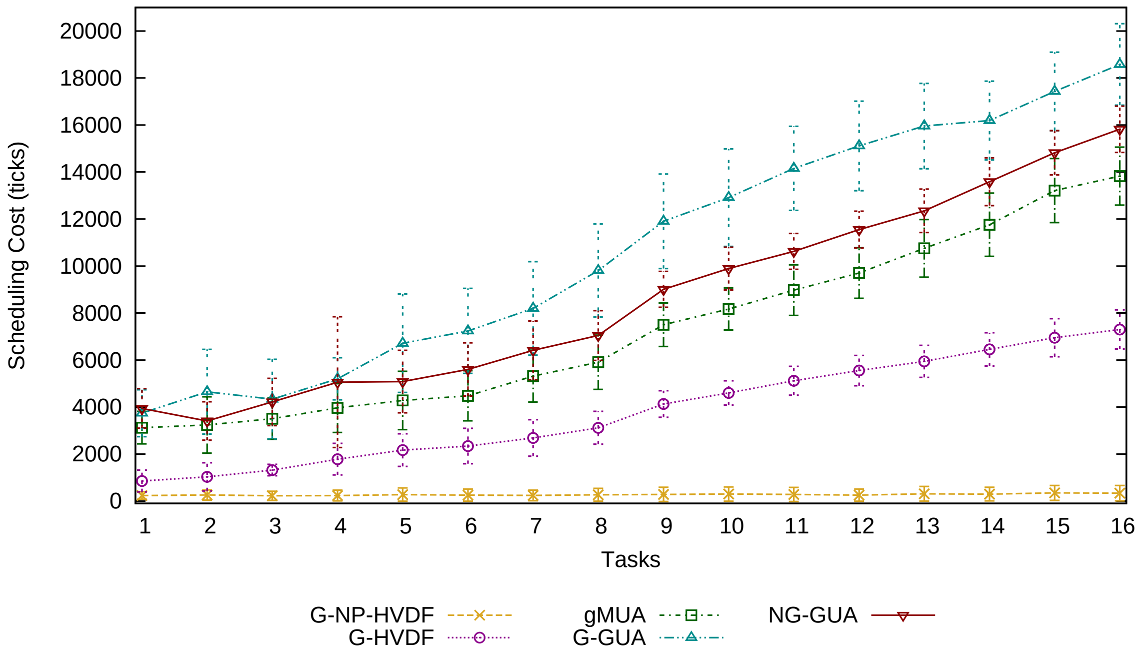


Figure 9.2: Scheduling cost of the global utility accrual scheduling algorithms on the 8-core platform

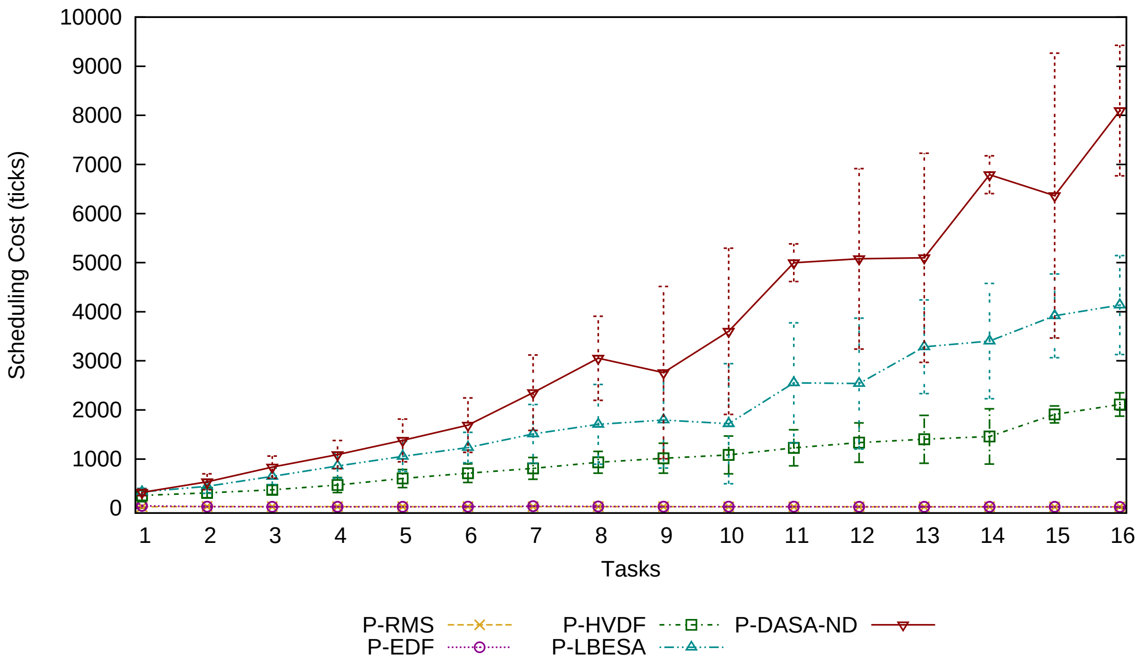


Figure 9.3: Scheduling cost of the partitioned scheduling algorithms on the 8-core platform

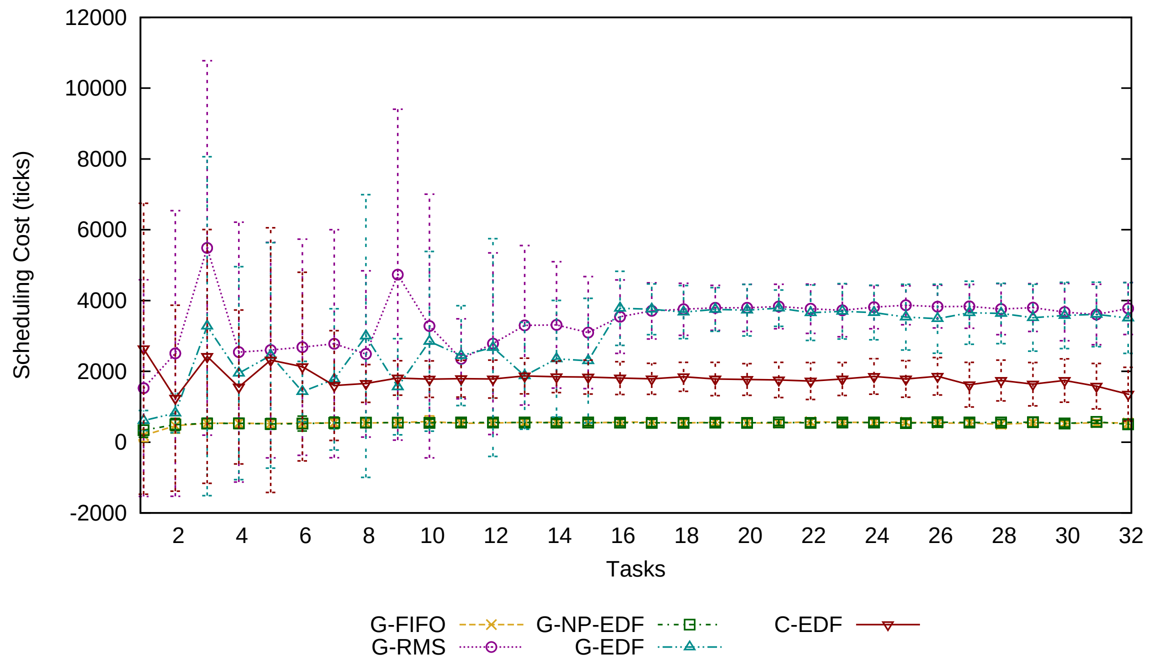


Figure 9.4: Scheduling cost of the traditional global scheduling algorithms on the 16-core platform

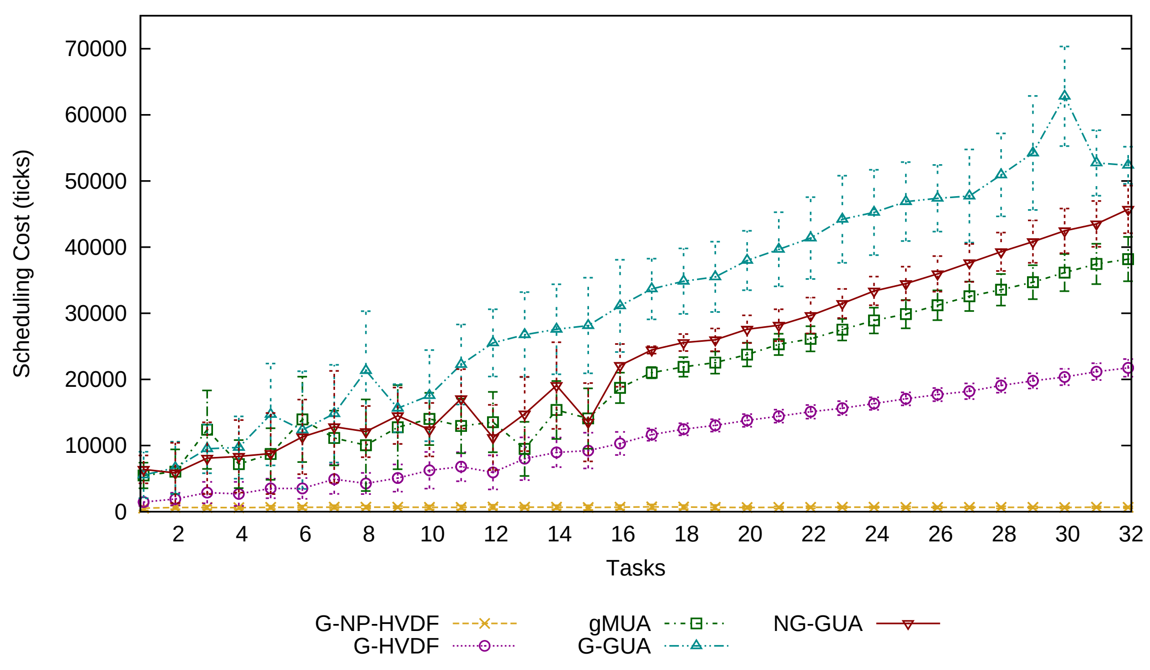


Figure 9.5: Scheduling cost of the global utility accrual scheduling algorithms on the 16-core platform

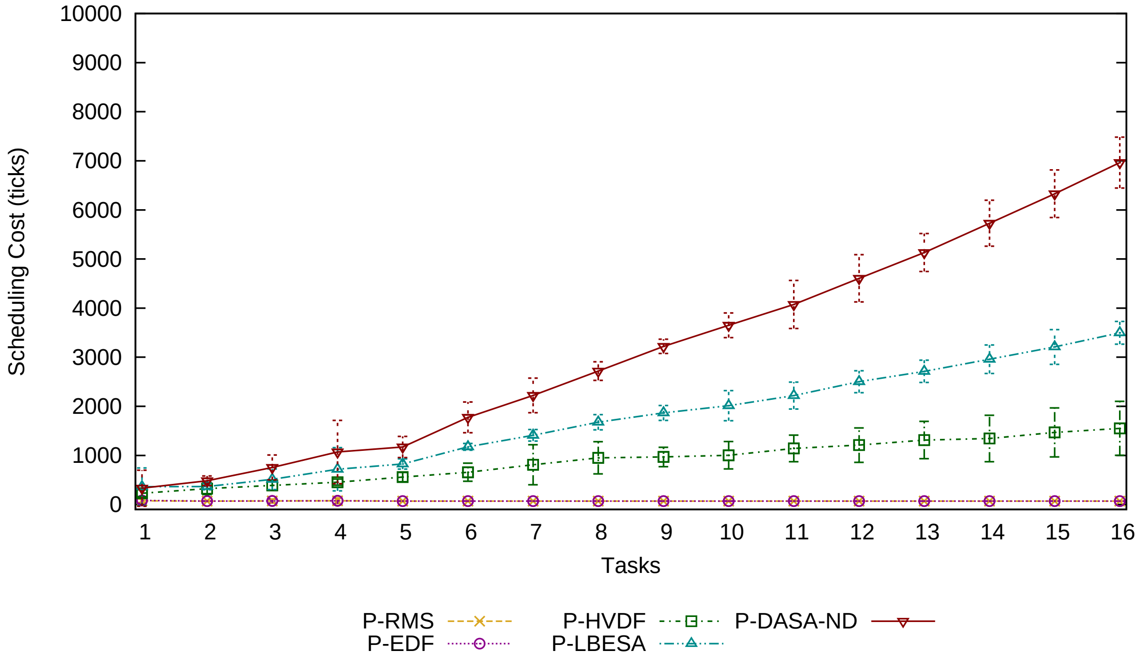


Figure 9.6: Scheduling cost of the partitioned scheduling algorithms on the 16-core platform

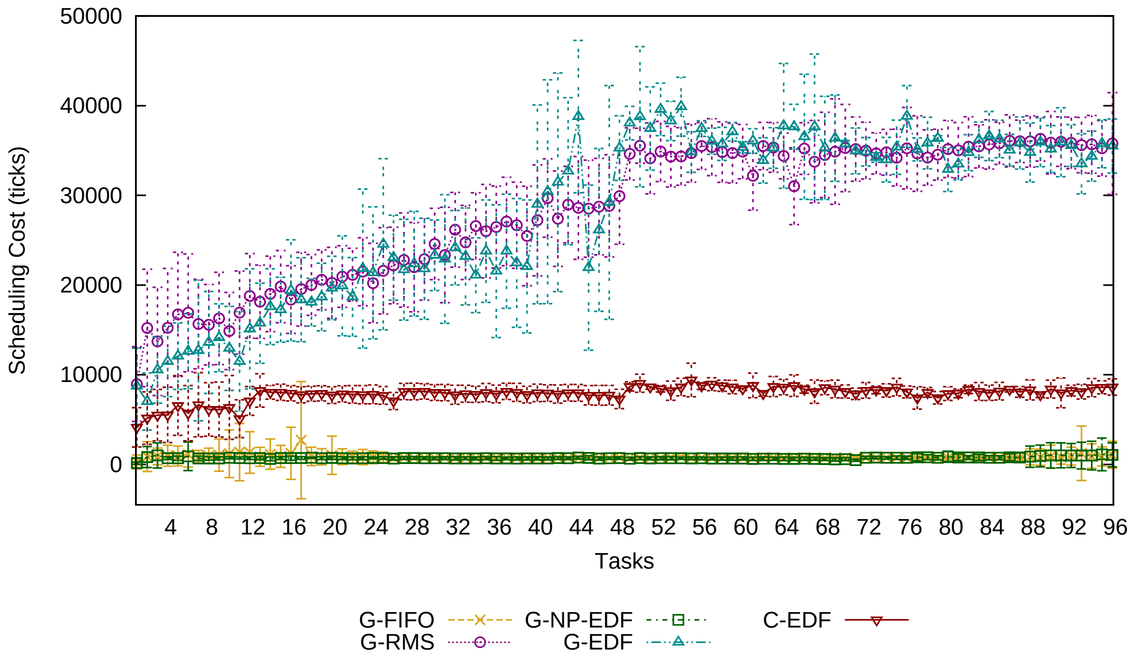


Figure 9.7: Scheduling cost of the traditional global scheduling algorithms on the 48-core platform

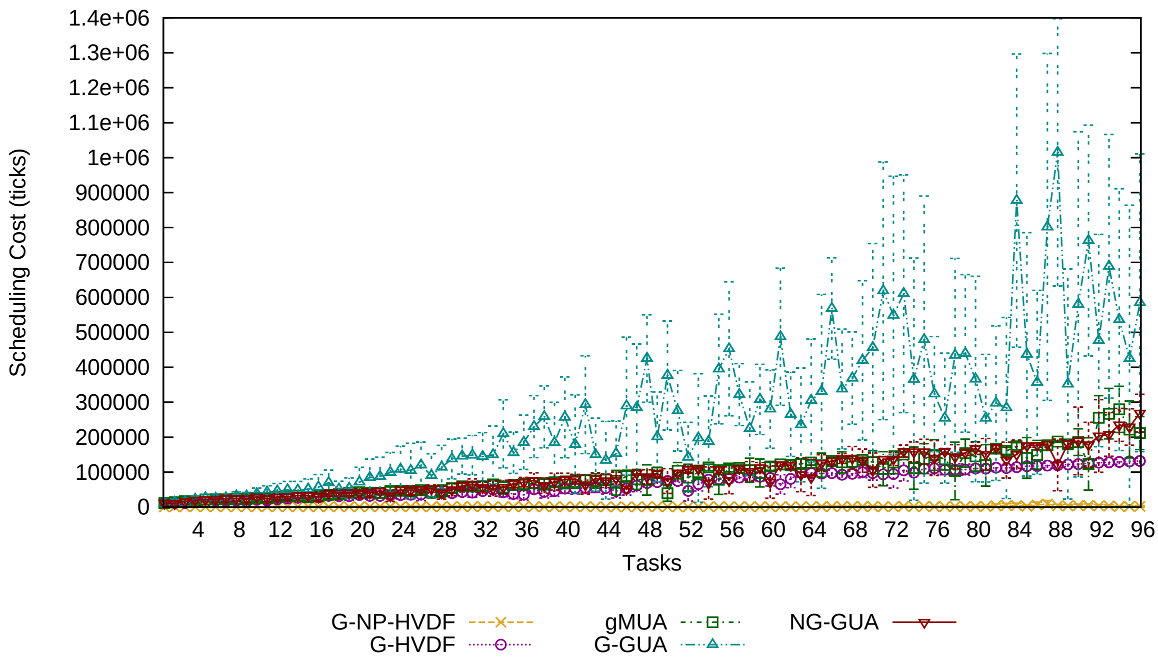


Figure 9.8: Scheduling cost of the global utility accrual scheduling algorithms on the 48-core platform

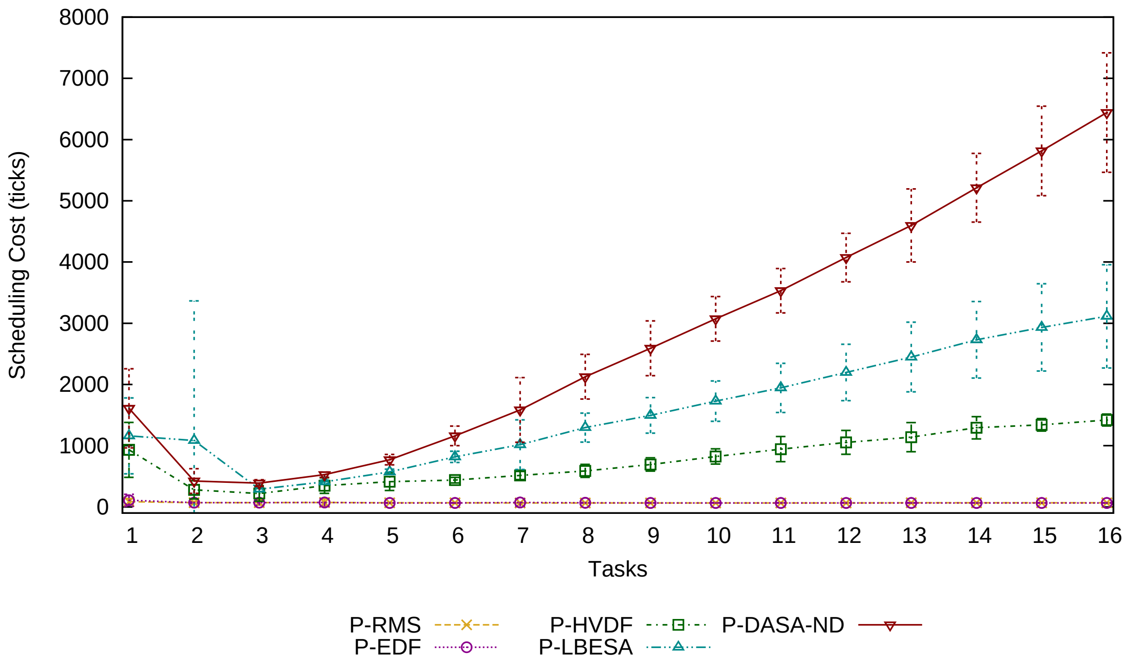


Figure 9.9: Scheduling cost of the partitioned scheduling algorithms on the 48-core platform

but both are still quite high. The maximum observed overheads for gMUA and NG-GUA are 371341 and 352552 ticks, respectively.

9.2 Preemption and Migration Overheads

Another source of system overhead is due to cache misses after a task is preempted or migrated. When a 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 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 the task is migrated between 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 therefore the cost of the cache misses. These measurements are performed for working sets of 1, 2, 4, 8, 16, 32, and 64 pages.

We measure the cost of three different migration paths. First, we measure the cost of migrating between two cores which share L3 cache, but not L2 or L1. Second, we measure the cost to migrate to a different processor, thereby loosing direct access to memory. On the 48-core platform, we test two variants of this path: migrating from processor 0 to processor 1 and to processor 2. Third, on our 16 and 48-core platforms, Each chip shares a bank of memory, but has two separate L3 caches, each for half of the processor's cores. Therefore, on these two platforms we test migrating between cores on the same processor which do not share L3 cache. This path is not measured on the 8-core, due to the cache architecture. On our systems, 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, both AMD and Intel implement sequential cache line prefetching [8], [55]. 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.

Figures 9.10, 9.11, and 9.12 show the cost of these migration routes on the various platforms.

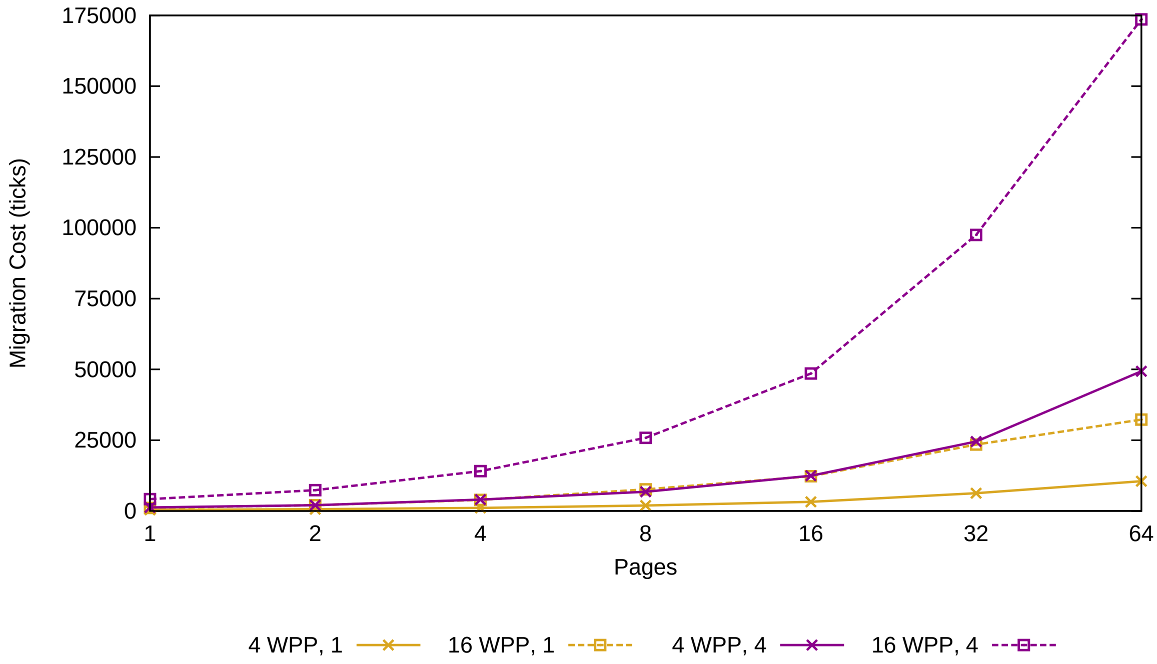


Figure 9.10: Migration costs of two migration paths on the 8-core

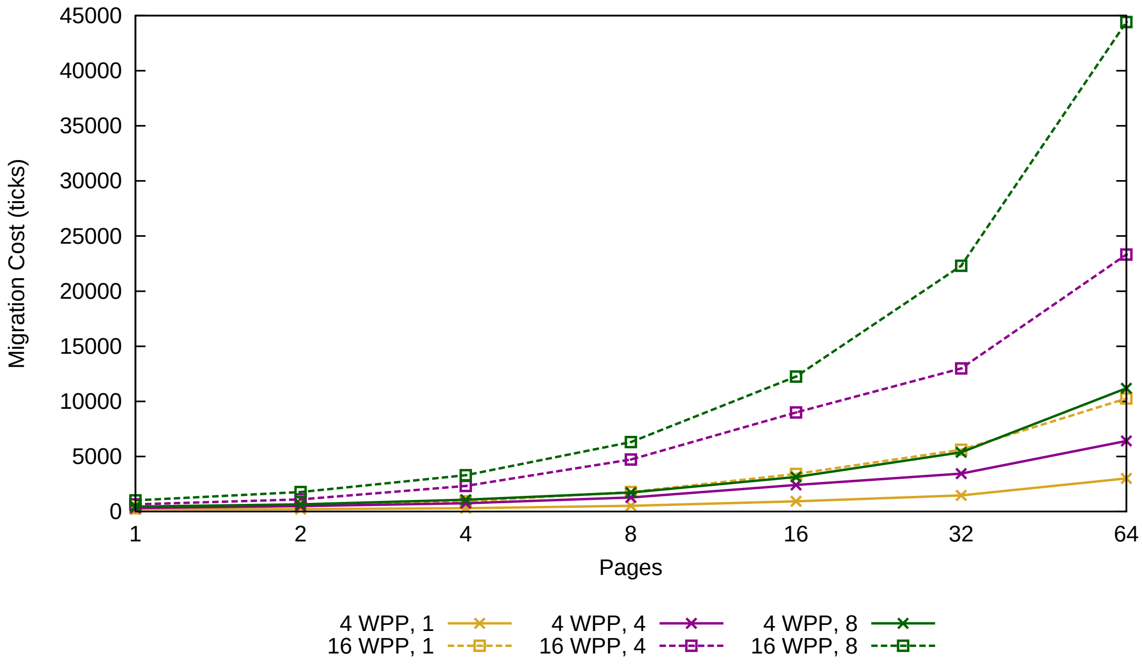


Figure 9.11: Migration costs of three migration paths on the 16-core

Although standard deviations were calculated, they never exceeded 72 ticks, and are therefore indistinguishable on the graph.

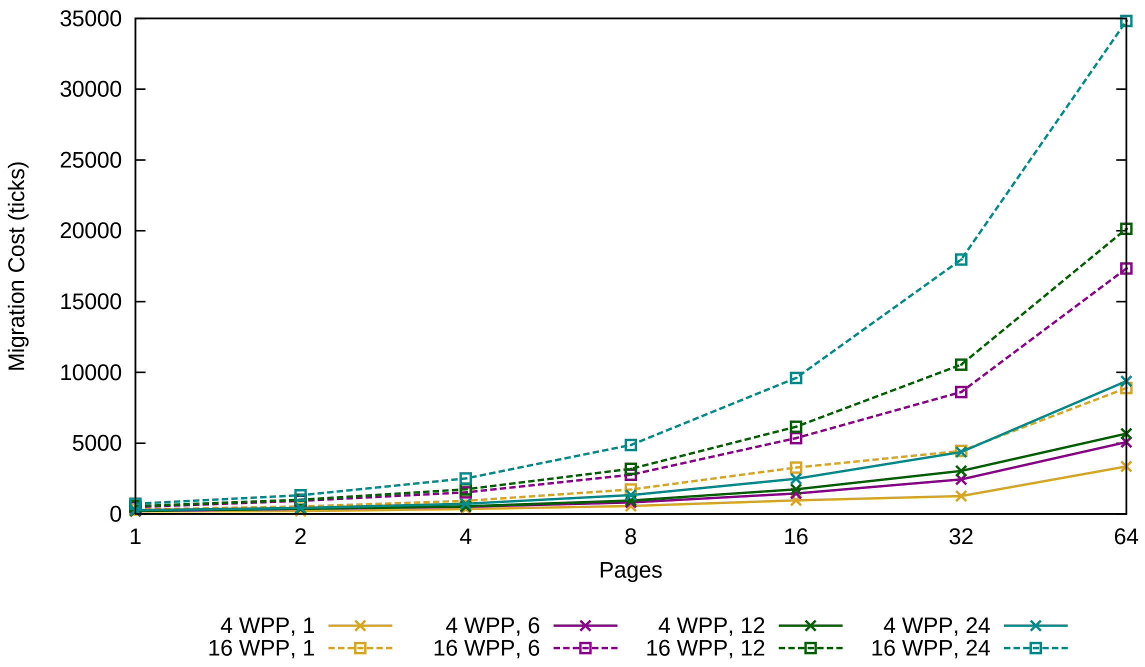


Figure 9.12: Migration costs of four migration paths on the 48-core

From these results, we can clearly see several things. First, there is a measurable difference in the various migration paths. In fact, the cache-miss overhead when migrating a task between two physically distinct processors is around four times the cost to migrate within an L3 cluster on all platforms. 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 that, as expected, memory access times are significantly shorter when accessing the calling processor's memory. The one exception to this is that, on our 48-core platform, calling processor 1's memory from processor 2 or 4 only incurred around a 15% overhead, while accessing processor 1's memory from processor 3 doubles the cost. We ran several further tests, and found that this effect also happened when fetching between processors 2 and 4. This is likely due to the interconnect topology. Additionally, because of the interconnect topology, we found that since only half of the cores in our 48-core system are directly connected to each other (see Figure 3.3) access times increased for all two-hop paths relative to their corresponding one-hop paths.

Second, the migration overhead scales inversely with clock speed. In all of our platforms, the memory speed is independent of the processor's clock speed. Therefore, slowing down the clock speed does not effect the time required to fetch data from memory. Therefore, fewer cycles are spent waiting for the fetched values. This is especially apparent on our AMD platforms, which share memory speed and processor architecture, but have different clock speeds. At most points, the difference between measured times for identical cases is almost exactly proportional to the difference in clock speeds.

The second aspect of task migration which needs measured is the cost of performing the actual migration in the scheduler. Table 9.1 shows the average cost of migrations for each scheduling architecture on each platform. 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. These results were collected with G-NP-EDF and G-EDF.

Table 9.1: Migration cost for various architectures and platforms (ticks)

	Concurrent			Stop-The-World		
Platform	Average	St. Dev.	Worst	Average	St. Dev.	Worst
8-Core	2183	727	12672	4318	1808	12844
16-Core	3326	728	23238	7346	2477	13728
48-Core	4516	1423	48459	14866	8401	38867

The difference between the two architectures is mainly due to lock contention for the per-core `runqueue`. When a 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 in the scheduler, so the puller must often block. However, the number of migrations is generally small, 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, leading to a lower average time. However, it is also possible that several migrations are interfering with each other. Hence, the worst observed time is significantly higher.

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 9.13 shows a histogram plot of the context switch times on our three platforms. Both the 8- and 16-core platforms show two distinct peaks, one representing context switches to a local task, and one representing a context switch to a recently-migrated task. The 48-core platform shows three peaks, representing local migration and the two possible migration paths discussed above.

9.3 System Call and Mutex Overhead

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 segment, and therefore, the sum of their execution time represents the minimum possible segment length. Figures 9.14 a, c, and e show the overheads of various system calls, including `begin_rt_seg()` and `end_rt_seg()`. Both `gettid()` and `clock_getres()` are relatively short, and therefore both

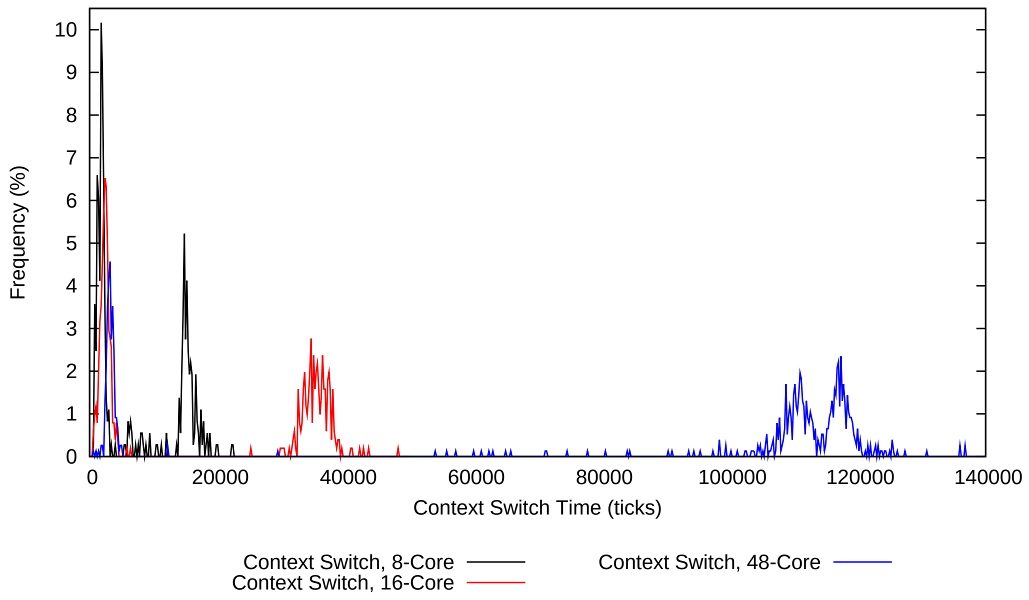


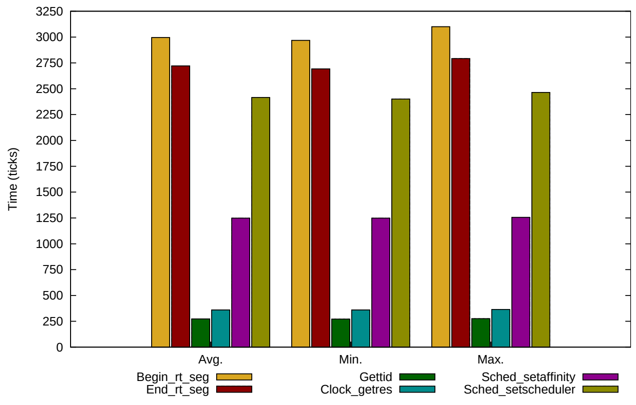
Figure 9.13: Context switch costs on our three platforms

of them provide reasonable 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. Taking these system call costs into account, we can improve our understanding of when the systems enter overload. By multiplying the average `begin_rt_seg()` and `end_rt_seg()` costs with no tasks in the system by the average arrival and departure frequencies, we generate the estimates shown in Table 9.2 for when our systems are fully loaded based on system call overhead.

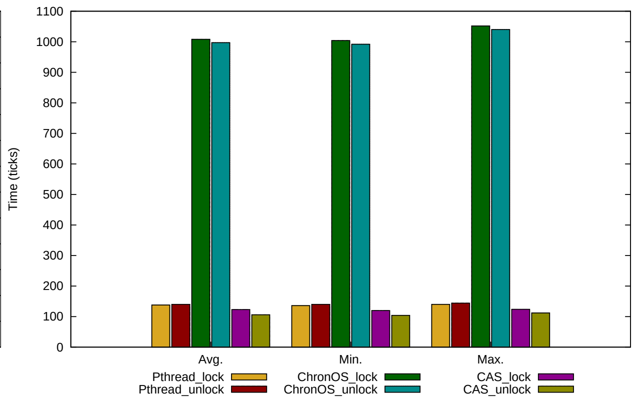
Table 9.2: Taskset load values at which the system is fully loaded based on system call overhead

Platform	BHB	BHU	BMB	BMU	BLB	BLU
8-Core	8.00	8.00	8.00	8.00	8.00	7.98
16-Core	16.00	16.00	15.99	15.99	15.99	15.96
48-Core	47.98	47.99	47.98	47.97	47.97	47.84

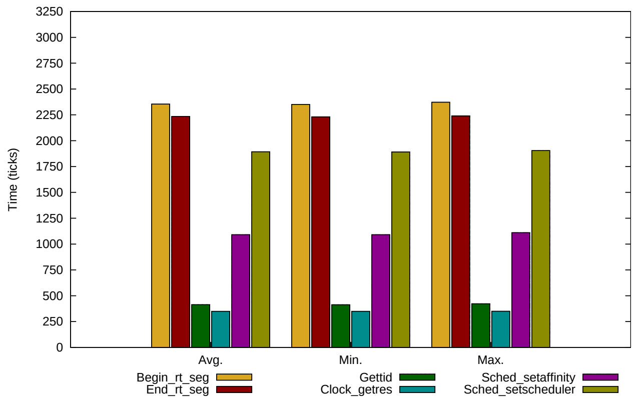
Another source of overhead exhibited in ChronOS is the cost of locking mutexes. While scheduler-managed mutexes are not exercised in any of the tests performed in this thesis, they are still an important feature, and therefore worth measuring. Figures 9.14 b, d, and f show the uncontested locking and unlocking costs of various types of locks, including the scheduler-managed locks provided in ChronOS. The overhead is significant, but not unexpected. The



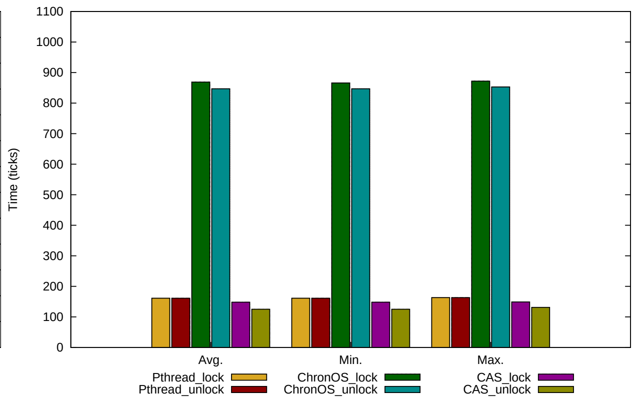
(a)



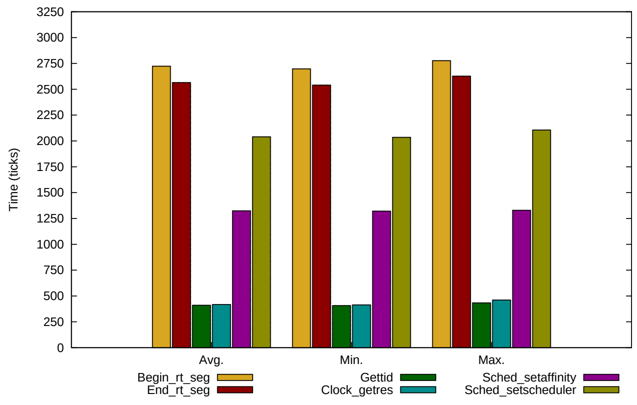
(b)



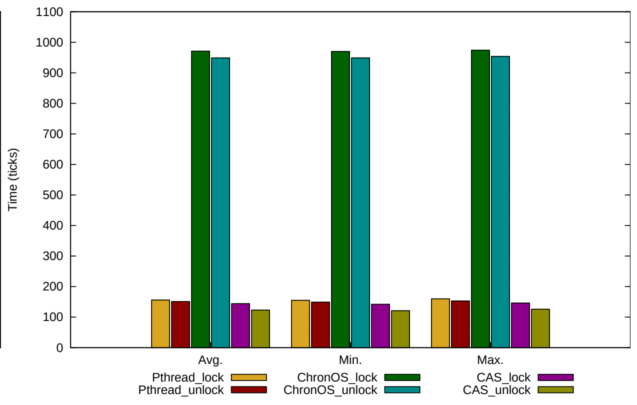
(c)



(d)



(e)



(f)

Figure 9.14: System call overheads for (a) 8-core (c) 16-core and (e) 48-core platforms and mutex locking and unlocking overheads for (b) 8-core (d)-16-core and (f) 48-core platforms

majority of the overhead is due to the fact that `schedule()` is called during every locking and unlocking call. This is potentially wasteful, since if the call is uncontested, this will most likely result in the calling task being scheduled, and therefore will not change the system schedule. However, it is necessary to provide correct scheduling behavior for algorithms like G-GUA and NG-GUA, which consider each resource request and release to be a scheduling event, even if the resource is uncontested.

9.4 Timing Accuracy

As mentioned in Chapter 3, there are two possible threading models, each of which can be implemented in Linux and rely on different timing mechanisms. We have implemented test programs to measure the accuracy of each model. The "thread-per-task" model was tested by creating a high priority thread which repetitively calls the Linux `usleep()` function and measures the actual time slept. The "thread-per-job" model was tested by creating a timer using `pthread_create()` with the `SIGEV_THREAD` set. The resulting thread reads the TSC and then terminates. Figures 9.15 and 9.16 shows the results for the two models. Since the units presented to the timing mechanisms are are units of time, our cycle measurements have been converted to units of time. Although standard deviations were calculated for all tests, they were omitted from the graphs because they never exceeded 0.6μs, and were therefore invisible on the graphs.

Clearly, both models provide sufficient accuracy above 1ms. The "thread-per-job" model appears to retain this accuracy down to around 100μs, after which point it becomes inaccurate. This is due to the overhead of thread creation and destruction. The "thread-per-task" model begins to lose accuracy slightly earlier, but is much more accurate in the in the 1-100μs range.

To further explore the accuracy, we measured each model on our 16-core platform under four different circumstances; the system was tested in an unloaded state and in a fully loaded state as described in Chapter 4. Additionally, we tested with the measuring thread locked to the lowest and highest cores in the system. The results of these tests are shown in Figures 9.17 and 9.18.

These tests clearly shown that the previous test results are pessimistic for the "thread-per-task" model; under fully loaded conditions the accuracy of the "thread-per-task" model improves significantly. As was the case with our tests in 4, this is because of the overhead of a core becoming idle and then returning. The "thread-per-job" model remains consistent across CPUs and load conditions. The degree of accuracy for both models is within the known range for the Linux timing infrastructure [96].

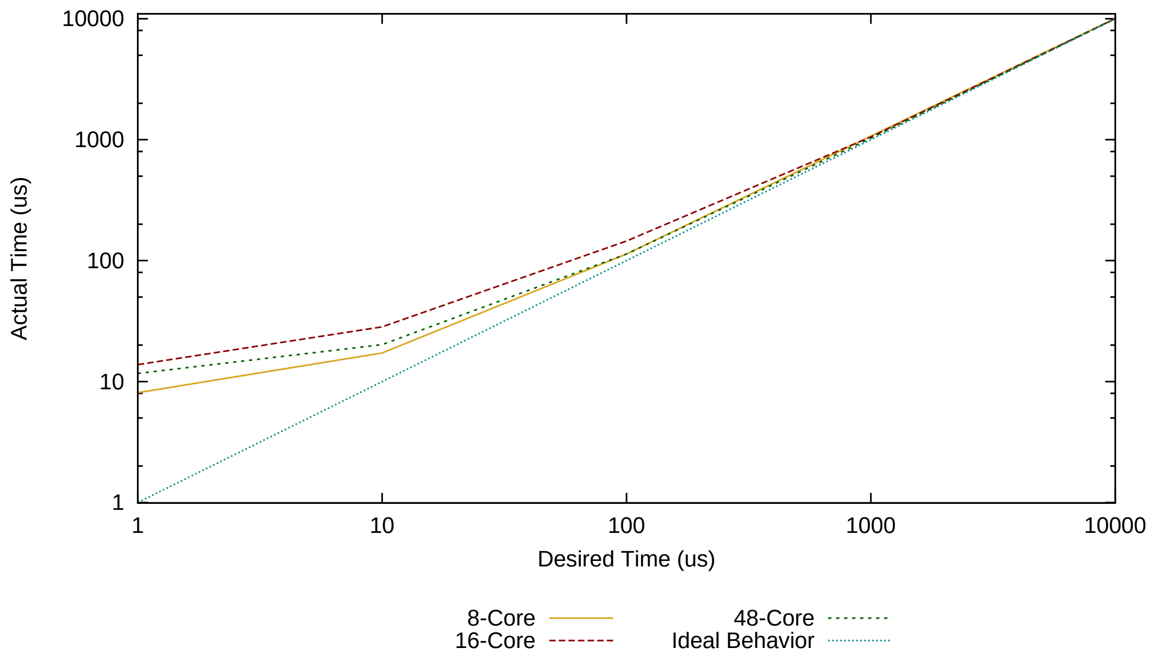


Figure 9.15: “Thread-per-task” accuracy on a variety of platforms

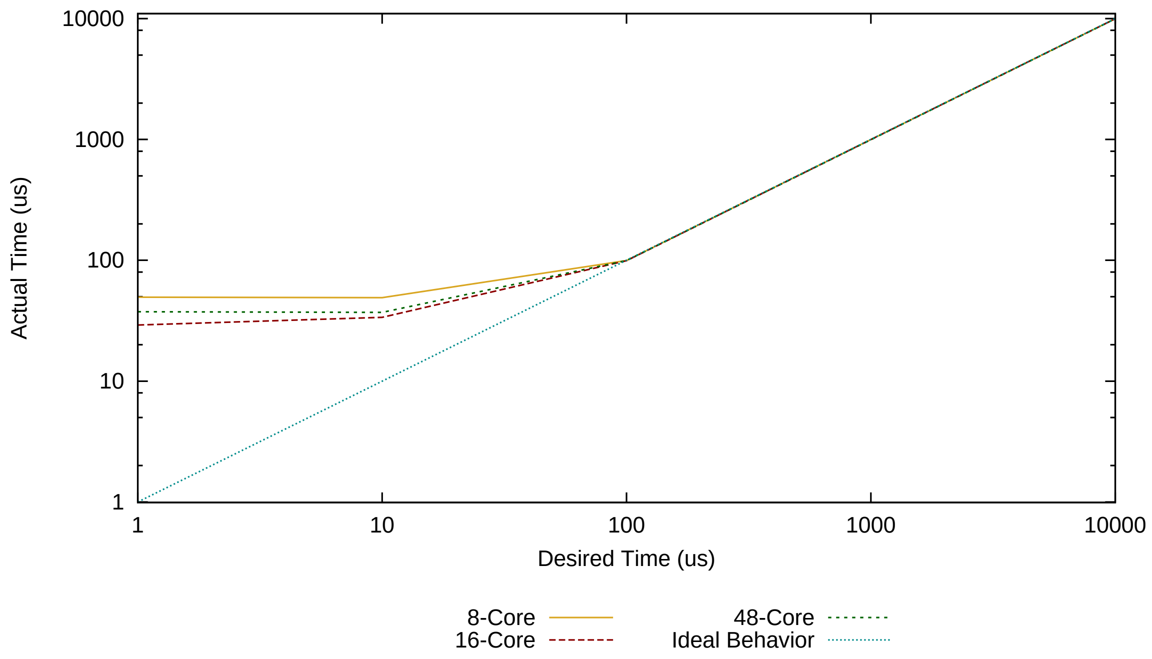


Figure 9.16: “Thread-per-job” accuracy on a variety of platforms

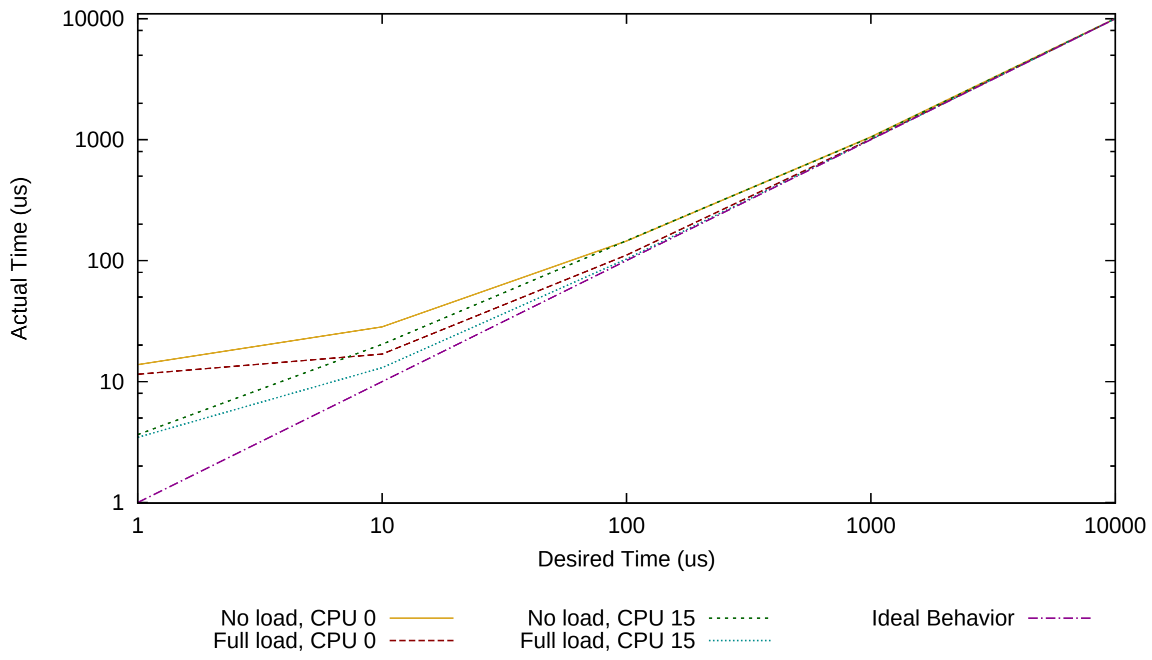


Figure 9.17: “Thread-per-task” accuracy under various conditions

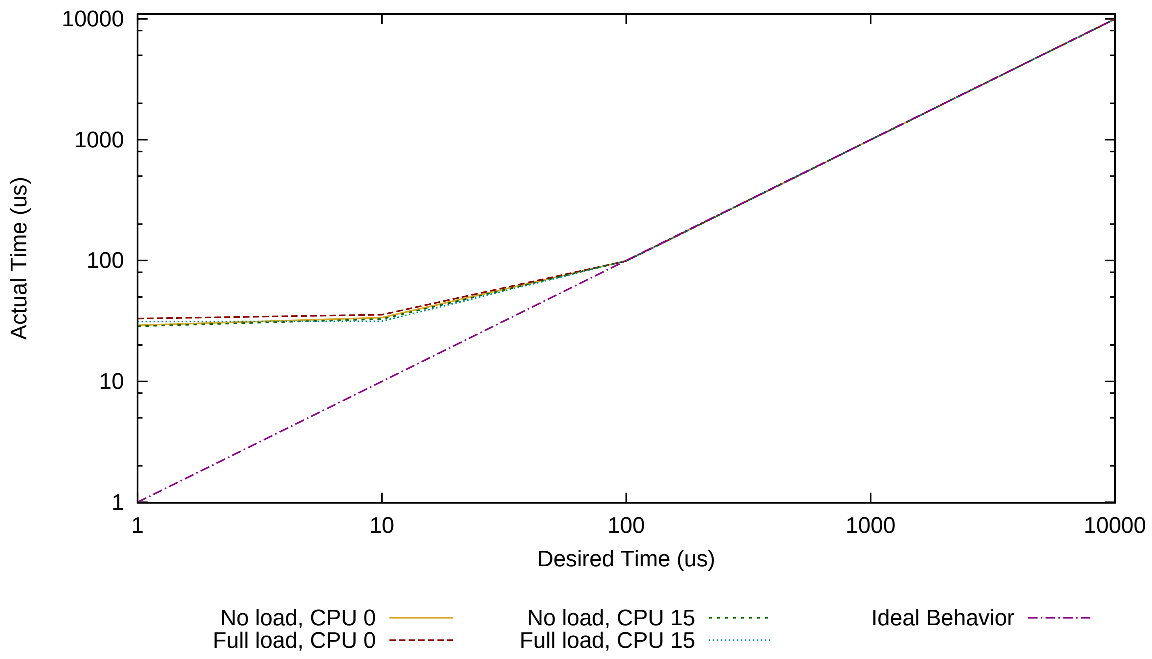


Figure 9.18: “Thread-per-job” accuracy under various conditions

Chapter 10

Conclusions

In this thesis, we experimentally evaluated the scalability of sixteen real-time scheduling algorithms on large-scale multicore platforms (e.g. 48-core AMD Opteron). These algorithms include global, clustered, and partitioned algorithms, and range from simplistic algorithms like G-FIFO to highly complex heuristic algorithms such as NG-GUA. Such an evaluation has not been done previously. Our 48-core platform is the highest core-count platform ever used to study global real-time scheduling.

Additionally, we presented the ChronOS Linux kernel, which is extended from the `PREEMPT_RT` real-time Linux patch. We present a set of modifications made to ChronOS to minimize lock contention, decrease lock blocking time, and optimize inter-processor synchronization.

Our experimentation consists of executing 288,000 tasksets divided among 6 different per-task weight distributions on the 16 schedulers. Our experimentation reveals that on our 48-core platform, all of our algorithms except G-GUA, NG-GUA, gMUA, and G-HVDF demonstrate performance consistent with their theoretical behavior for at least some tasksets. We see that three global algorithms (G-FIFO, G-NP-EDF, and G-NP-HVDF) and three partitioned algorithms (P-RMS, P-EDF, and P-HVDF) and C-EDF are quite close to their theoretical performance on the 48-core for all tasksets. Additionally, G-RMS, G-EDF, G-HVDF, P-LBESA, and P-DASA-ND are able to provide nearly correct performance for some of the six classes of tasksets. Furthermore, on our 48-core platform, our single-queue based G-FIFO implementation outperformed the Linux kernel's `SCHED_FIFO` implementation.

Based on this, we conclude that the conclusion reached by previous authors [4] that "global scheduling research should focus on modest processor counts (e.g. ≤ 8)" only applies to certain classes of algorithms. We find that on the 48-core, P-EDF and C-EDF are both able to provide high levels of schedulability, which is consistent with the results shown by Brandenburg et. al. However, unlike Brandenburg et. al. [31] we conclude that in our implementation, scalability is restricted by high scheduling overheads for some algorithms, cache-miss overheads, contention over the global scheduling lock and the cost of inter-processor communication, rather than the implementation of the global queue. Furthermore, we note that G-NP-EDF is able to provide high levels of scalability on our 48-core system.

Table 10.1 shows the asymptotic cost of each algorithm implemented and the performance of each algorithm on our three platforms. Each algorithm's performance is listed as "None", meaning it failed to provide theoretically correct performance for any of the six taskset distributions, "Some", implying correct behavior for at least one of the six, or "All", implying correct or nearly correct behavior under all conditions. Based on this, we conclude that in our implementation, there is a direct relationship between the complexity of an algorithm and its ability to scale to large-scale multicore platforms.

Table 10.1: Summarized results for all algorithms

Algorithm	Asymptotic Cost	8-Core	16-Core	48-Core
G-FIFO	$O(1)$	All	All	All
G-NP-EDF	$O(1)$	All	All	All
G-RMS	$O(m)$	All	All	Some
G-EDF	$O(m)$	All	All	Some
C-EDF	$O(m)$	All	All	All
G-NP-HVDF	$O(1)$	All	All	All
G-HVDF	$O(n)$	All	Some	None
gMUA	$O(mn^2)$	All	Some	None
NG-GUA	$O(mn^2)$	All	Some	None
G-GUA	$O(mn^2)$	All	Some	None
P-RMS	$O(1)$	All	All	All
P-EDF	$O(1)$	All	All	All
P-HVDF	$O(n)$	All	All	All
P-LBESA	$O(n^2)$	All	All	Some
P-DASA-ND	$O(n^2)$	All	All	Some

To refine these conclusions, we note that for all six taskset distributions:

- G-NP-EDF and G-FIFO are able to bound tardiness under nearly all loads on all three platforms ChronOS.
- G-EDF and G-RMS are able to meet their theoretically computed schedulability bounds for all distributions on the 8 and 16-core platforms and for some distributions on the 48-core platform.
- P-EDF and P-RMS are able to meet their theoretically computed schedulability bounds for all three platforms.

- For all cases, C-EDF provides full schedulability under a higher load than any global algorithm, except G-NP-EDF in the BMU and BLU cases on the 16-core platform and the BMU case on the 48-core.

Our research further demonstrates that the complexity and scheduling model of an algorithm directly affect its scalability. In theory, NG-GUA and G-EDF perform identically, however on our 48-core platform, we observed the following:

- G-EDF always provides full schedulability up to a load between 77.7% and 228% higher than NG-GUA on our 48-core platform.
- G-EDF achieves up to 3004% higher deadline satisfaction ratio on the 48-core platform, and up to 1035% higher deadline satisfaction ratio on the 16-core platform.
- G-EDF accrues up to 3664% more utility than NG-GUA on the 48-core platform, and up to 1195% more utility on the 16-core platform.

These differences are caused by the differences in scheduling overheads and by optimizations we are able to make to G-EDF which cannot be made to NG-GUA because of its scheduling model.

Additionally, our research showed that a simple heuristic can significantly outperform theoretically better heuristics in high core-count systems. Although G-NP-HVDF provides no schedulability guarantees, and schedules without regard to deadlines or other typical timing constraints, we observe the following:

- At full load, G-NP-HVDF meets at least 66% more deadlines than any other global utility accrual scheduler for all six distributions.
- At full load, G-NP-HVDF accrues at least 57% more deadlines than any other global utility accrual algorithm for all six distributions.

Based on these results, we conclude that scheduling algorithms implemented under the concurrent architecture are able to provide scalable global real-time scheduling up to 48 cores in ChronOS. In our experiments, these algorithms are G-FIFO, G-NP-EDF, and G-NP-HVDF. Furthermore, we conclude that it is possible to implement reasonably scalable global scheduling algorithms under the stop-the-world architecture, as is evidenced by the performance of G-RMS and G-EDF. Lastly, we conclude that clustered and partitioned algorithms also provide highly scalable solutions, but do so with a reduction in application flexibility.

We also present a set of modifications made to ChronOS to minimize lock contention, decrease lock blocking time, and optimize inter-processor synchronization. These modifications are experimentally tested and shown to increase the performance of stop-the-world architecture scheduling algorithms in ChronOS by up to 263% on 48-core platform.

Chapter 11

Future Work

11.1 Distributed Scheduling

As demonstrated by the Linux `SCHED_FIFO` scheduler and our concurrent architecture, a distributed approach to scheduling scales significantly better than a scheduling model which creates a schedule for the entire system. However, as demonstrated by `SCHED_FIFO`, this approach can require much more complex mechanisms to provide correct performance. Despite this difficulty, a distributed approach to scheduling appears the best model for scalable global scheduling. Future work must therefore focus on producing distributed algorithms which also provide theoretical bounds. While this has proven quite difficult in the hard real-time space, almost no effort has been given to the design of such algorithms in the utility accrual scheduling space. It is likely that relatively simple algorithms could be designed which would fit in our concurrent scheduling approach and also provide tighter bounds than G-NP-HVDF.

11.2 Parallel Scheduling

Previous work had suggested that it would be beneficial to create parallel scheduling algorithms, especially for complex algorithms such as G-GUA and NG-GUA [52]. While this approach is valid in theory, it is difficult to implement in practice. IPIs are not received instantaneously, cannot always be sent, and even when an IPI is sent, its reception cannot be guaranteed. This means that we cannot count on a given core performing part of the schedule and cannot guarantee that if performed it will be performed in a timely manner. Based on our research, we therefore conclude that unless a better mechanism for inter-processor synchronization can be found, parallel scheduling does not appear to be a feasible way of improving the schedulability of global scheduling algorithms.

11.3 Simpler Heuristics

As demonstrated by G-HVDF and G-NP-EDF, simpler heuristics can significantly outperform more complex heuristics on high core-count systems. There are a large number of other possible simple heuristics, such as fully abortive G-NP-HVDF, and global highest static value density first (G-HSVDF), which could be implemented and tested, and may in fact yield higher performance than those shown here. Further research is needed to explore such simple heuristics.

11.4 Reducing Contention and Improving Performance

There are a number of ways performance within ChronOS could be improved. First, it would likely significantly reduce migration distance to implement a distance-optimal mapping algorithm. This would in turn reduce bus contention after a scheduling event and increase scalability. However, this would not reduce the $m - 1$ migrations in the worst case for some classes of algorithms. This could be improved with cache-aware scheduling algorithms, or by redesigning existing algorithms to use job-static priorities. Such a redesign would also remove the need to calculate a job's priority at every scheduling event, reducing overhead for algorithms like G-HVDF, G-GUA, and NG-GUA.

Second, the global queue could be stored as a more advanced data structure, such as a heap. A binomial heap was shown to be an effective data structure for storing the global task list in [30]. Even if the global queue is stored as a linked-list, there are likely a range of improvements that can be made to improve the average and worst-case insertion times.

11.5 Additional Workloads

This thesis has only explored CPU-intensive workloads composed of independent tasks. While this is a large subset of real-time workloads, further understanding would be gained into the scalability of these algorithms by exploring memory-intensive workloads and deponent tasks. There exist a wide range of scientific calculations which commonly occur in real-time systems and exhibit both of these behaviors when executed in a multi-threaded manner. Three example workloads are multi-threaded matrix multiplication, multi-threaded FFT computation, and multi-threaded sorting of a large dataset.

11.6 Improved interrupt Handling

In this thesis, we scoped out experimenting with a variety of interrupt handling patterns. However, given that the `PREEMPT_RT` places interrupts in Linux kernel threads, a logical extension would be to make the priorities of these interrupts user space manageable. This would allow the user to selectively chose which interrupts it needs and receive interference from only those interrupts by managing both application and interrupt handling priorities. Furthermore, based on the work of Brandenburg et. al. [30], it is worth further investigating the binding of interrupt handling to specific processors.

Bibliography

[1] AMD. High Performance Computing with Amd, March 2010. `http://sites.amd.com/us/business/it-solutions/compute-intensive-hpc/Pages/compute-intensive-hpc.aspx`.

[2] AMD. Magny-Cours and Direct Connect Architecture 2.0, March 2010. `http://developer.amd.com/documentation/articles/pages/magny-cours-direct-connect-architecture-2.0.aspx`.

[3] James Anderson, Vasile Bud, and UmaMaheswari Devi. An edf-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Systems*, 38:85–131, 2008. 10.1007/s11241-007-9035-0.

[4] James H. Anderson. Real-time multiprocessor scheduling: Connecting theory and practice, November 2010. `www.cs.unc.edu/~anderson/litmus-rt/slides/rtns2010-keynote.pptx`.

[5] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0:193, 2001.

[6] Bjrn Andersson and Lus Miguel Pinho. Implementing multicore real-time scheduling algorithms based on task splitting using ada 2012. In *Ada-Europe*, pages 54–67, 2010.

[7] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[8] Anderson Bailey. Going to barcelona: A modern architecture for breakthrough software performance, September 2007. `http://developer.amd.com/documentation/articles/pages/972007175_4.aspx`.

[9] T. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors., 2005. Technical Report TR-051101, Florida State University.

[10] Theodore P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 120–, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Theodore P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. Technical report, Florida State University, 2005.

[12] Theodore P. Baker. Further improved schedulability analysis of edf on multiprocessor platforms. Technical report, Florida State University, 2005.

[13] T.P. Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760–768, aug. 2005.

[14] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application. *Nuclear Science, IEEE Transactions on*, 55(1):435 –439, feb. 2008.

[15] Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 321–329, 2005.

[16] Sanjoy K. Baruah. The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. *Real-Time Syst.*, 32(1-2):9–20, 2006.

[17] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15(6):600–625, 1996.

[18] Sanjoy K. Baruah and Joel Goossens. Rate-monotonic scheduling on uniform multiprocessors. *Distributed Computing Systems, International Conference on*, 0:360, 2003.

[19] Sanjoy K. Baruah and Jayant R. Haritsa. Scheduling for overload in real-time systems. *IEEE Trans. Comput.*, 46(9):1034–1039, 1997.

[20] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. *Real-Time Systems Symposium, IEEE International*, 0:14–24, 2010.

[21] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Is semi-partitioned scheduling practical? In *ECRTS '11: Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2011. to appear.

[22] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the*

ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[23] Alessio Bechini and Cosimo Antonio Prete. Performance-steered design of software architectures for embedded multicore systems. *Softw. Pract. Exper.*, 32(12):1155–1173, 2002.

[24] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved Schedulability Analysis of EDF on Multiprocessor Platforms. In *ECRTS '05*, pages 209–218, 2005.

[25] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 20:553–566, April 2009.

[26] Aaron Block, Björn Brandenburg, James H. Anderson, and Stephen Quint. An Adaptive Framework for Multiprocessor Real-Time System. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 23–33, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Aaron Block, Hennadiy Leontyev, Bjorn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[29] B Brandenburg, A Block, J Calandrino, U Devi, H Leon-tyev, and J Anderson. LITMUS^{RT}: A Status Report. In *RTLWS '07*, 2007.

[30] Björn B. Brandenburg and James H. Anderson. On the implementation of global real-time schedulers. In *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pages 214–224, Washington, DC, USA, 2009. IEEE Computer Society.

[31] Bjorn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *RTSS '08*, pages 157–169, Washington, DC, USA, 2008. IEEE Computer Society.

[32] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. *Real-Time Systems, Euromicro Conference on*, 0:247–258, 2007.

[33] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmus$\hat{r}$t : A testbed for empirically comparing real-time multiprocessor schedulers. *Real-Time Systems Symposium, IEEE International*, 0:111–126, 2006.

[34] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models.* Chapman Hall/CRC, Boca, 2004.

[35] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Softw. Eng.*, 15:1261–1269, October 1989.

[36] Hyeonjoong Cho. *Utility Accrual Real-Time Scheduling and Synchronization on Single and Multiprocessors: Models, Algorithms, and Tradeoffs.* PhD thesis, Virginia Tech, August 2006.

[37] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110, Washington, DC, USA, 2006. IEEE Computer Society.

[38] R. Clark, E. Jensen, and F. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, 1993.

[39] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, CMU, 1990. CMU-CS-90-155.

[40] Raymond Clark, E. Douglas Jensen, Arkady Kanevsky, John Maurer, Paul Wallace, Thomas Wheeler, Yun Zhang, Douglas Wells, Tom Lawrence, and Pat Hurley. An Adaptive, Distributed Airborne Tracking System ("Process the Right Tracks at the Right Time"). In *In IEEE WPDRTS, volume 1586 of LNCS*, pages 353–362. Springer-Verlag, 1999.

[41] Intel Corporation. Using the rdtsc instruction for performance monitoring, 1997.

[42] Edward Curley. Recovering from Distributable Thread Failures with Assured Timeliness in Real-Time distributed Systems. Master's thesis, Virginia Tech, February 2007.

[43] Arnaldo Carvalho de Melo. signaltest: Using the rt priorities, May 2007. `rt.et.redhat.com/wiki/images/8/8e/Rtprio.pdf`.

[44] Umamaheswari C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Syst.*, 38:133–189, February 2008.

[45] UmaMaheswari C. Devi and James H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 330–341, Washington, DC, USA, 2005. IEEE Computer Society.

[46] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operations Research, 26(1)*, pages 127–140, 1978.

[47] Peter Dibble. *Real-Time Java Platform Programming.* BookSurge Publishing, June 2008.

[48] Jeremy Erickson, UmaMaheswari Devi, and Sanjoy Baruah. Improved tardiness bounds for global edf. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 14–23, Washington, DC, USA, 2010. IEEE Computer Society.

[49] S. Fahmy. *Collaborative Scheduling and Synchronization of Distributable Real-Time Threads.* PhD thesis, Virginia Tech, May 2010.

[50] Free Software Foundation. Gnu general public license. `http://www.gnu.org/licenses/gpl.html`.

[51] Ankita Garg. Real-time linux kernel scheduler, August 2009. `http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler`.

[52] Piyush Garyali. On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors. Master's thesis, Virginia Tech, Jul 2010.

[53] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25:187–205, September 2003.

[54] Linley Gwennap. Two-headed snapdragon takes flight. *Microprocessor Report*, 323:1–6, July 2010.

[55] Ravi Hegde. Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers, October 2008. `http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers/`.

[56] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[57] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, July 2008.

[58] Philip L. Holman. *On the implementation of pfair-scheduled multiprocessor systems.* PhD thesis, 2004.

[59] Paul Hyde. *Java Thread Programming.* Sams Publishing, 2001.

[60] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology – Portable Operating System Interface (POSIX) System Interfaces, Issue 6. 2001.* Open Group Technical Standard Base Specifications, Issue 6, 1992.

[61] Apple Inc. Grand Central Dispatch: A better way to do multicore. Technical report, August 2009.

[62] Lineo Inc. DIAPM RTAI Programming Guide 1.0. Technical report, September 2000.

[63] Damir Isović and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings of the 21st IEEE conference on Real-time systems symposium*, RTSS'10, pages 207–216, Washington, DC, USA, 2000. IEEE Computer Society.

[64] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th IEEE International Real-Time Systems Symposium*, RTSS '91, pages 129–139, San Antonio, Texas, December 1991.

[65] E. Jensen, C. Locke, and H. Tokuda. A Time Driven Scheduling Model for Real-Time Operating Systems, 1985. IEEE RTSS, pages 112–122, 1985.

[66] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance Implications of Cache Affinity on Multicore Processors. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 151–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[67] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer, second edition, 2011.

[68] G. Koren and D. Shasha. D-OVER; An Optimal On-line Scheduling Algorithm for Overloaded Real-Time Systems. In *Real-Time Systems Symposium, 1992*, pages 290–299, 2-4 1992.

[69] Gilad Koren and Dennis Shasha. MOCA: A Multiprocessor On-line Competitive Algorithm for Real-Time System Scheduling. *Theor. Comput. Sci.*, 128(1-2):75–97, 1994.

[70] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 239–248, 1-3 2009.

[71] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 239–248, Washington, DC, USA, 2009. IEEE Computer Society.

[72] J. P. Lehoczky, L. Sha, and Y. Ding. Rate-Monotonic Scheduling Algorithm: Exact characterization and average case behavior. In *Proc. of the 11th IEEE Real-time Systems Symposium*, pages 166–171, December 1989.

[73] Hennadiy Leontyev and James H. Anderson. Tardiness bounds for fifo scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–, Washington, DC, USA, 2007. IEEE Computer Society.

[74] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms.* PhD thesis, Virginia Tech, 2004.

[75] Peng Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms.* PhD thesis, Virginia Tech, July 2004.

[76] Peng Li, Binoy Ravindran, and E. Douglas Jensen. Adaptive time-critical resource management using time/utility functions: Past, present, and future. In *COMPSAC '04*, pages 12–13, Washington, DC, USA, 2004. IEEE Computer Society.

[77] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Transactions on Software Engineering*, 30(9):613–629, 2004.

[78] Peng Li, Haisang Wu, Binoy Ravindran, and E. Douglas Jensen. A Utility Accrual Scheduling Algorithm for Real-Time Activities with Mutual Exclusion Resource Constraints. *IEEE Trans. Comput.*, 55(4):454–469, 2006.

[79] Ville Likitalo. Threads and Scheduling in Real-Time Specification for Java Framework. In *Proceedings of Seminar on Real-Time Programming*, pages 9–15, Espoo, Finland, 2004. Helsinki University of Technology.

[80] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[81] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, CMU, 1986. CMU-CS-86-134.

[82] J. M. López, M. García, J. L. Díaz, and D. F. García. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, pages 25–33, Washington, DC, USA, 2000. IEEE Computer Society.

[83] Jos M. Lpez, Jos L. Daz, and Daniel F. Garca. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. In *In 13th Euromicro Conference on RealTime Systems*, pages 67–75, 2001.

[84] Tim Mattson. The future of many core computing: A tale of two processors, March 2010. `og-hpc.com/Rice2010/Slides/Mattson-OG-HPC-2010-Intel.pdf`.

[85] Paul McKenney. A realtime preemption overview, August 2005. `http://lwn.net/Articles/146861/`.

[86] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[87] Sun Microsystems. Memory Management in the Java Hotspot Virtual Machine. Technical report, April 2006.

[88] Jonas Mitschang. Harte echtzeit unter linux fallstudie rtai vs. rt-preempt, March 2007. `mitschang.net/download/IESE-Report%2058.pdf`.

[89] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: a power/performance/thermal view. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 177–186, New York, NY, USA, 2006. ACM.

[90] Douglas Niehaus. Kusp: Kernel/user systems programming, 2010. `http://www.ittc.ku.edu/kurt/`.

[91] J. Duane Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.* Academic Press, 1987.

[92] National Institute of Standards and Technology. Introduction to linux for real-time control, 2002.

[93] D. I. Oh and T. P. Baker. Utilization bounds for n -processor rate monotone scheduling with stable processor assignment. *Real Time Systems*, 15(2):183–193, September 1998.

[94] OMG. Real-time CORBA Specification 1.2: : Dynamic Scheduling Specification. Technical report, Object Management Group, January 2005.

[95] Binoy Ravindran, Edward Curley, Jonathan S. Anderson, and E. Douglas Jensen. On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In *ISORC '07*, pages 344–353. IEEE Computer Society, 2007.

[96] Steve Rostedt and Darren V. Hart. Internals of the RT patch. In *Proceedings of the Linux Symposium*, volume 2, pages 161–172, 2007.

[97] Paulo Baltarejo Sousa. Implementing a multiprocessor linux scheduler for real-time sporadic tasks. In *Proceedings of the 4th Doctoral Symposium on Infomatics Engineering*, February 2009.

[98] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.*, 84:93–98, October 2002.

[99] John A. Stankovic and Krithi Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Softw.*, 8:62–72, May 1991.

[100] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 80–91, New York, NY, USA, 1992. ACM.

[101] A. Stoyenko and L. Georgiadis. On optimal lateness and tardiness scheduling in real-time systems. *Computing*, 47:215–234, 1992. 10.1007/BF02320193.

[102] Boleslaw K. Szymanski. Mutual exclusion revisited. In *Proceedings of the fifth Jerusalem conference on Information technology*, JCIT, pages 110–119, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[103] Teik Guan Tan and Wynne Hsu. Scheduling multimedia applications under overload and non-deterministic conditions. *IEEE RTSS*, 0:178, 1997.

[104] Ringlord Technologies. Posix Signal Handling in Java. Technical report, 2006.

[105] Inc. Tokyo Stock Exchange Group. Tse launches next-generation "arrowhead" trading system, January 2010. `http://www.fujitsu.com/global/news/pr/archives/month/2010/20100108-01.html`.

[106] Theodore Ts'o, Darren Hart, and John Kacur. Real-time linux wiki, 2011. `https://rt.wiki.kernel.org/`.

[107] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *RTSS '99*, page 246, 1999.

[108] L. R. Welch, B. Ravindran, B. A. Shirazi, and C. Bruggeman. Specification and modeling of dynamic, distributed real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 72–, Washington, DC, USA, 1998. IEEE Computer Society.

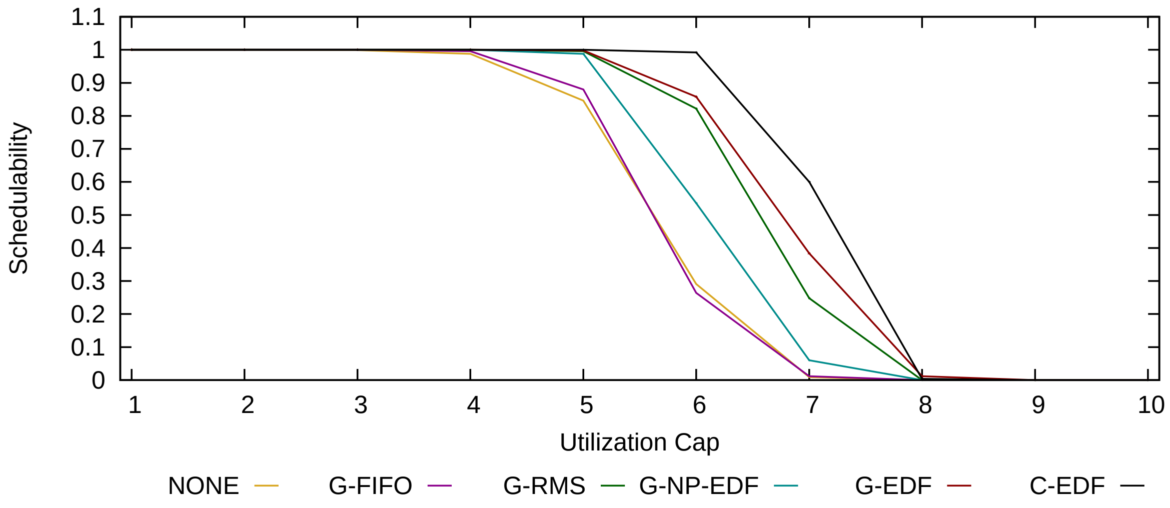
Appendix A

Complete Schedulability Results

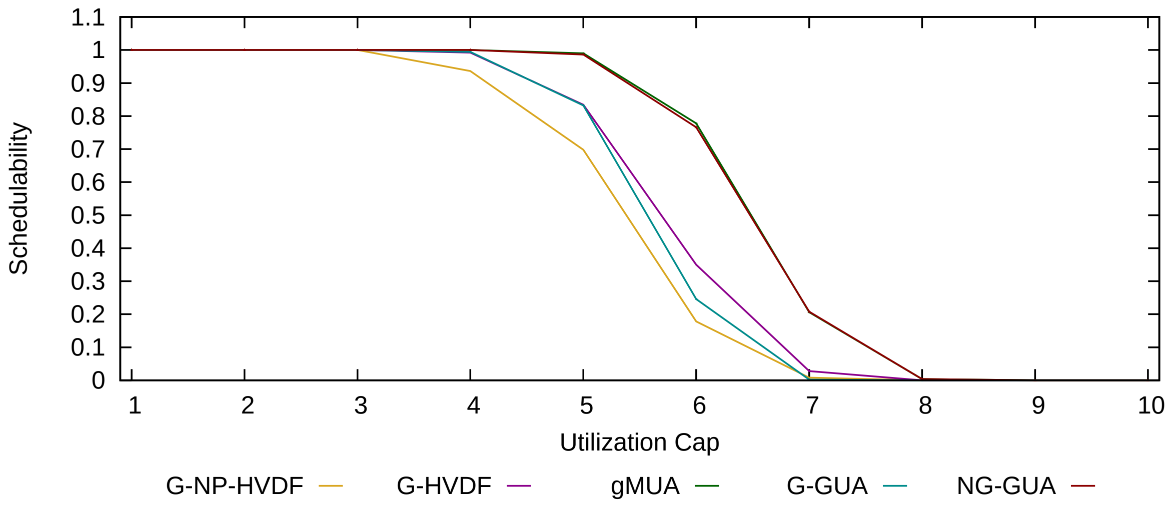
This appendix provides our complete schedulability results. It contains 36 plots grouped in 12 figures by taskset distribution and machine. It is organized as follows:

- Figure A.1 shows schedulability results for our 8-core platform under heavy bimodal load
- Figure A.2 shows schedulability results for our 8-core platform under heavy uniform load
- Figure A.3 shows schedulability results for our 8-core platform under medium bimodal load
- Figure A.4 shows schedulability results for our 8-core platform under medium uniform load
- Figure A.5 shows schedulability results for our 8-core platform under light bimodal load
- Figure A.6 shows schedulability results for our 8-core platform under light uniform load
- Figure A.7 shows schedulability results for our 16-core platform under heavy bimodal load
- Figure A.8 shows schedulability results for our 16-core platform under heavy uniform load
- Figure A.9 shows schedulability results for our 16-core platform under medium bimodal load

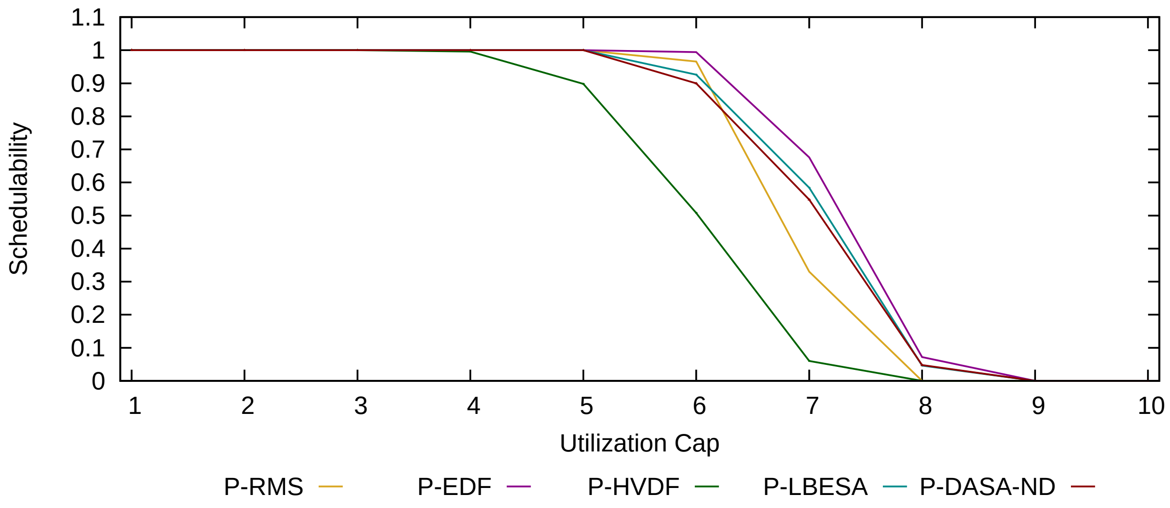
- Figure A.10 shows schedulability results for our 16-core platform under medium uniform load
- Figure A.11 shows schedulability results for our 16-core platform under light bimodal load
- Figure A.12 shows schedulability results for our 16-core platform under light uniform load
- Figure A.13 shows schedulability results for our 48-core platform under heavy bimodal load
- Figure A.14 shows schedulability results for our 48-core platform under heavy uniform load
- Figure A.15 shows schedulability results for our 48-core platform under medium bimodal load
- Figure A.16 shows schedulability results for our 48-core platform under medium uniform load
- Figure A.17 shows schedulability results for our 48-core platform under light bimodal load
- Figure A.18 shows schedulability results for our 48-core platform under light uniform load



(a)

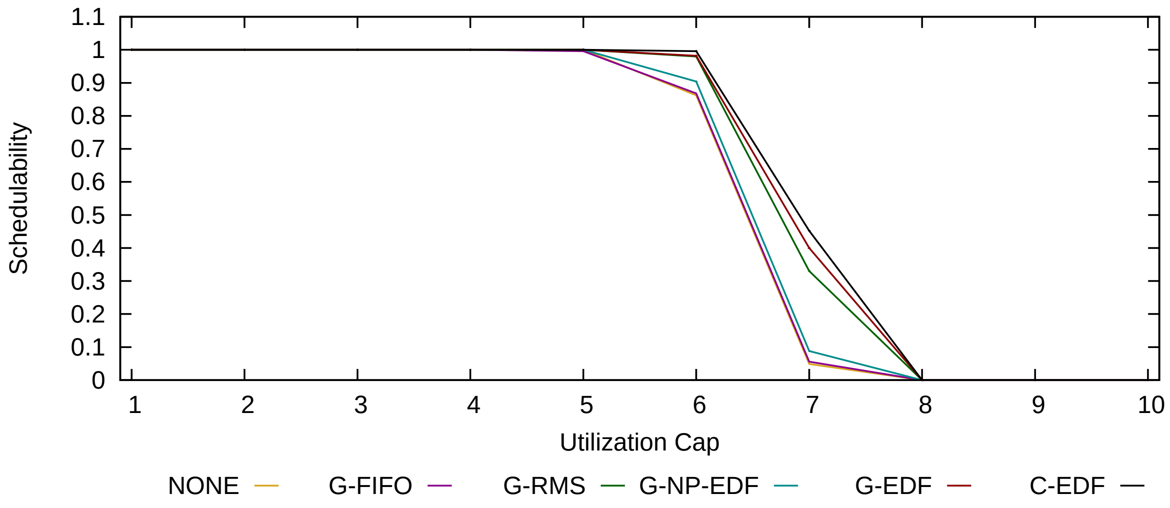


(b)

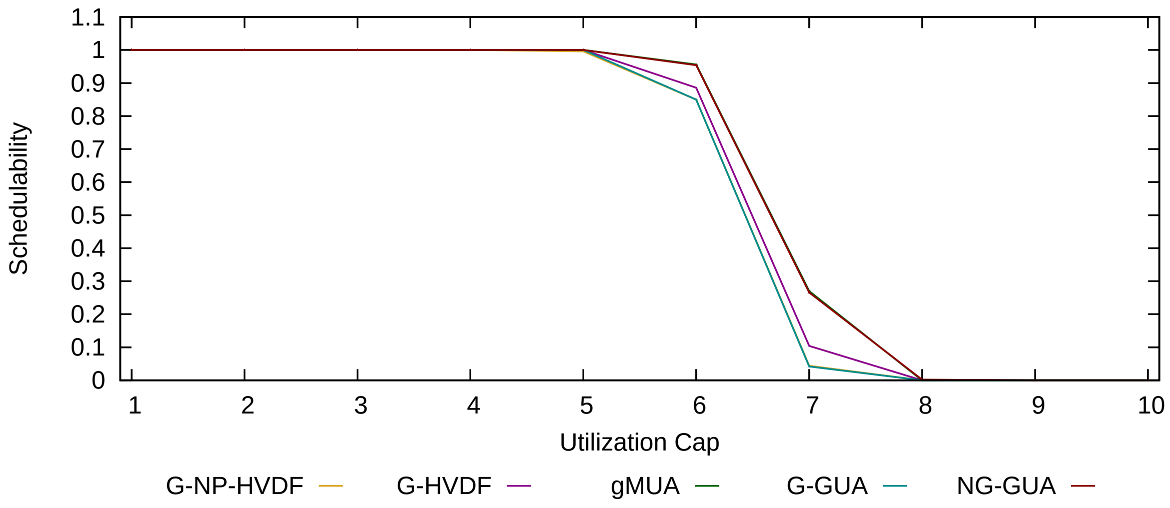


(c)

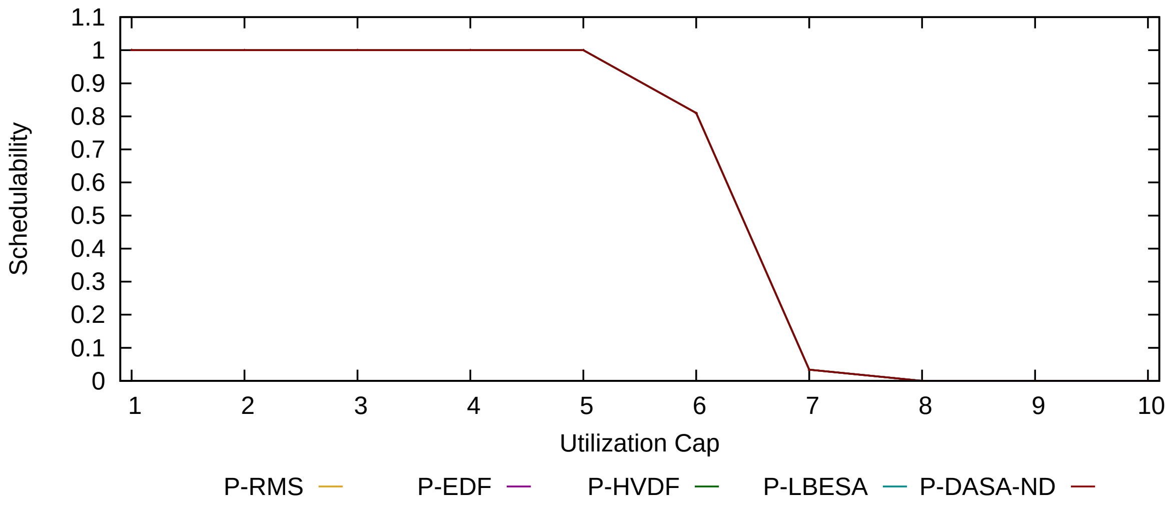
Figure A.1: 8-Core Schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

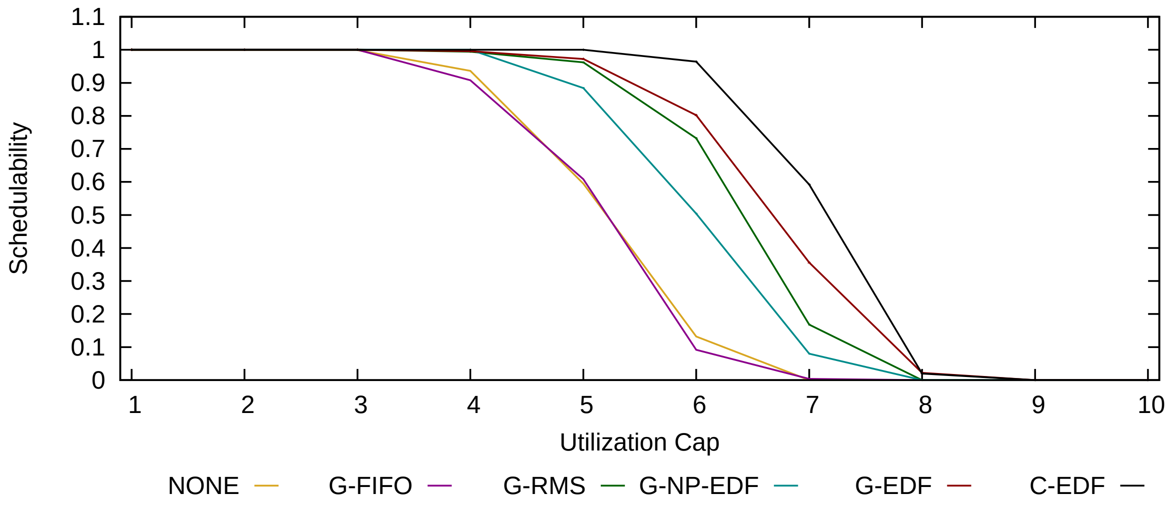


(b)

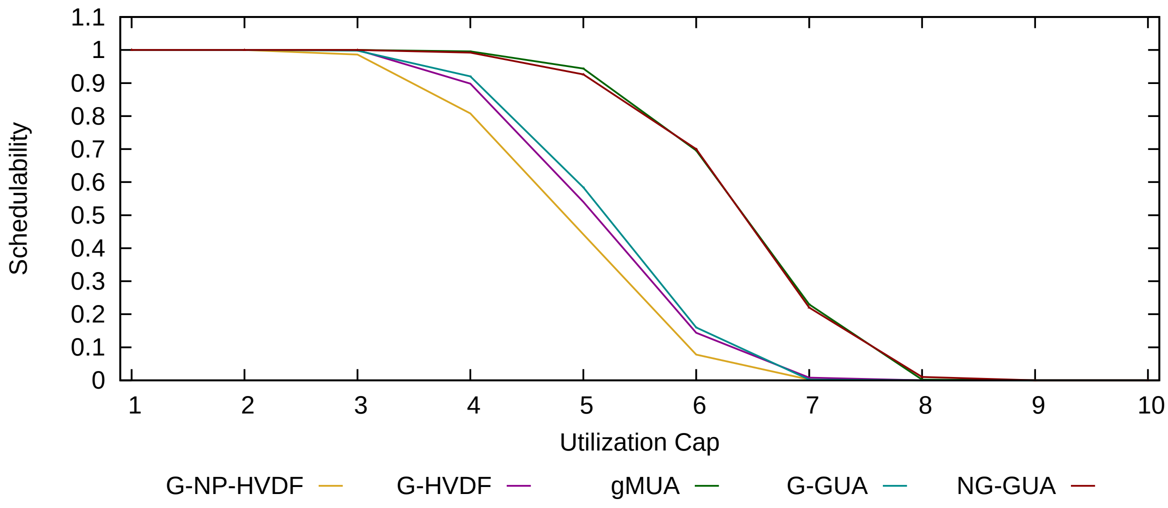


(c)

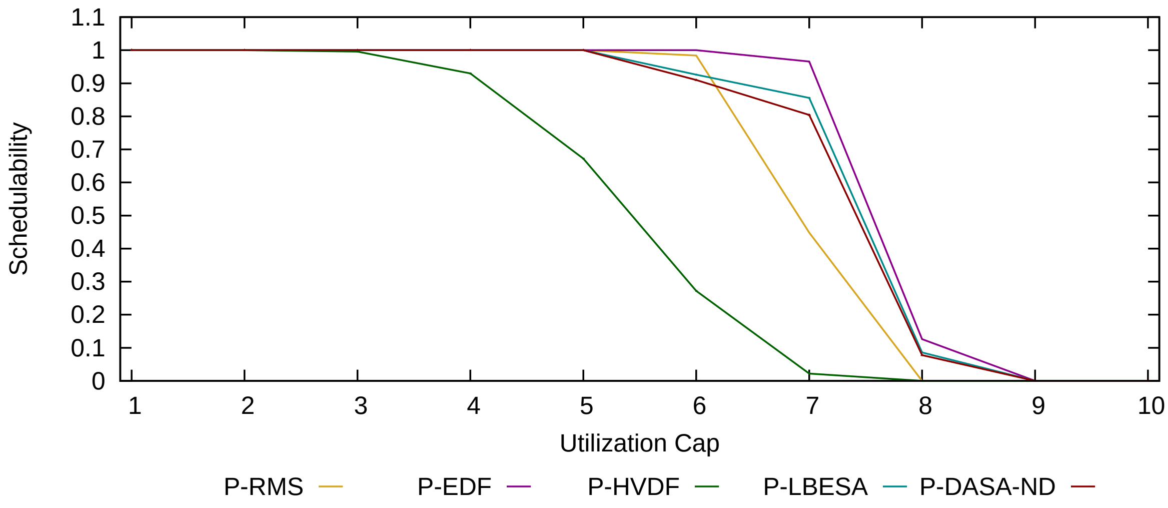
Figure A.2: 8-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

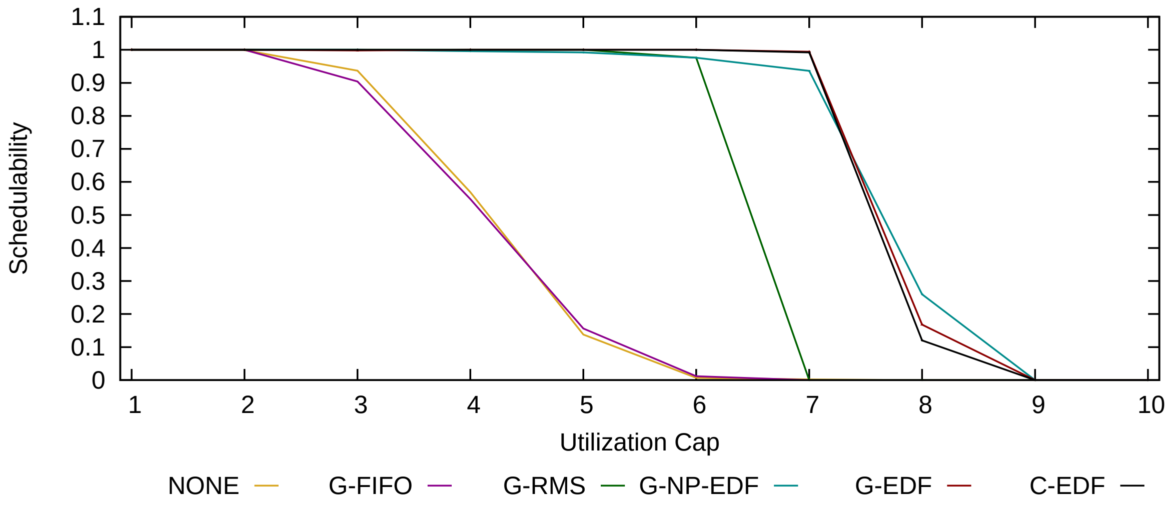


(b)

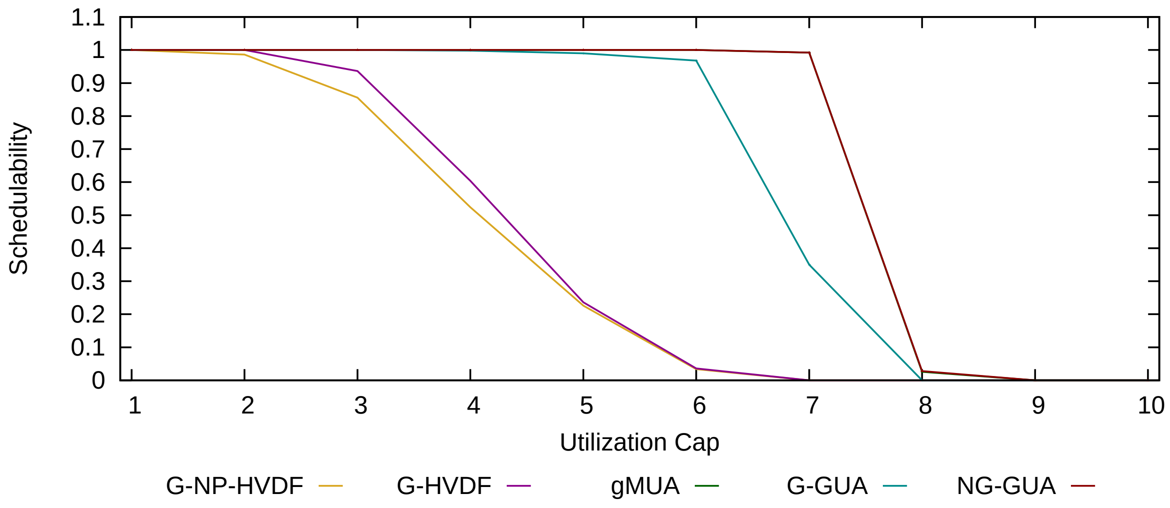


(c)

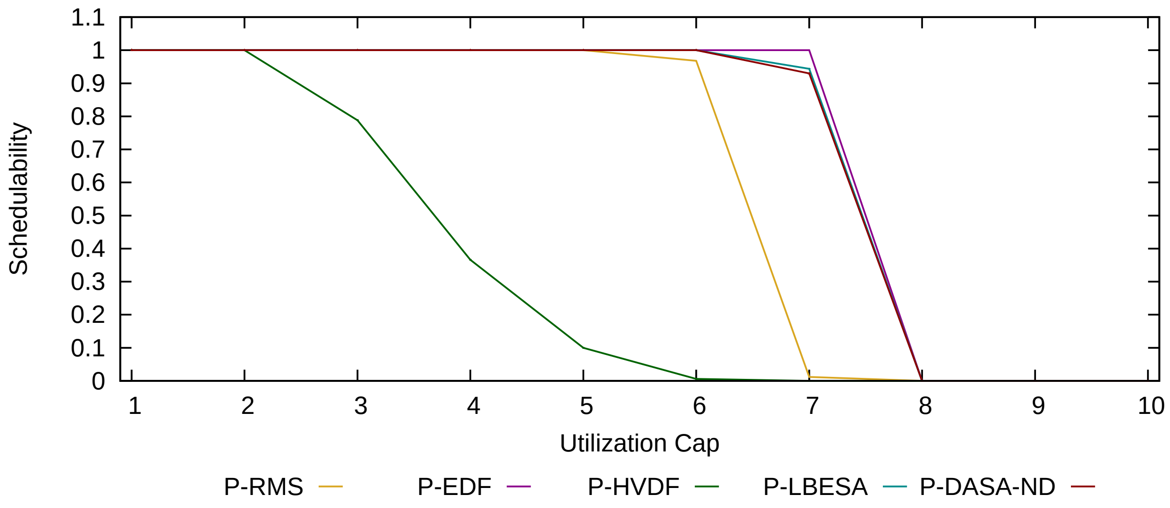
Figure A.3: 8-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

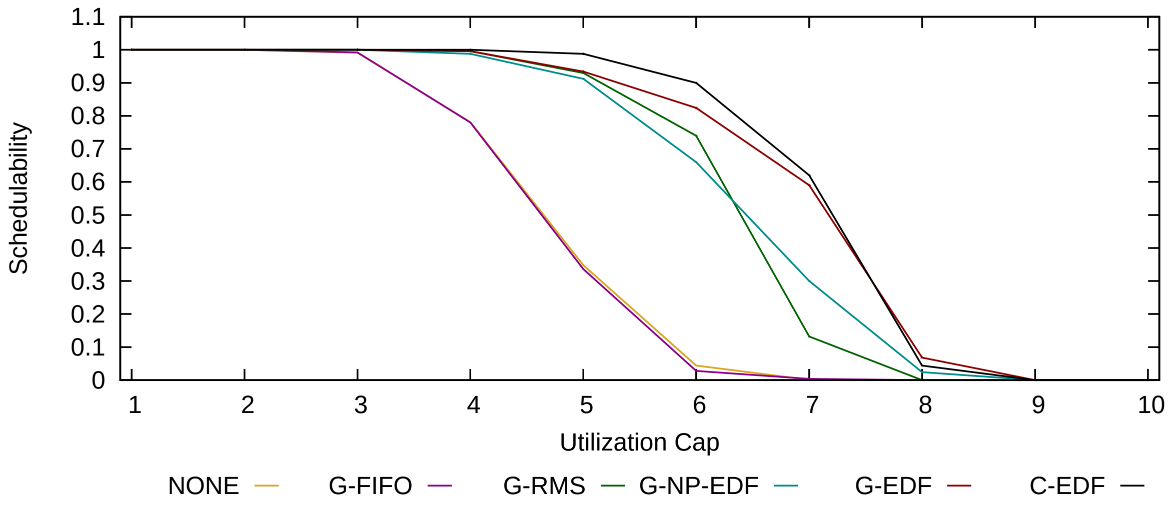


(b)

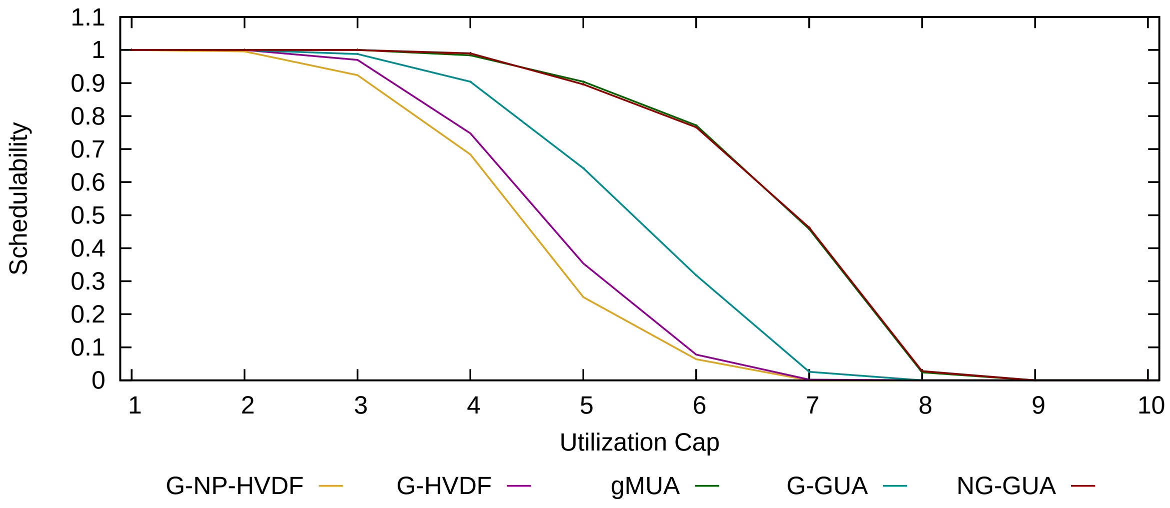


(c)

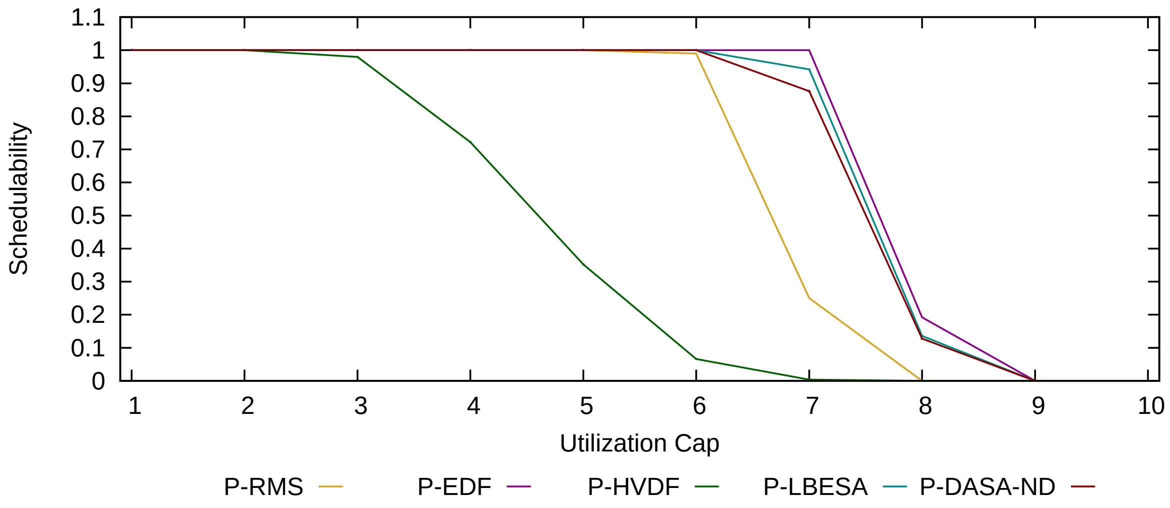
Figure A.4: 8-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

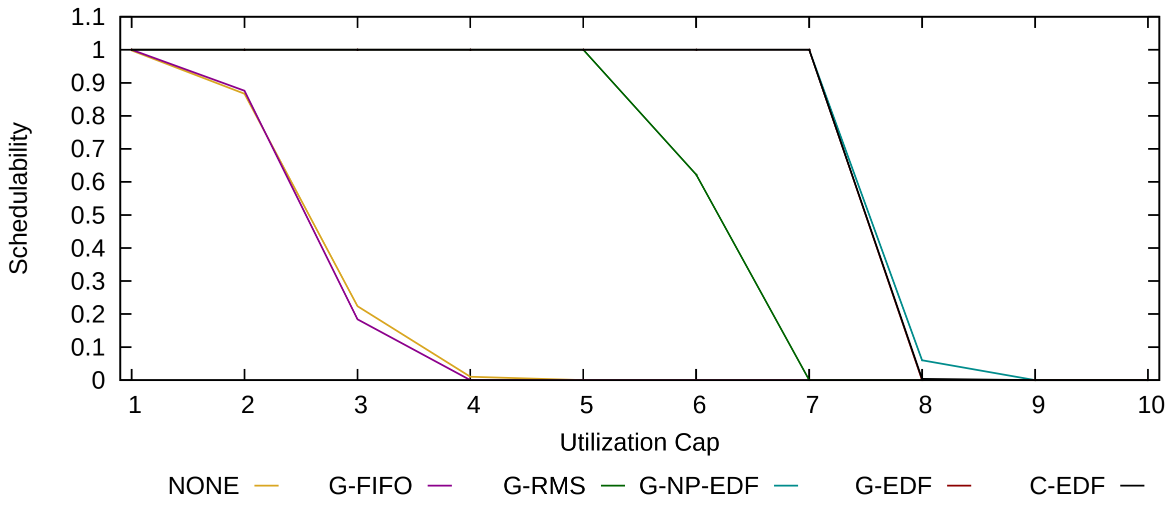


(b)

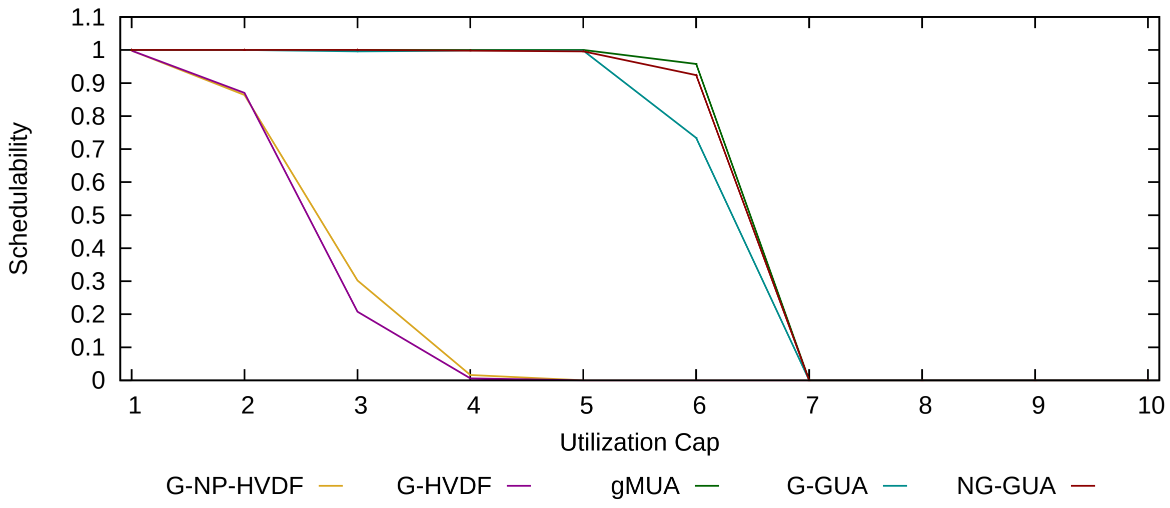


(c)

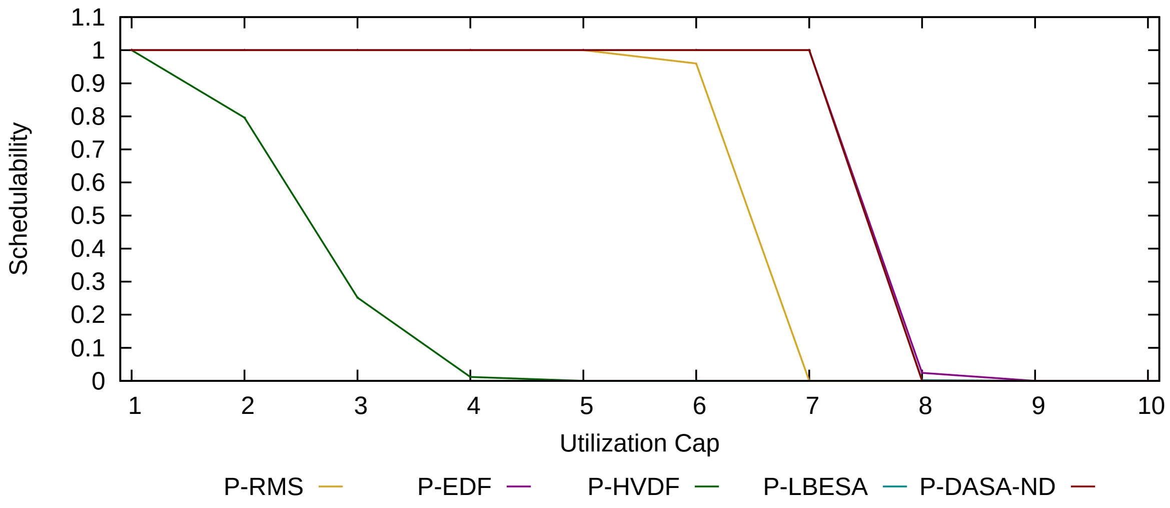
Figure A.5: 8-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

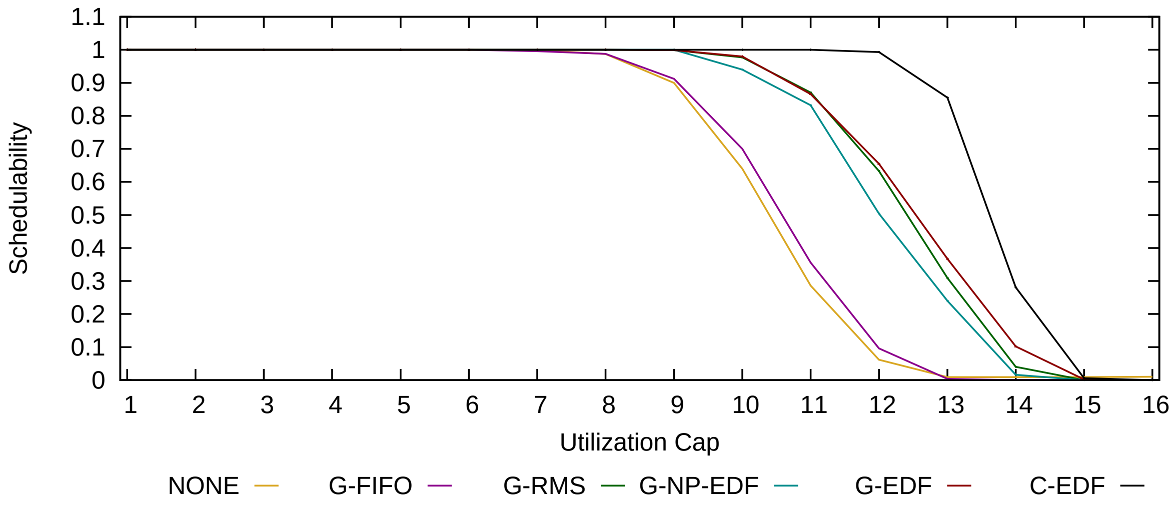


(b)

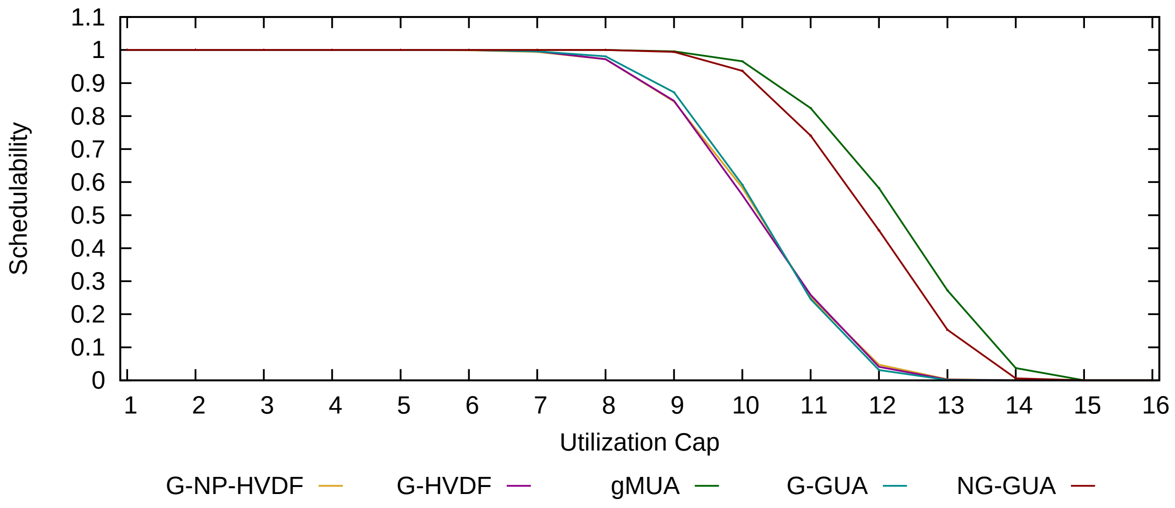


(c)

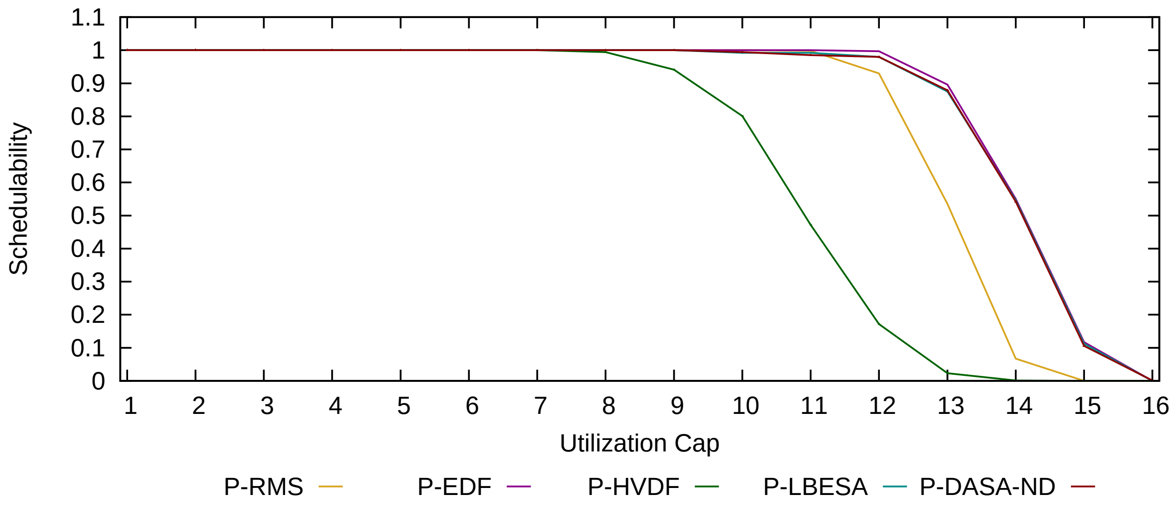
Figure A.6: 8-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

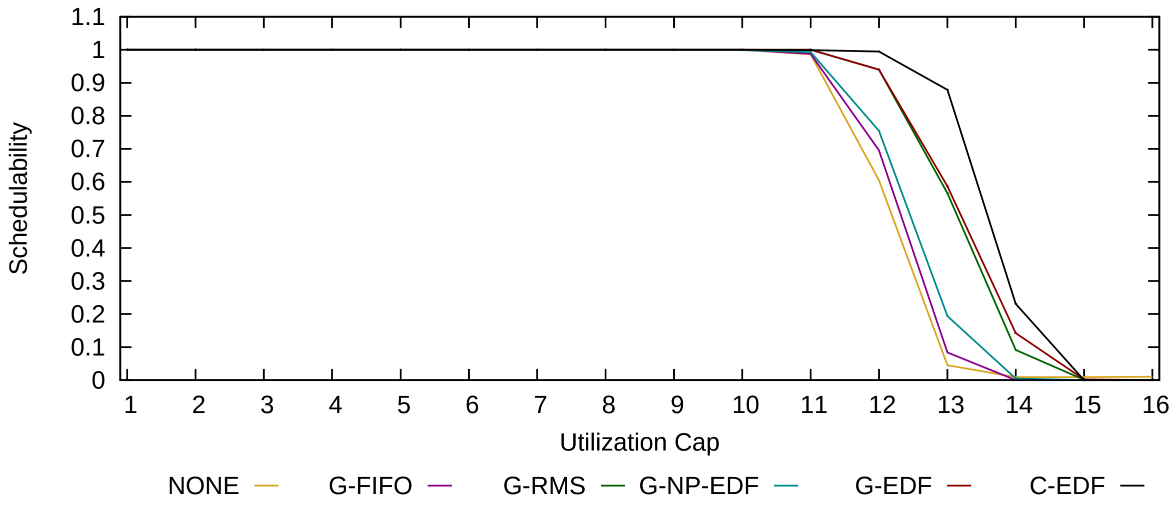


(b)

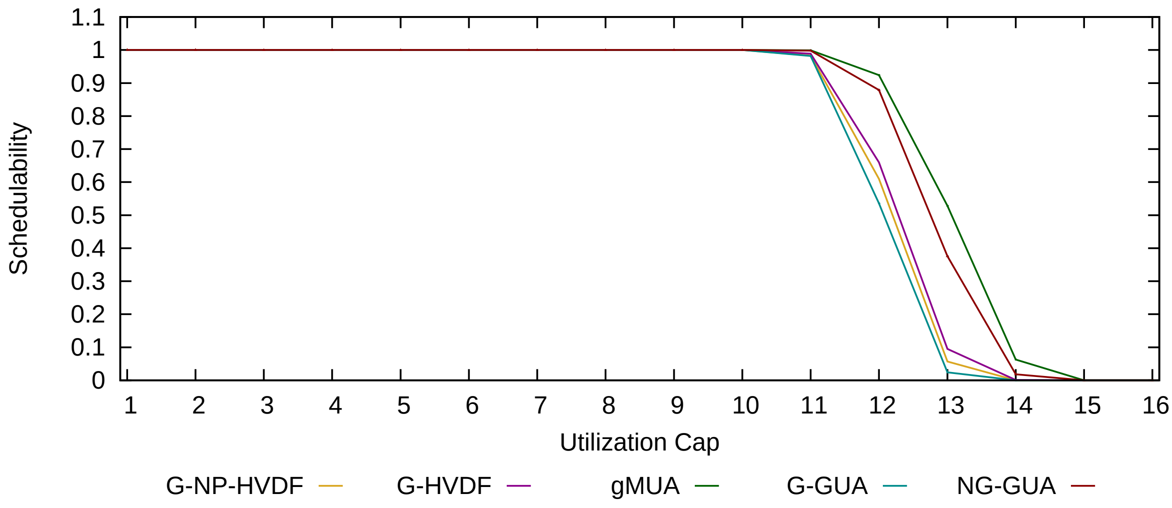


(c)

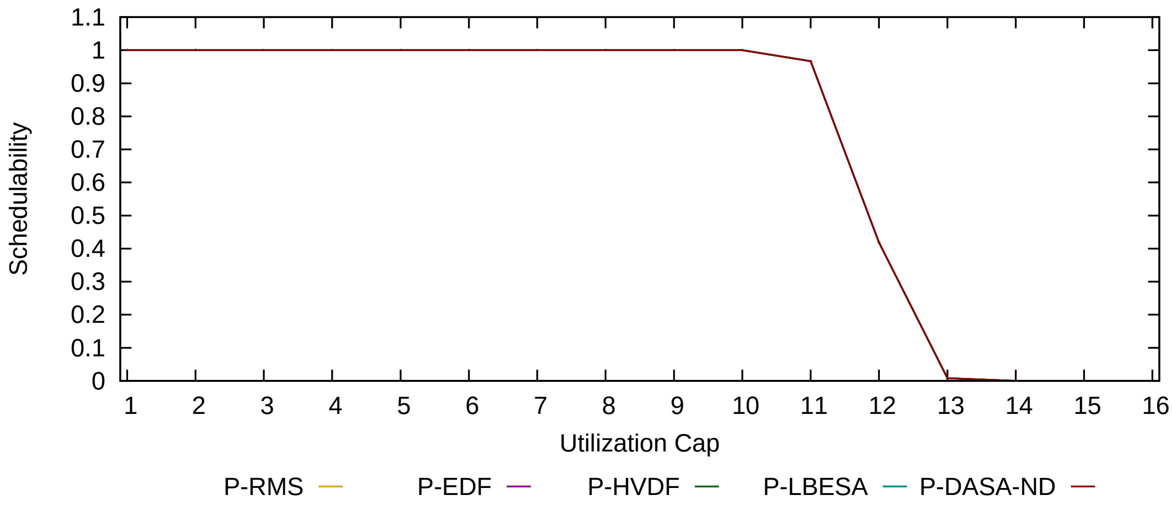
Figure A.7: 16-Core Schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

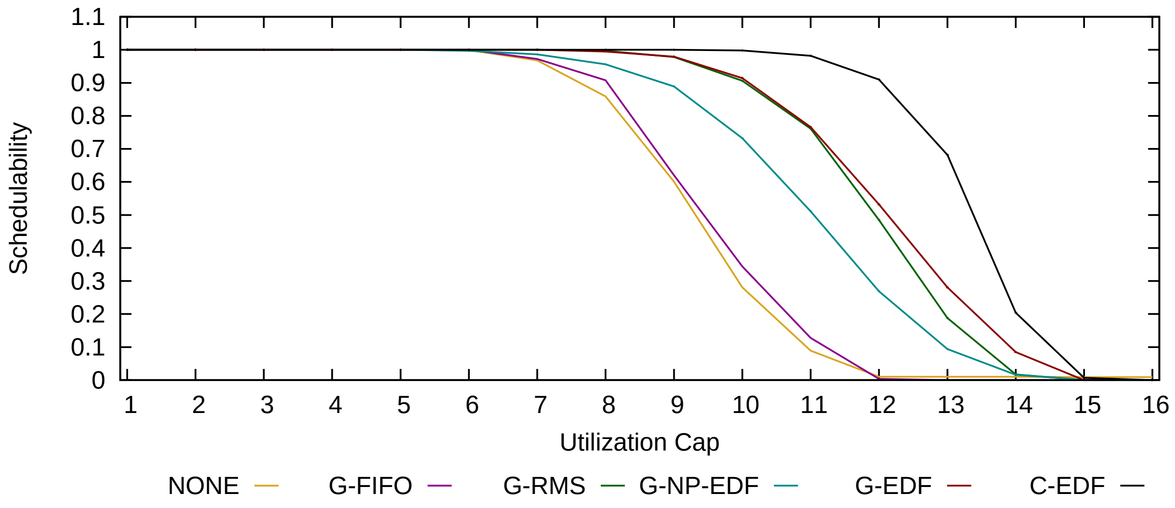


(b)

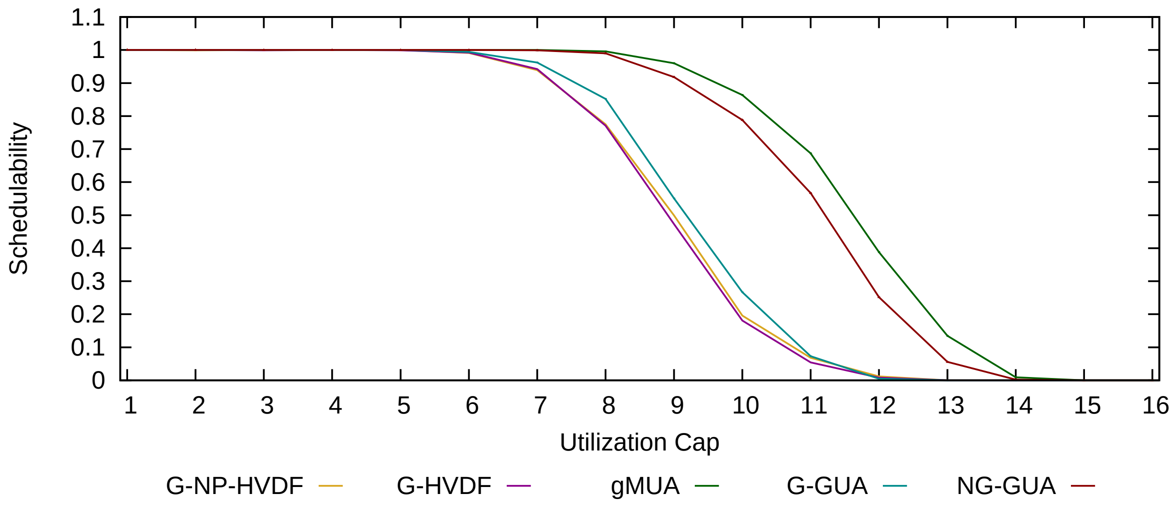


(c)

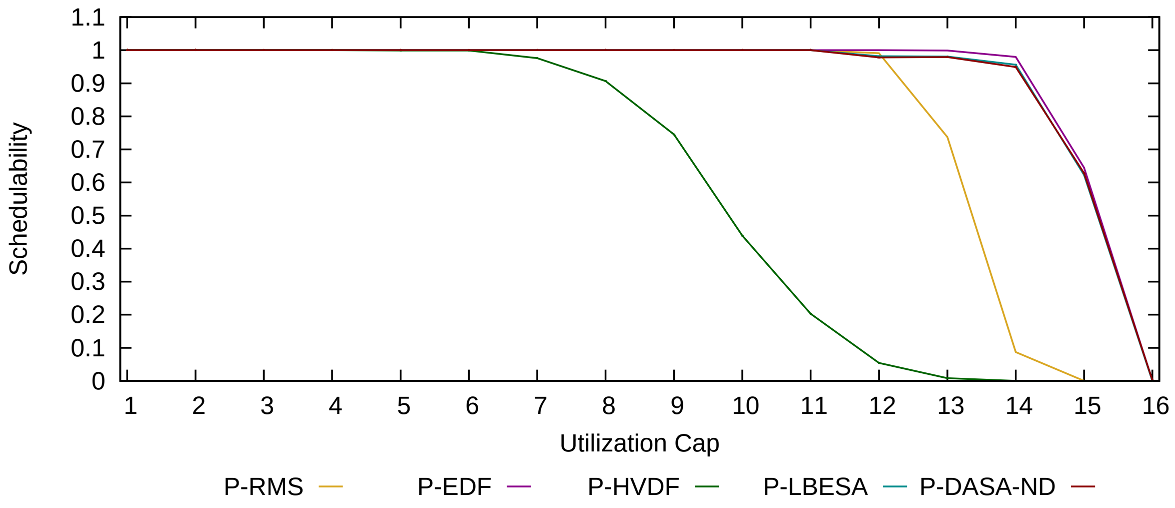
Figure A.8: 16-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

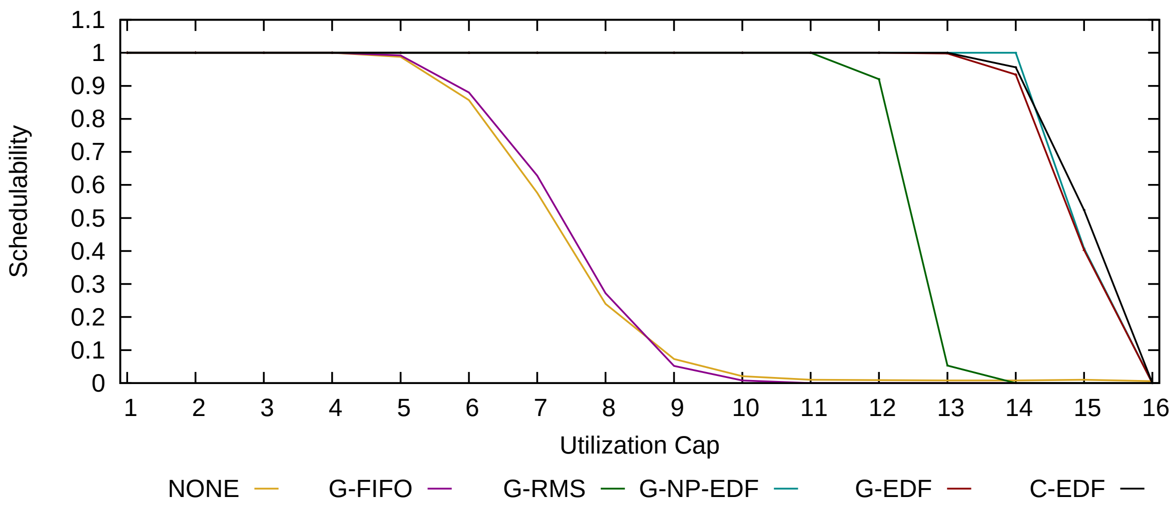


(b)

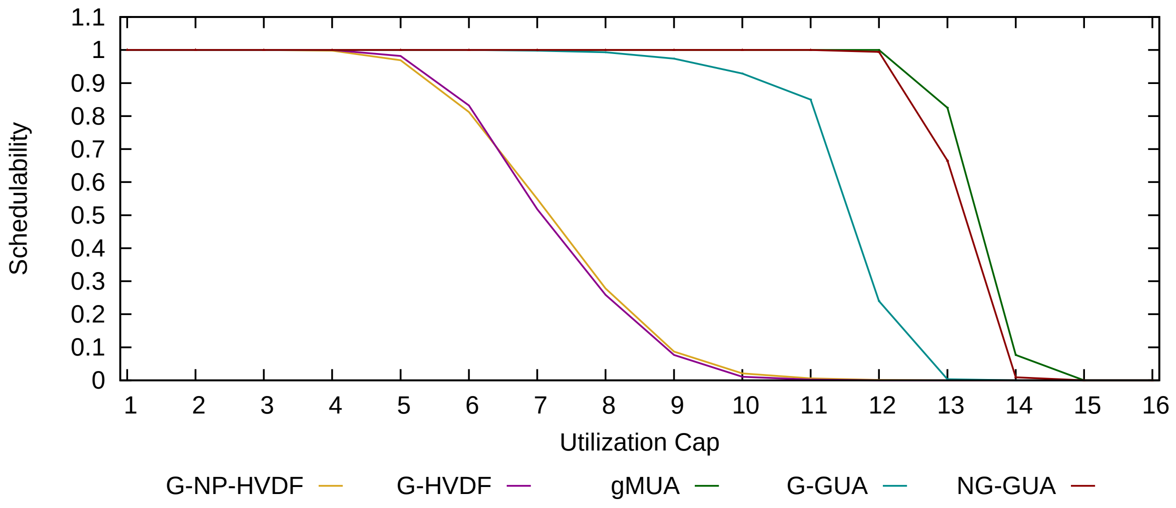


(c)

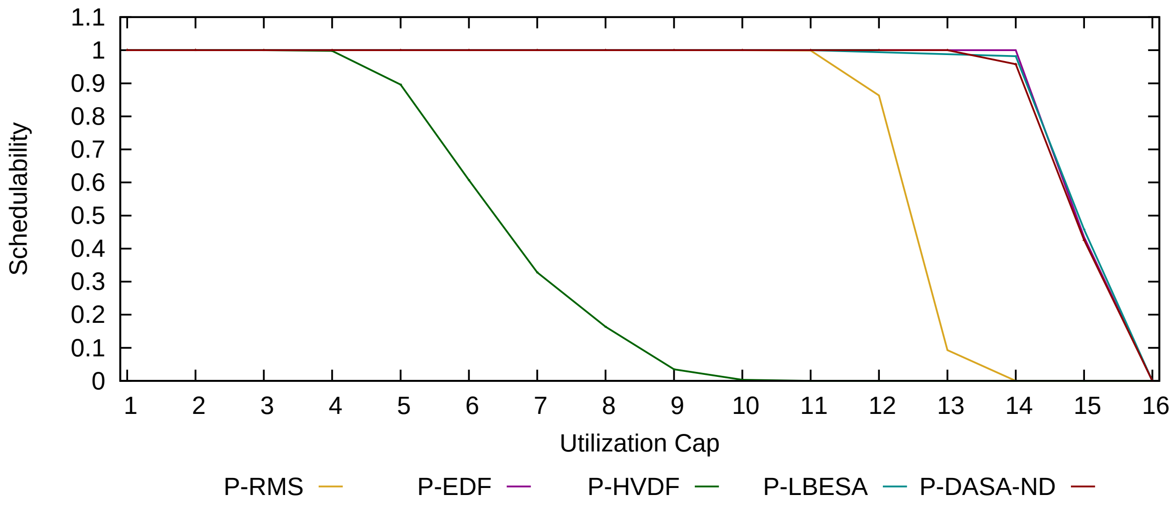
Figure A.9: 16-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

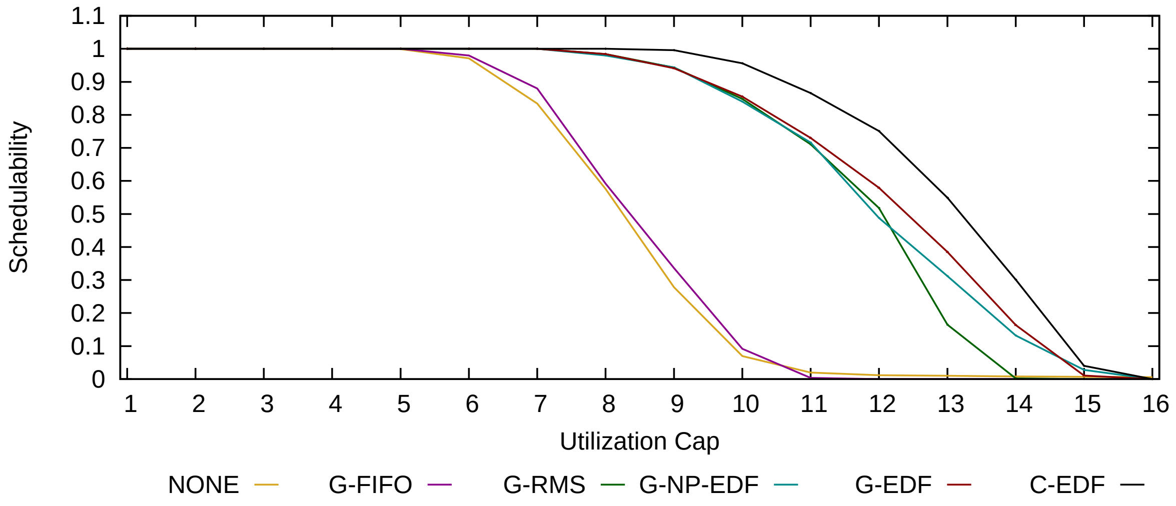


(b)

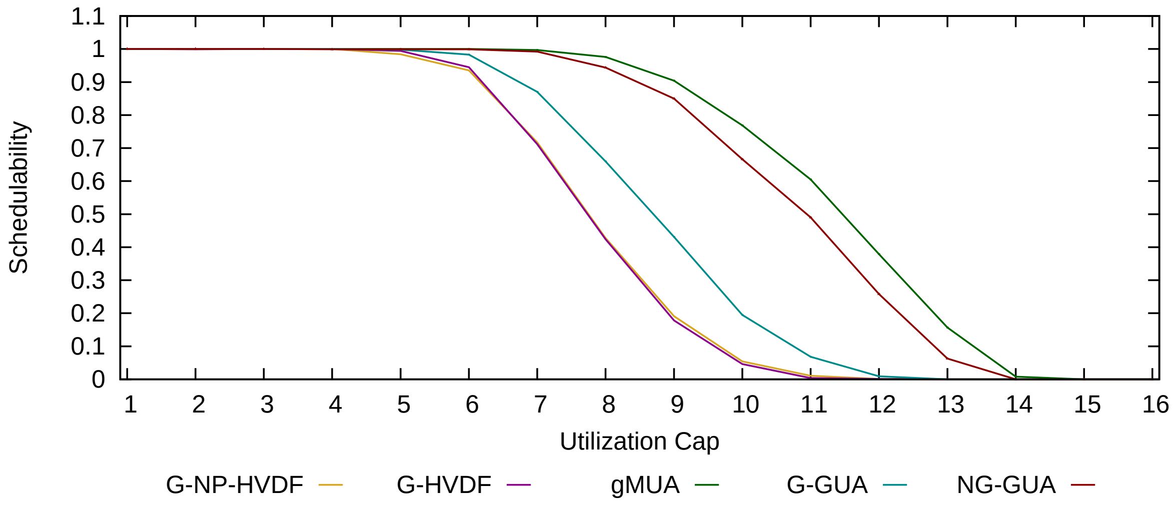


(c)

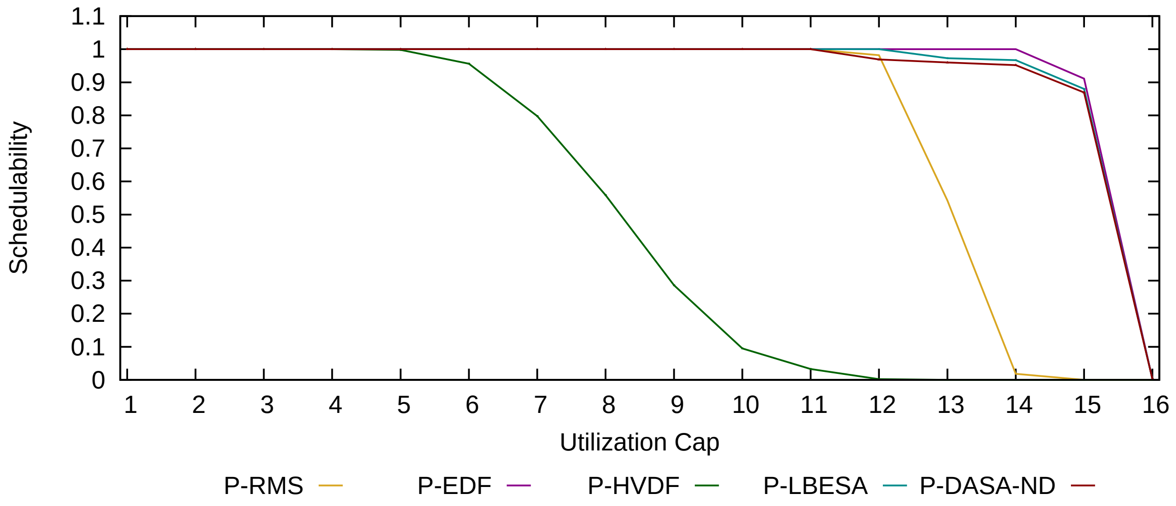
Figure A.10: 16-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

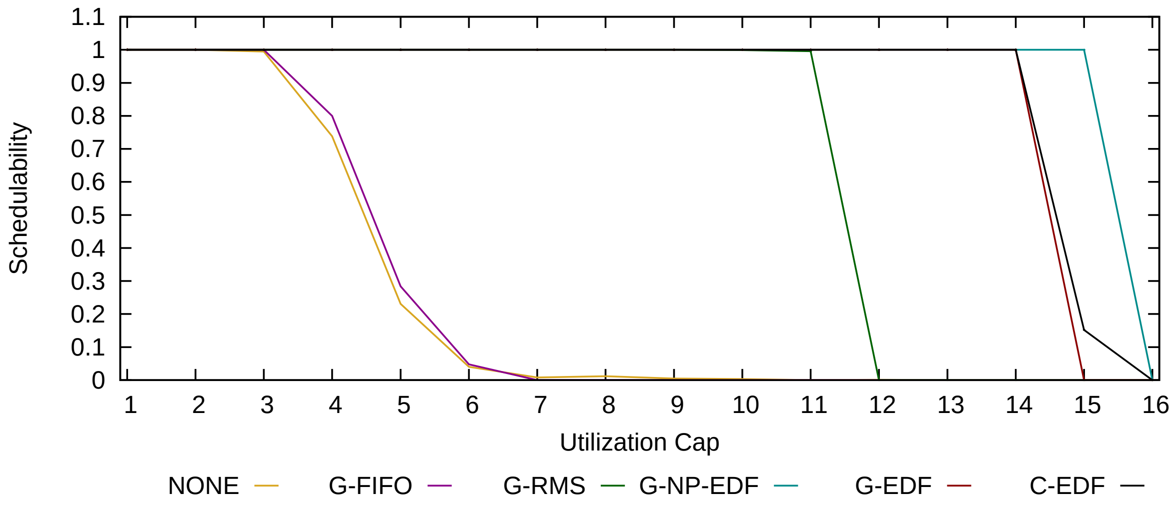


(b)

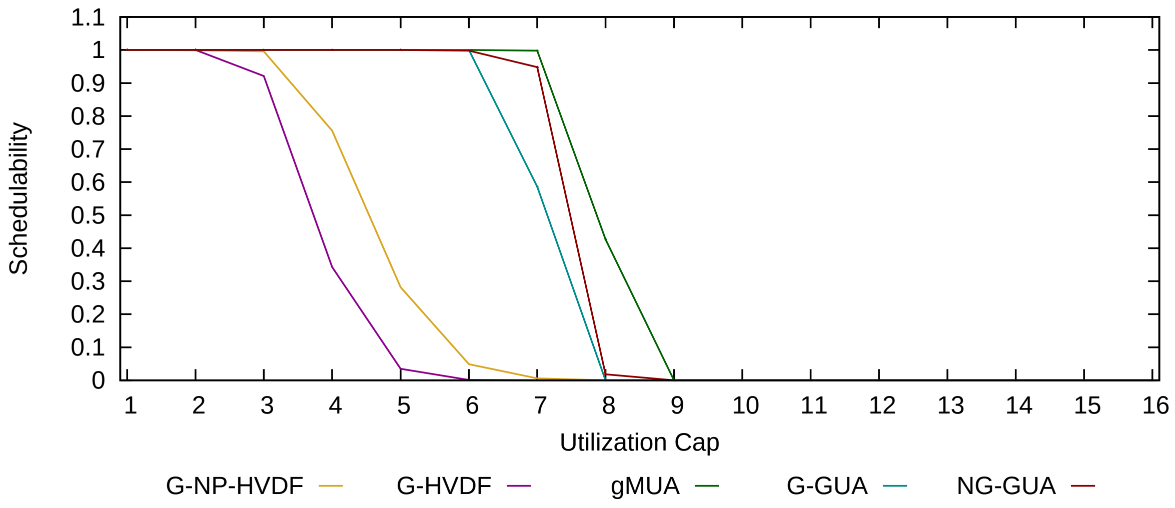


(c)

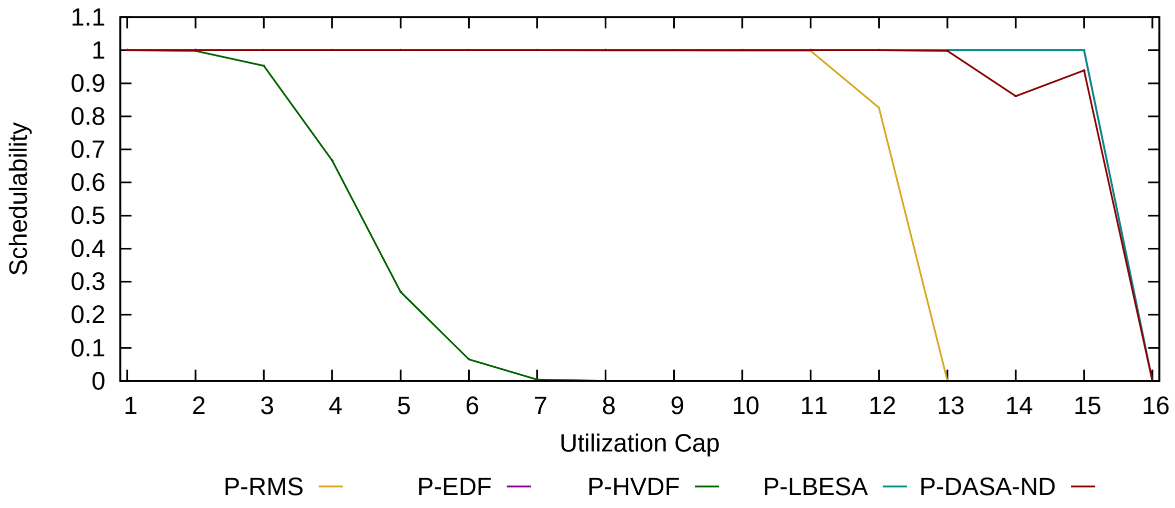
Figure A.11: 16-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

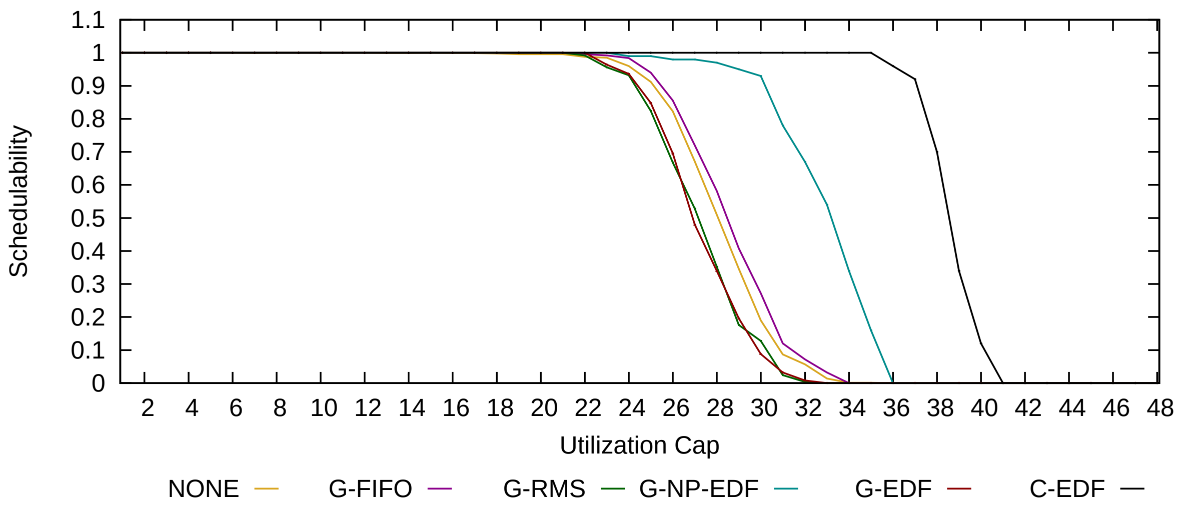


(b)

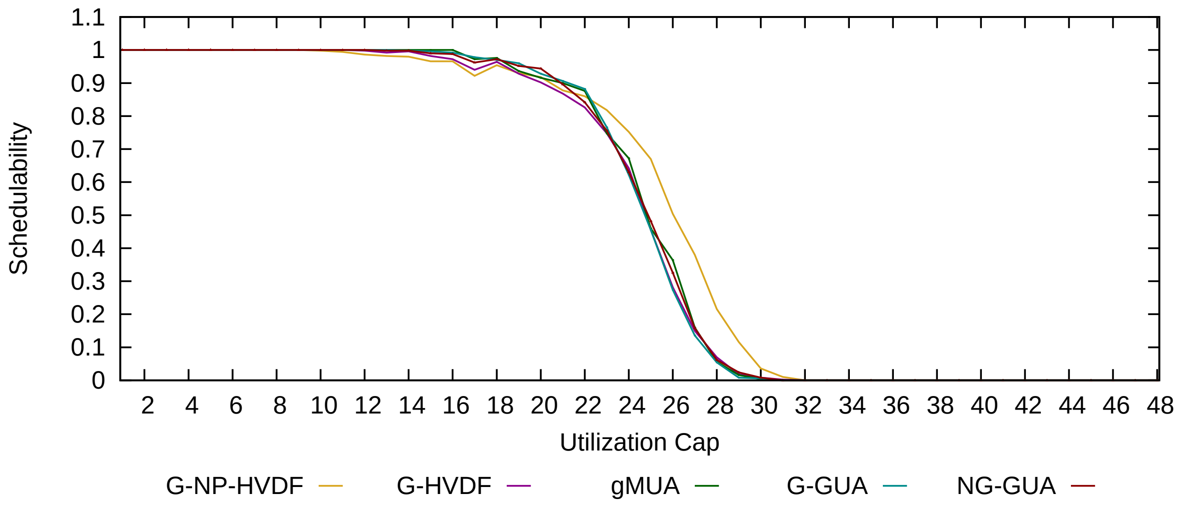


(c)

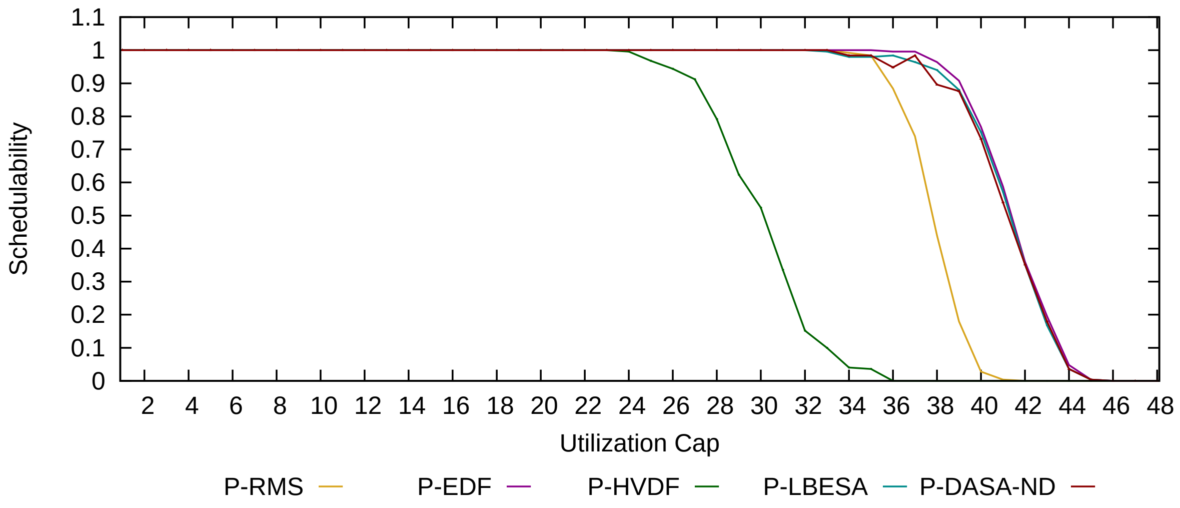
Figure A.12: 16-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

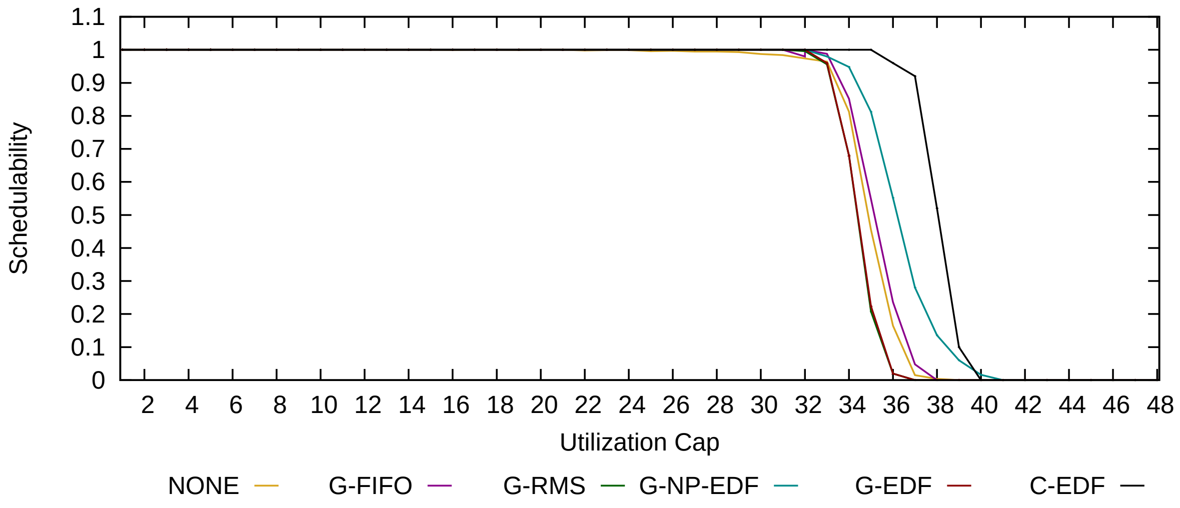


(b)

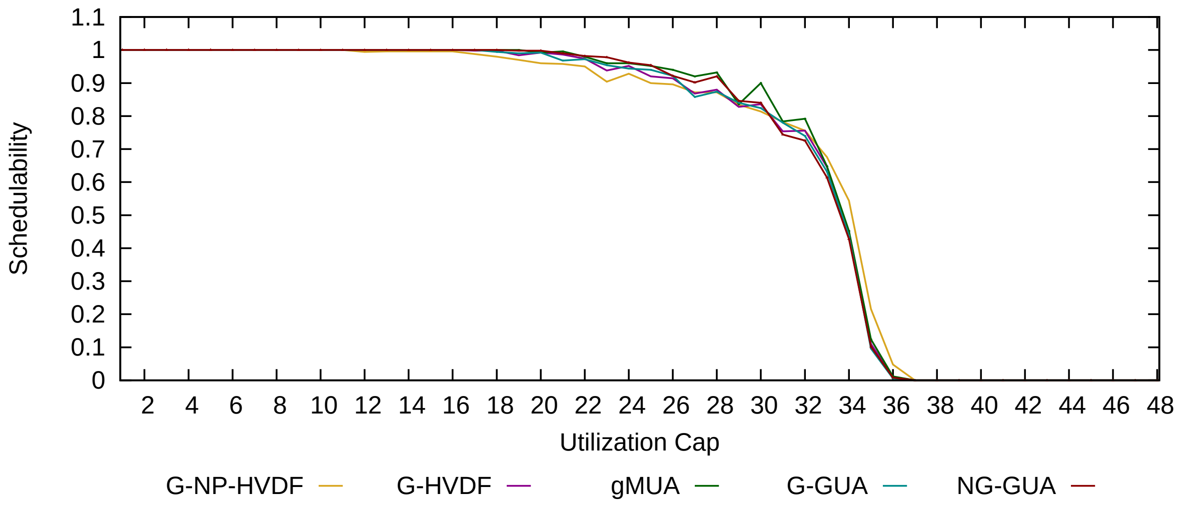


(c)

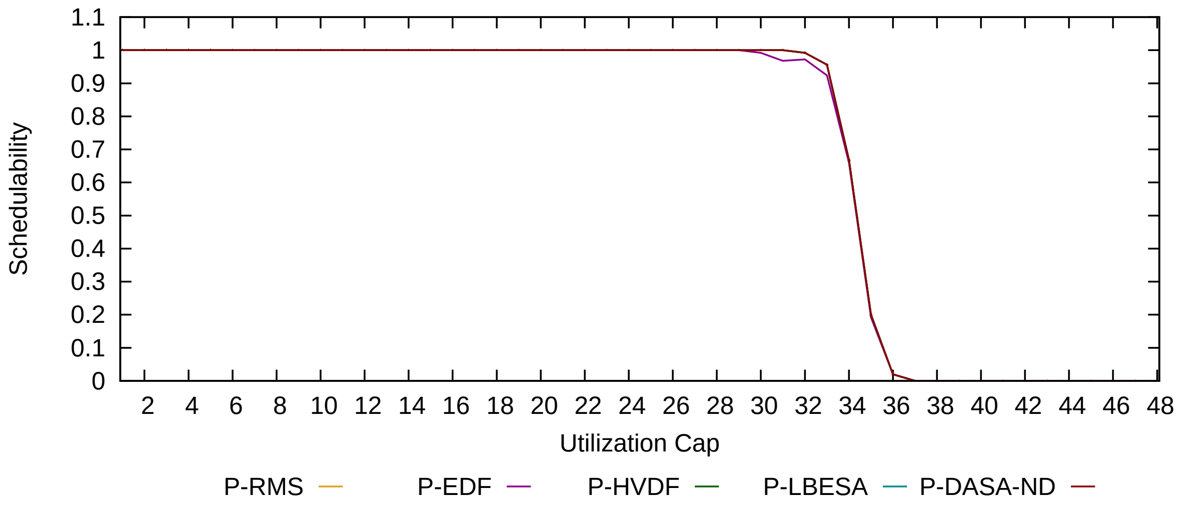
Figure A.13: 48-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

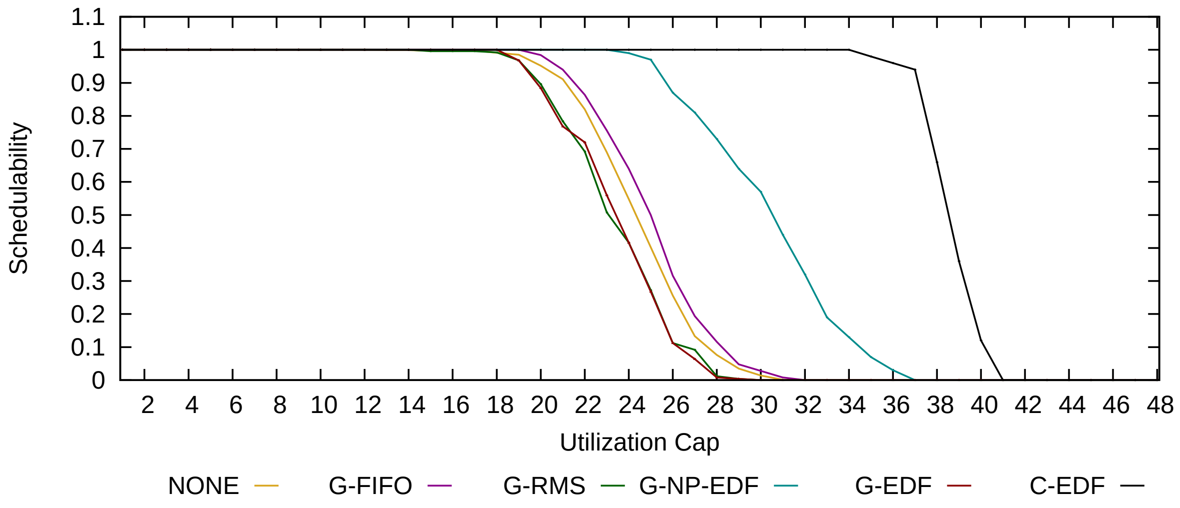


(b)

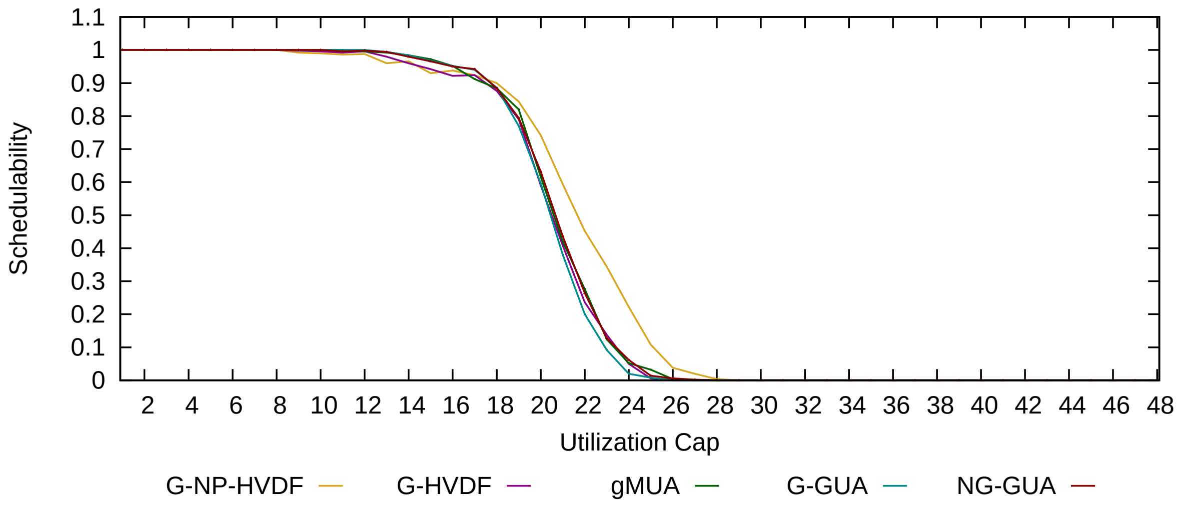


(c)

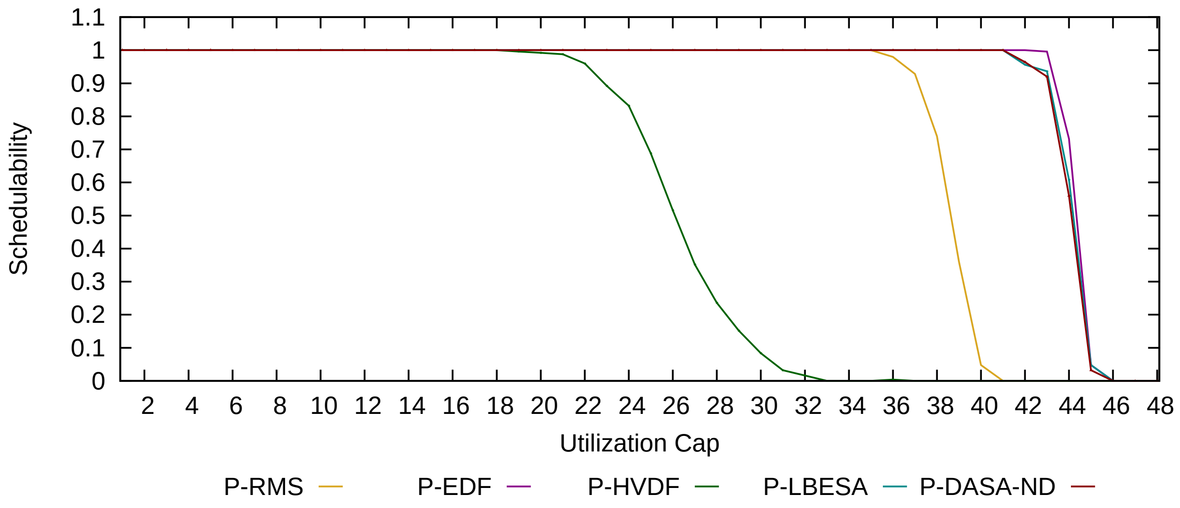
Figure A.14: 48-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

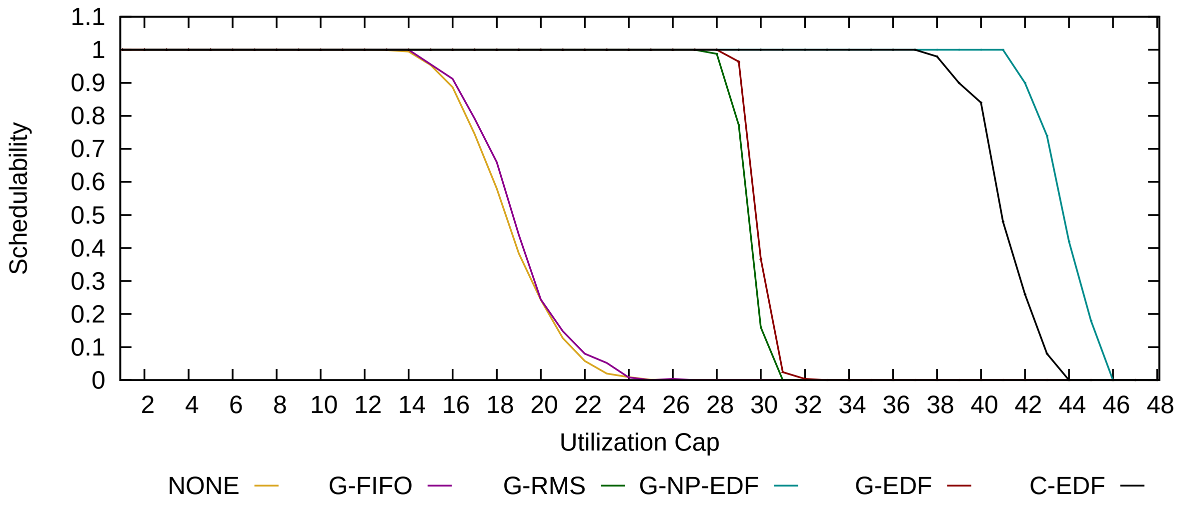


(b)

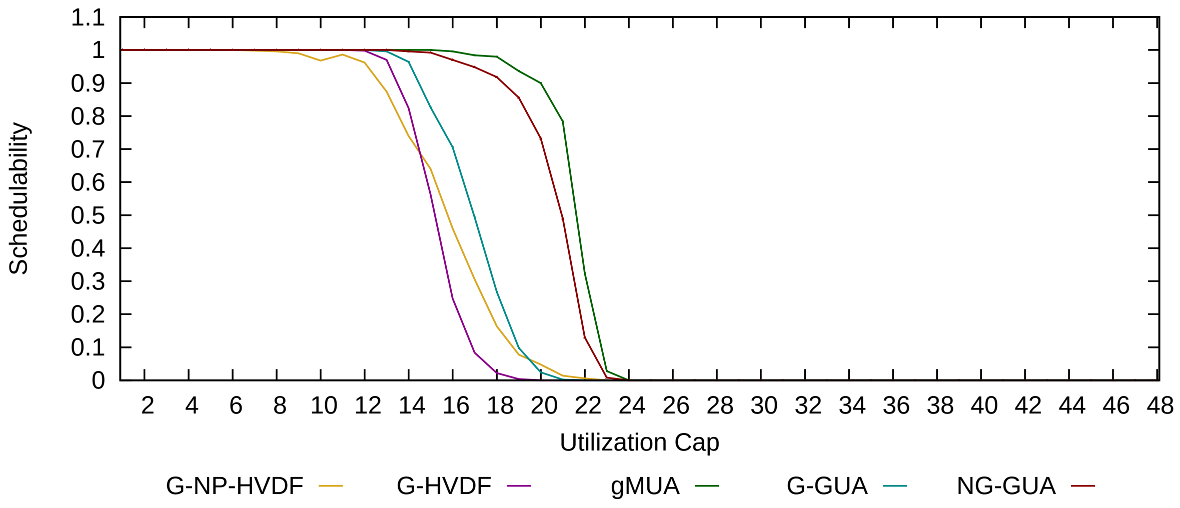


(c)

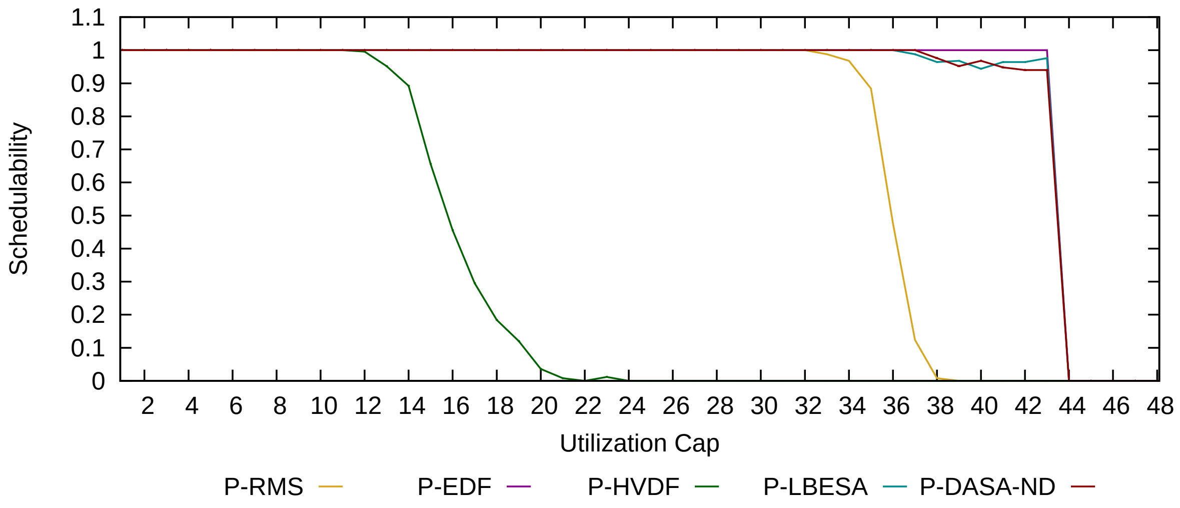
Figure A.15: 48-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

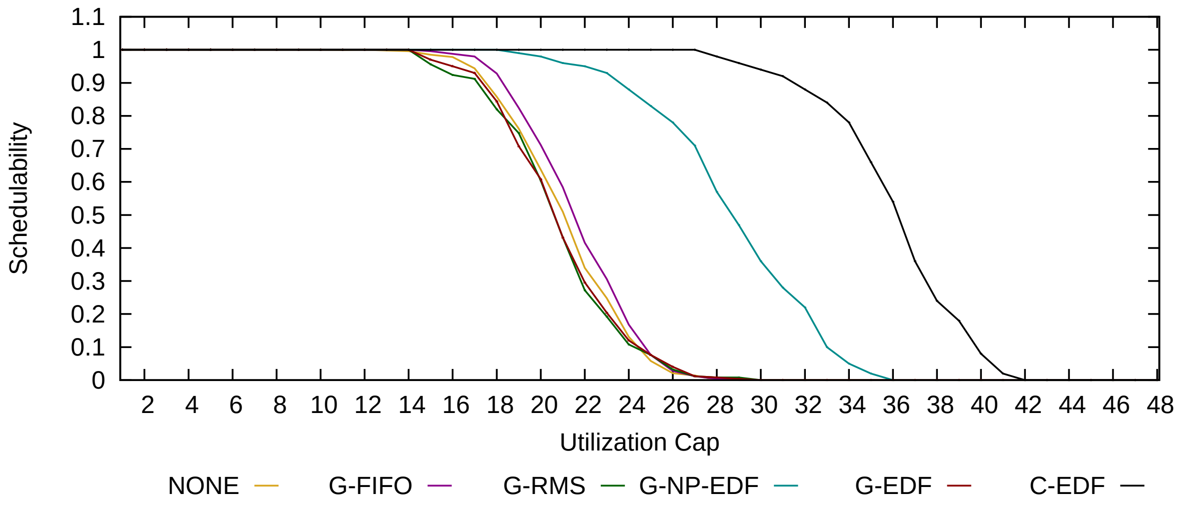


(b)

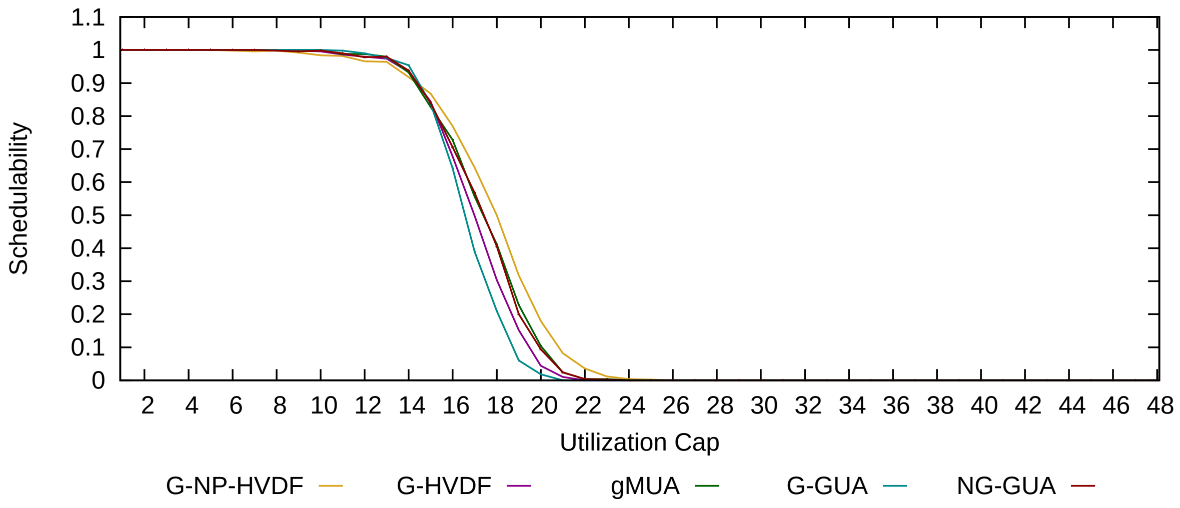


(c)

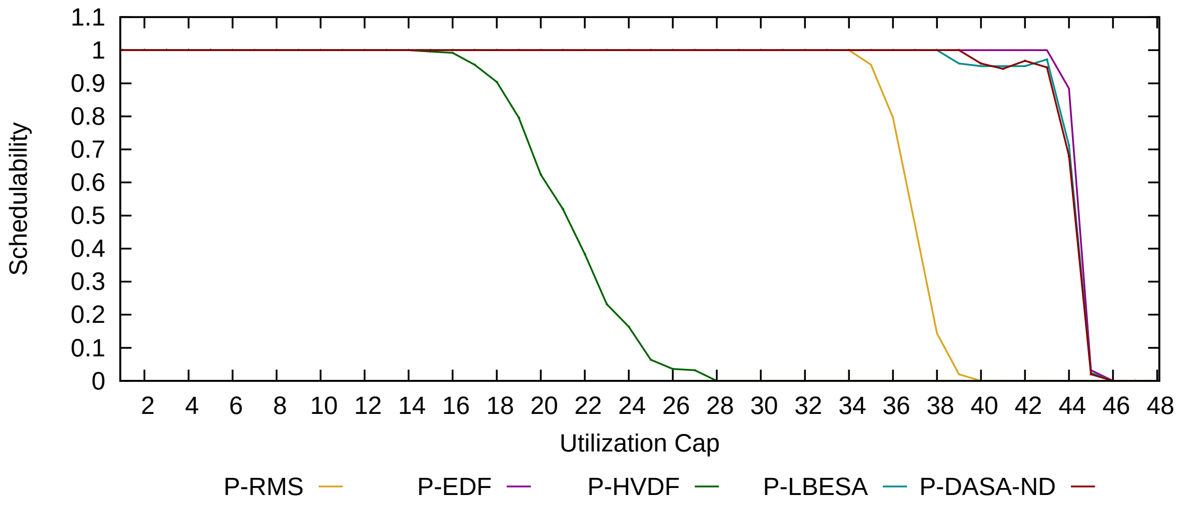
Figure A.16: 48-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

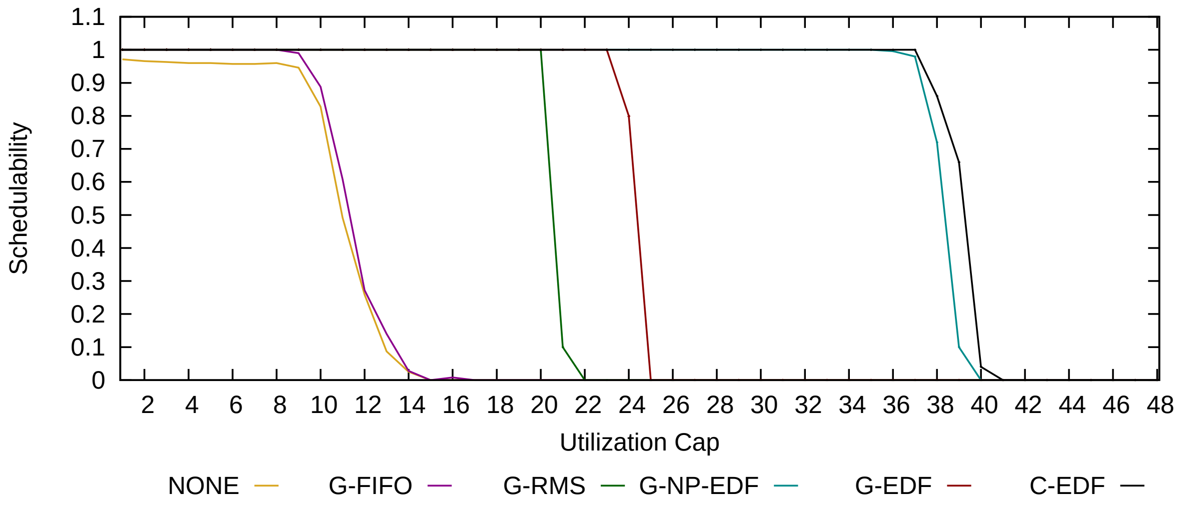


(b)

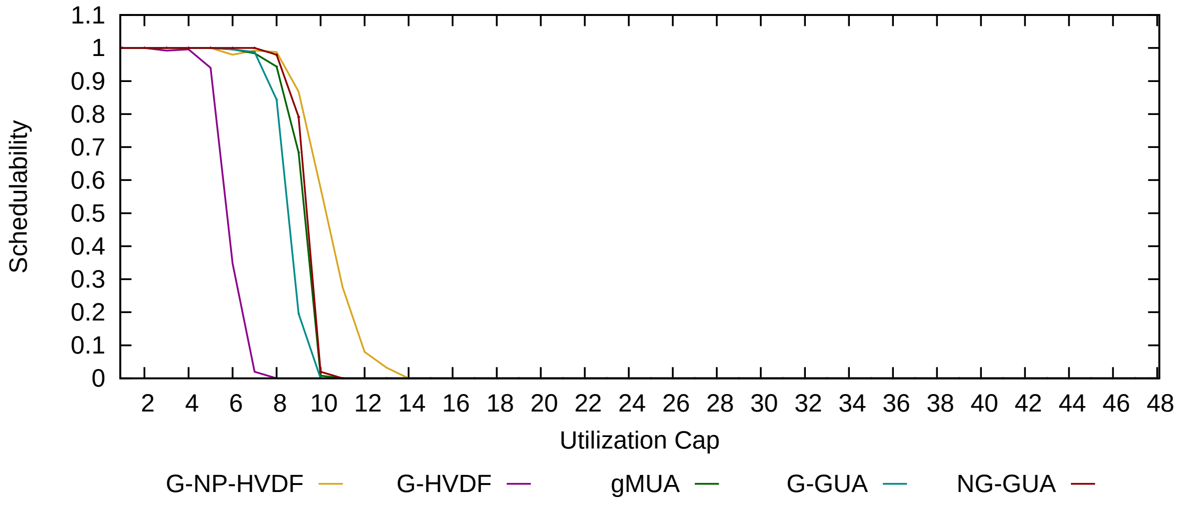


(c)

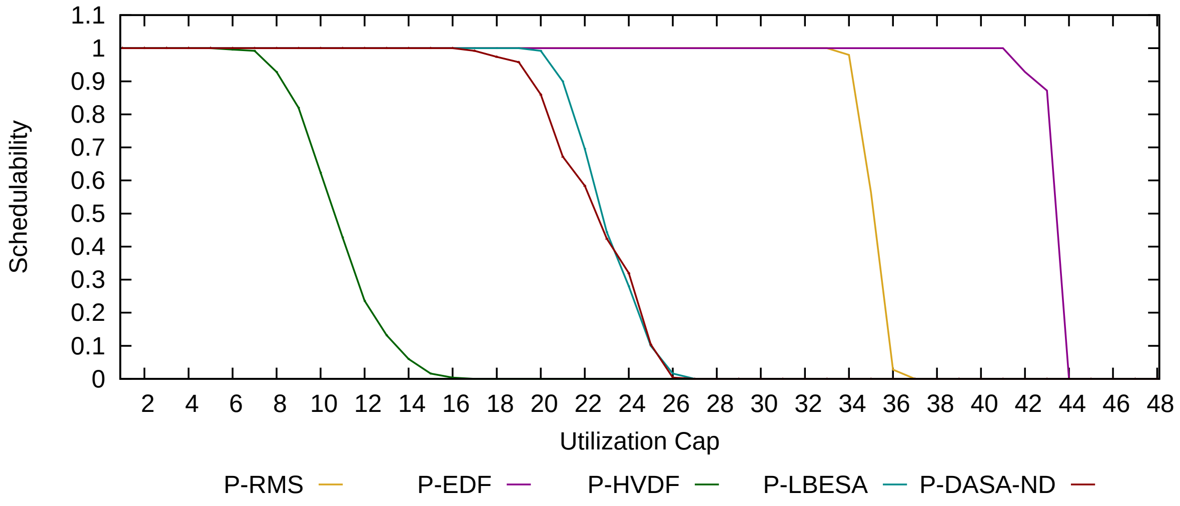
Figure A.17: 48-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)



(b)



(c)

Figure A.18: 48-Core schedulability results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions

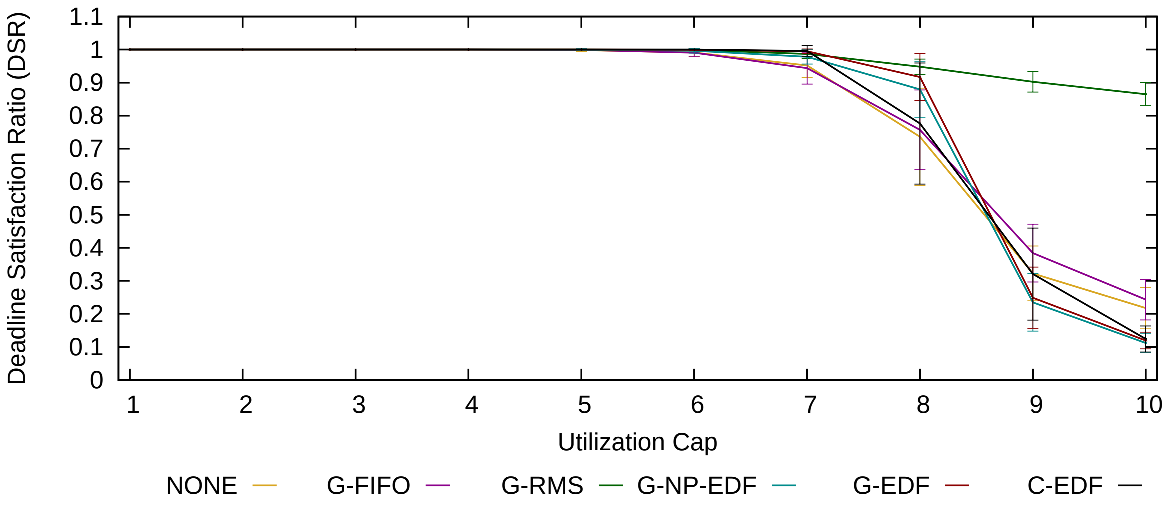
Appendix B

Complete Deadline Satisfaction Results

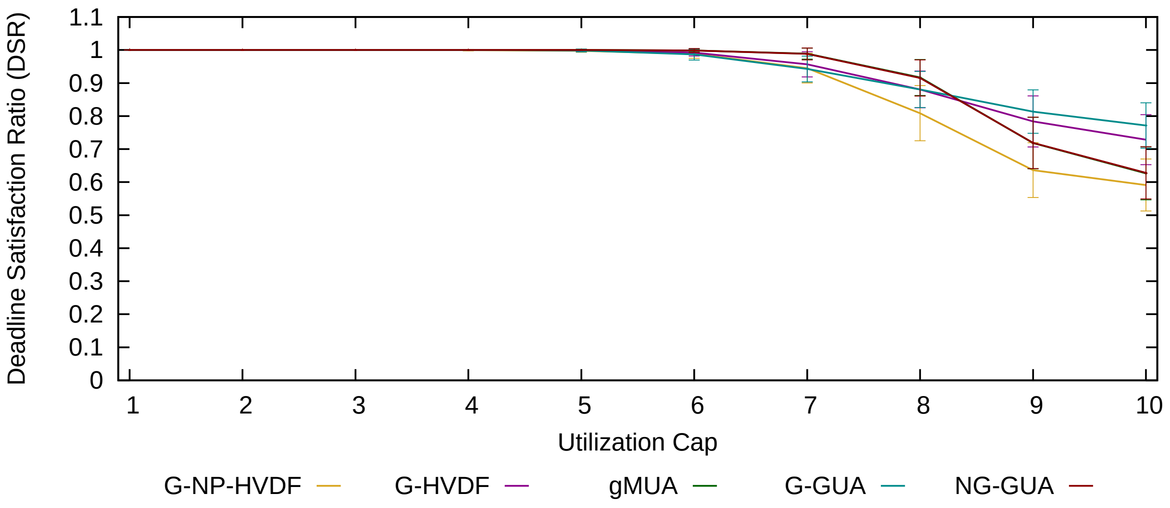
This appendix provides our complete deadline satisfaction ratio results. It contains 36 plots grouped in 12 figures by taskset distribution and machine. It is organized as follows:

- Figure B.1 shows DSR results for our 8-core platform under heavy bimodal load
- Figure B.2 shows DSR results for our 8-core platform under heavy uniform load
- Figure B.3 shows DSR results for our 8-core platform under medium bimodal load
- Figure B.4 shows DSR results for our 8-core platform under medium uniform load
- Figure B.5 shows DSR results for our 8-core platform under light bimodal load
- Figure B.6 shows DSR results for our 8-core platform under light uniform load
- Figure B.7 shows DSR results for our 16-core platform under heavy bimodal load
- Figure B.8 shows DSR results for our 16-core platform under heavy uniform load
- Figure B.9 shows DSR results for our 16-core platform under medium bimodal load
- Figure B.10 shows DSR results for our 16-core platform under medium uniform load
- Figure B.11 shows DSR results for our 16-core platform under light bimodal load
- Figure B.12 shows DSR results for our 16-core platform under light uniform load
- Figure B.13 shows DSR results for our 48-core platform under heavy bimodal load
- Figure B.14 shows DSR results for our 48-core platform under heavy uniform load

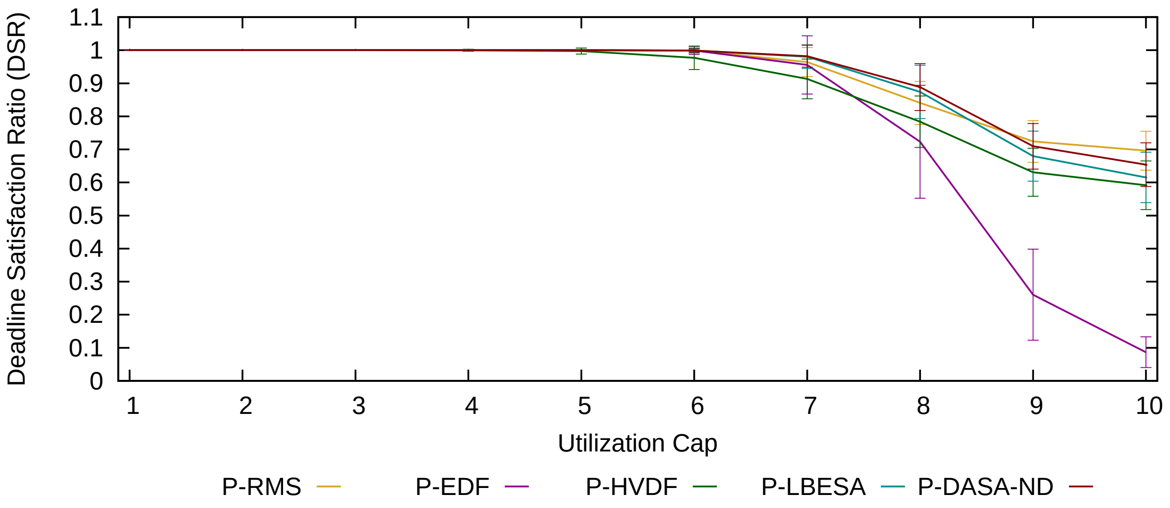
- Figure B.15 shows DSR results for our 48-core platform under medium bimodal load
- Figure B.16 shows DSR results for our 48-core platform under medium uniform load
- Figure B.17 shows DSR results for our 48-core platform under light bimodal load
- Figure B.18 shows DSR results for our 48-core platform under light uniform load



(a)

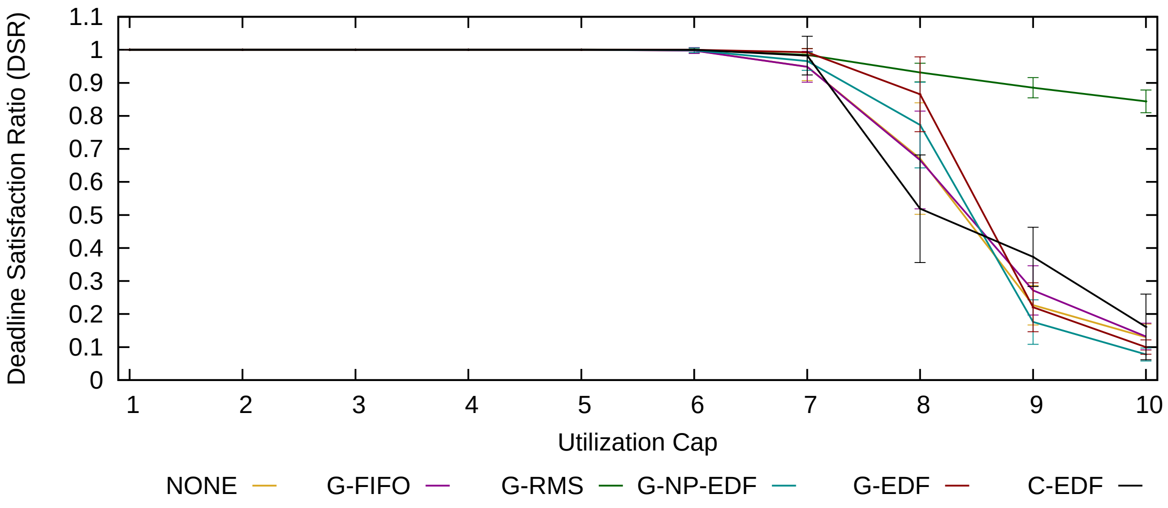


(b)

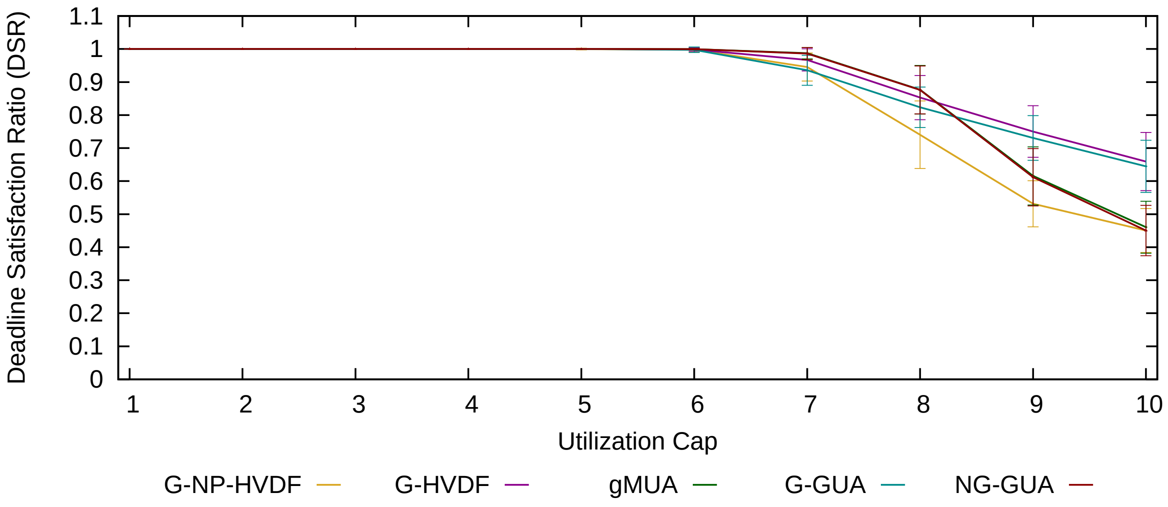


(c)

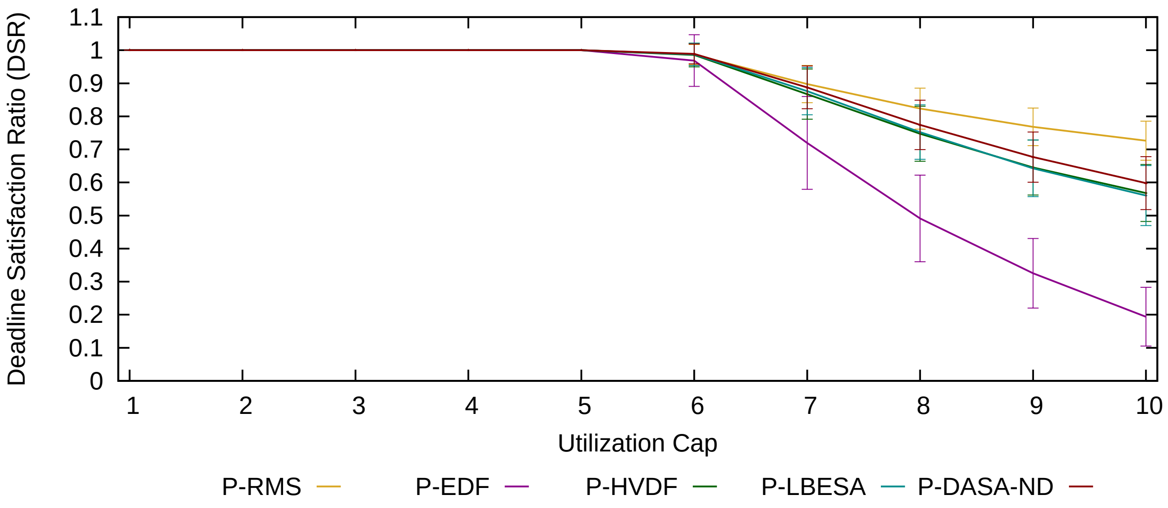
Figure B.1: 8-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

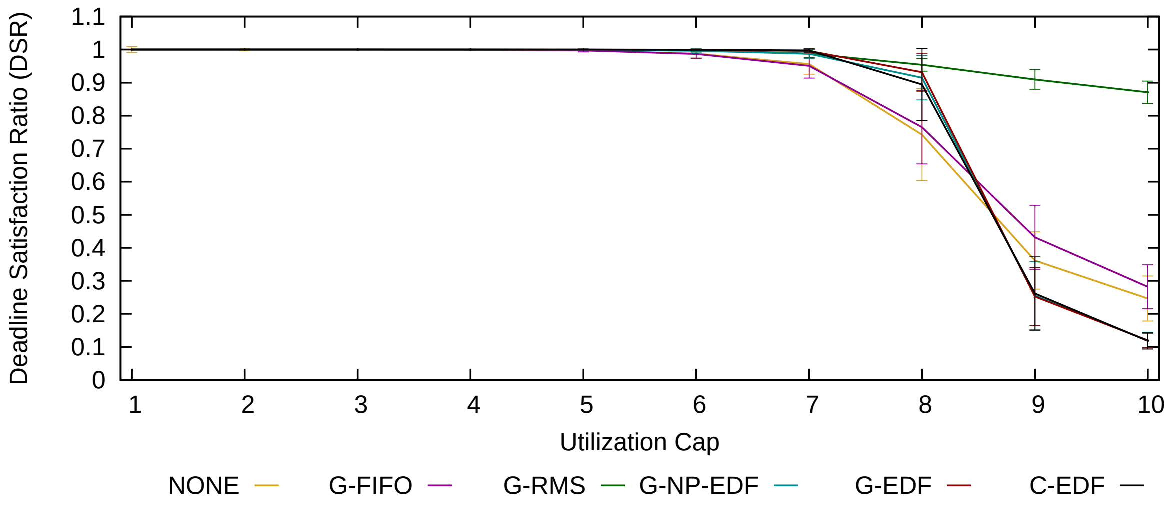


(b)

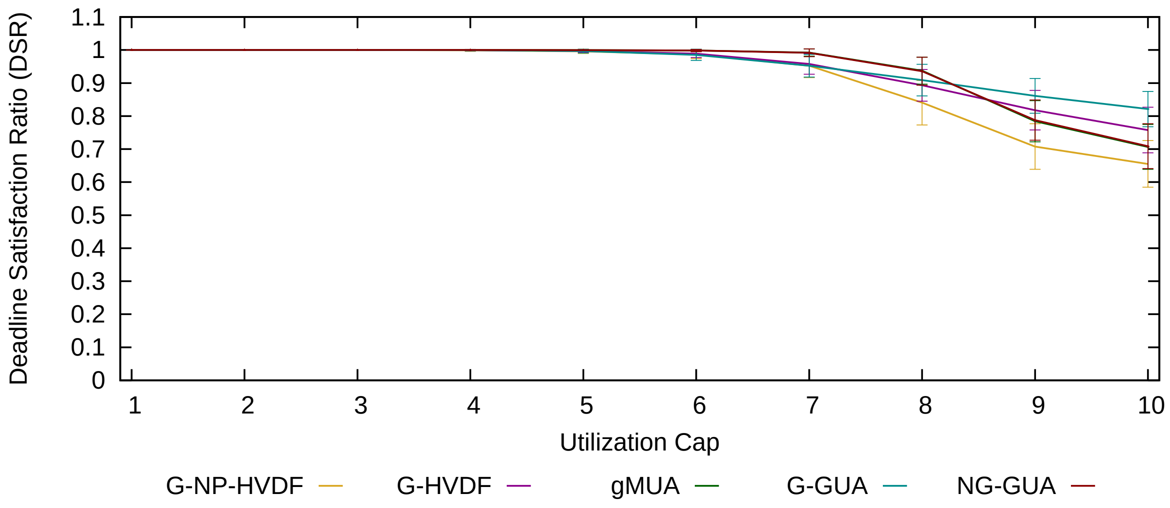


(c)

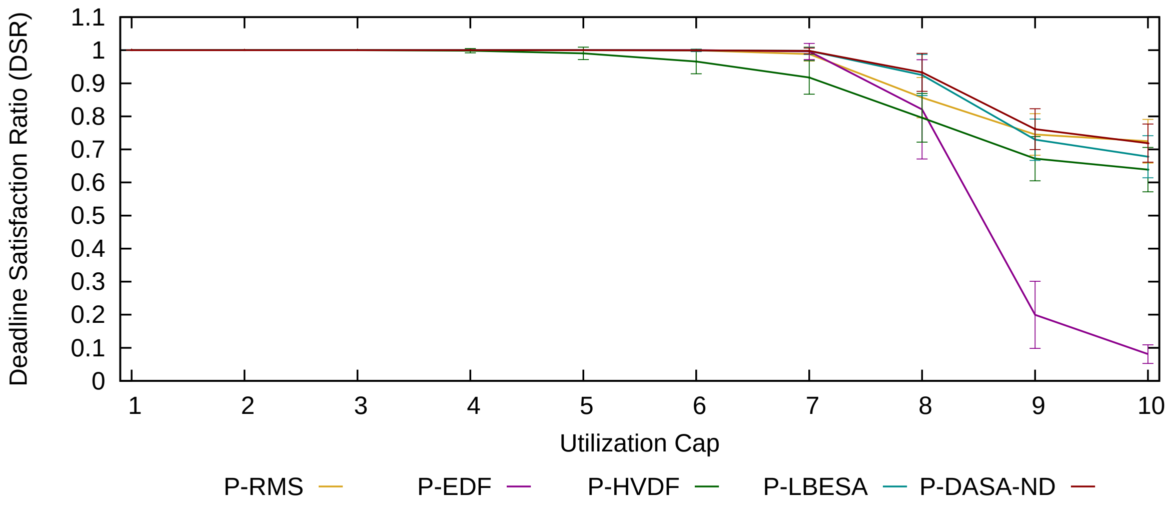
Figure B.2: 8-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

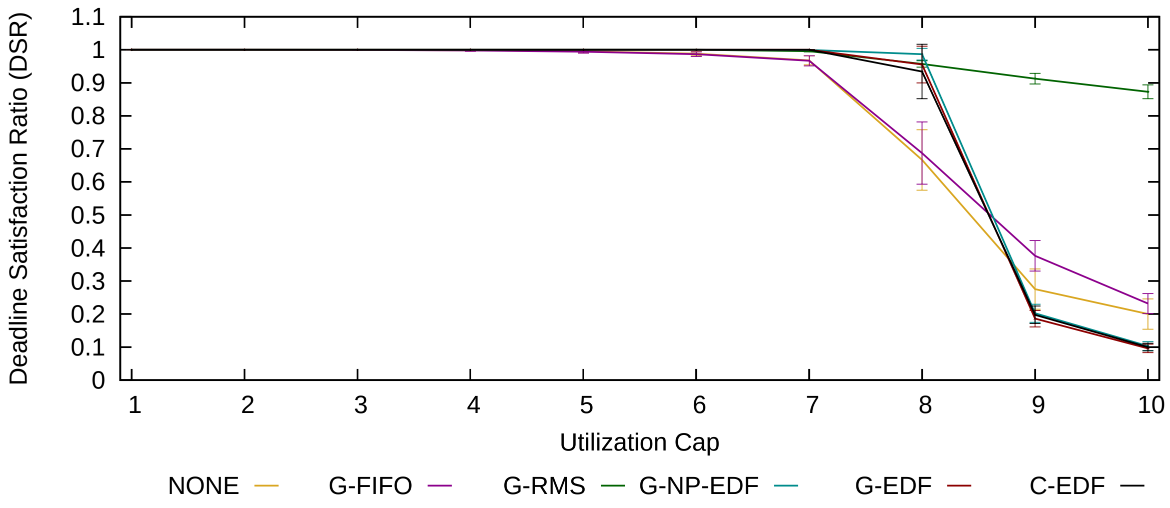


(b)

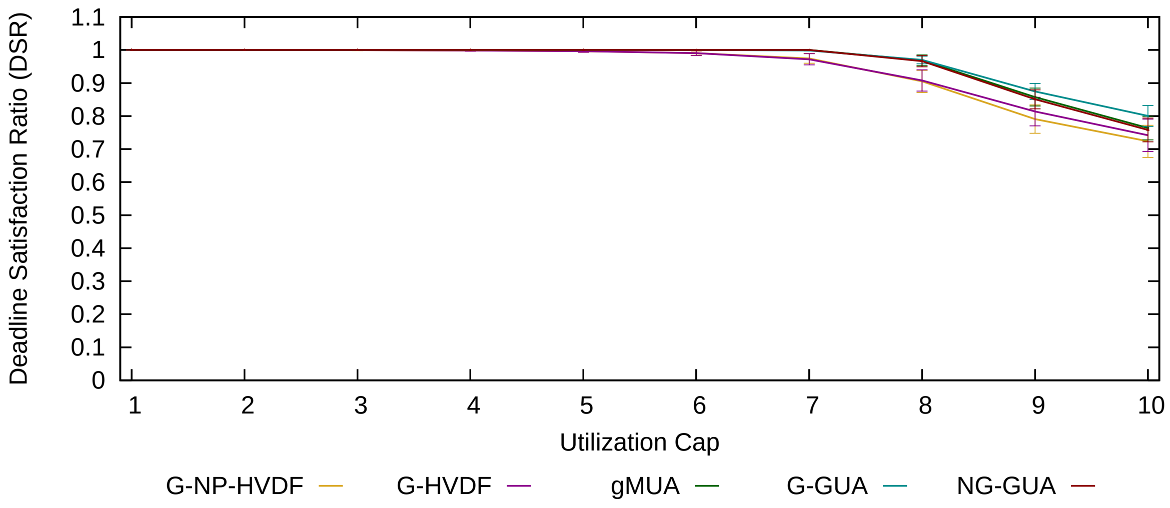


(c)

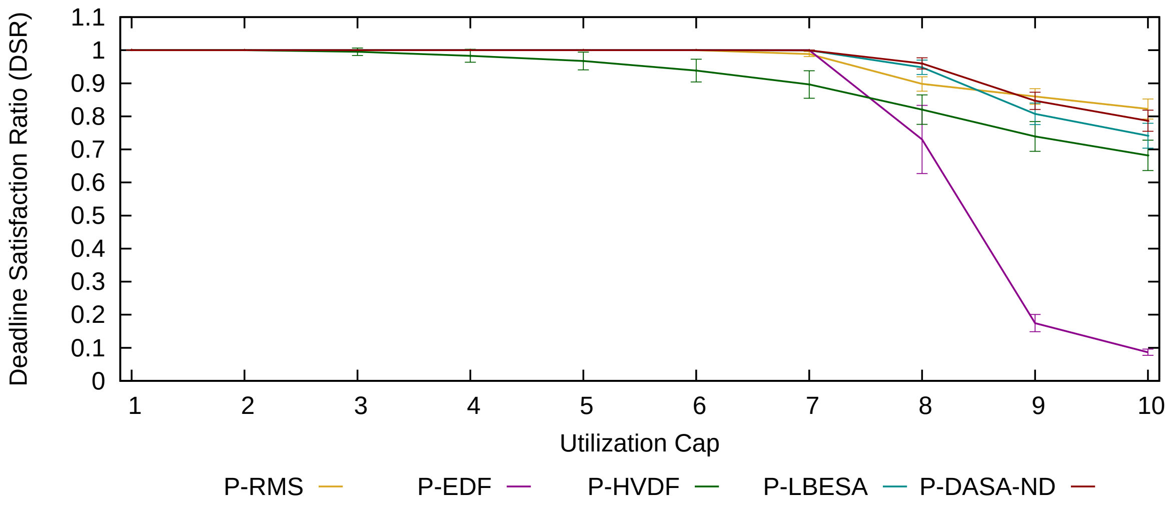
Figure B.3: 8-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

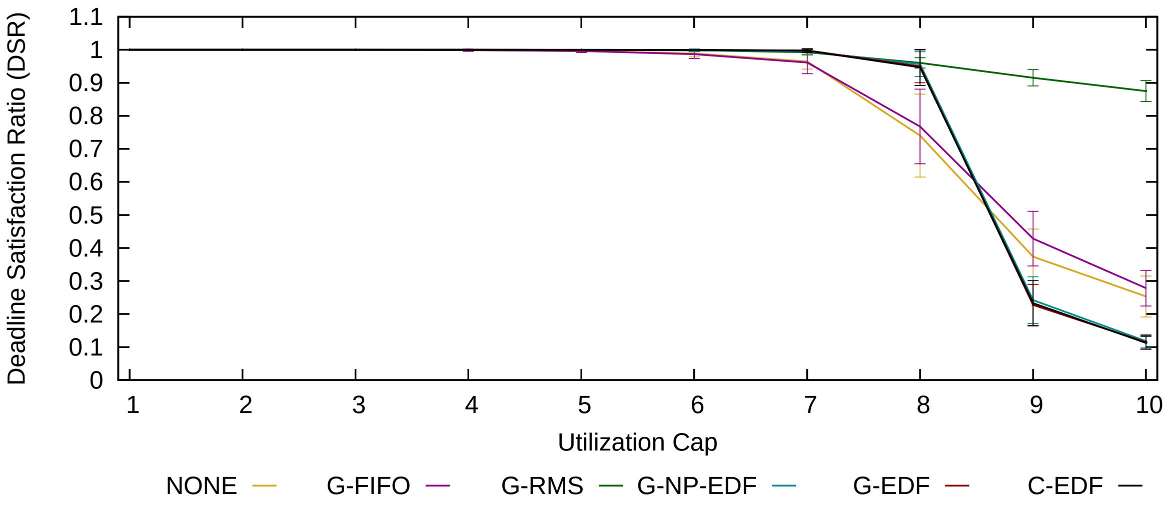


(b)

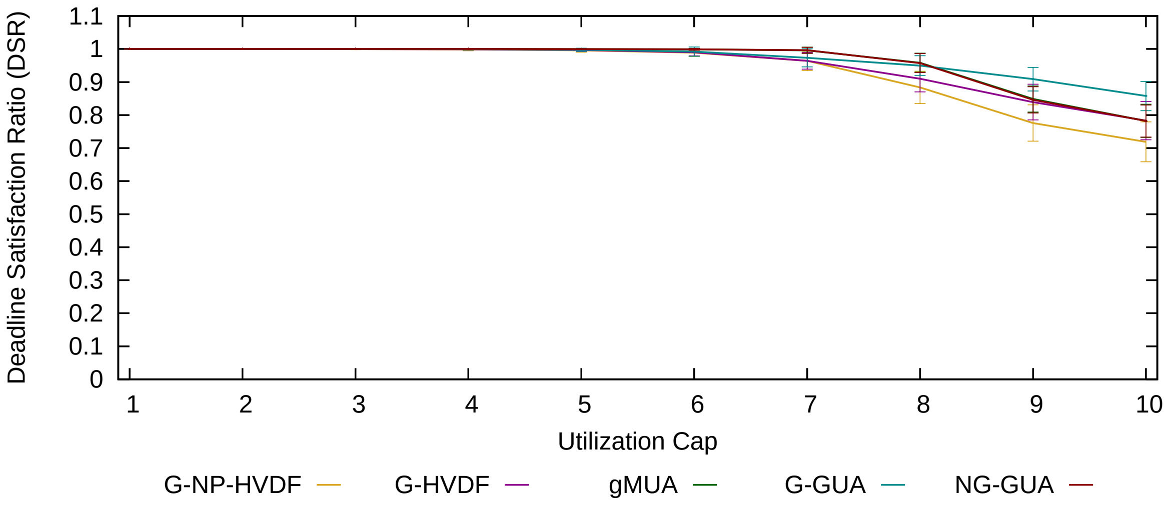


(c)

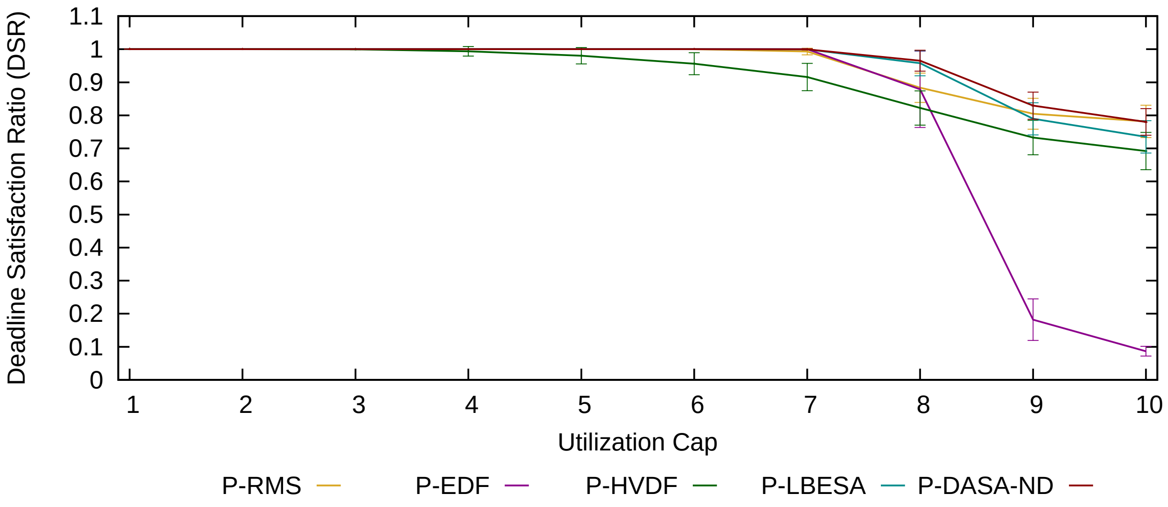
Figure B.4: 8-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

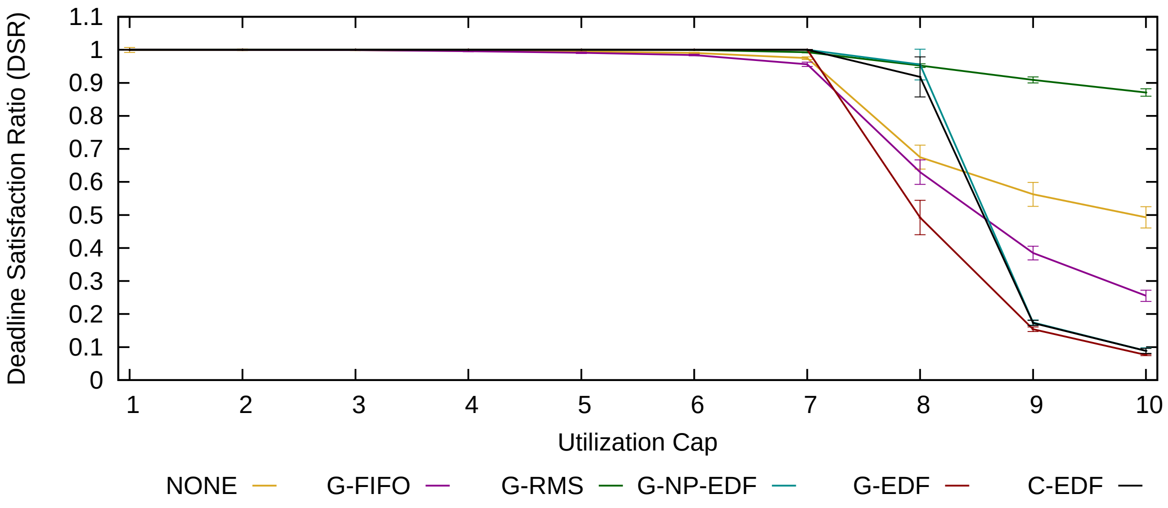


(b)

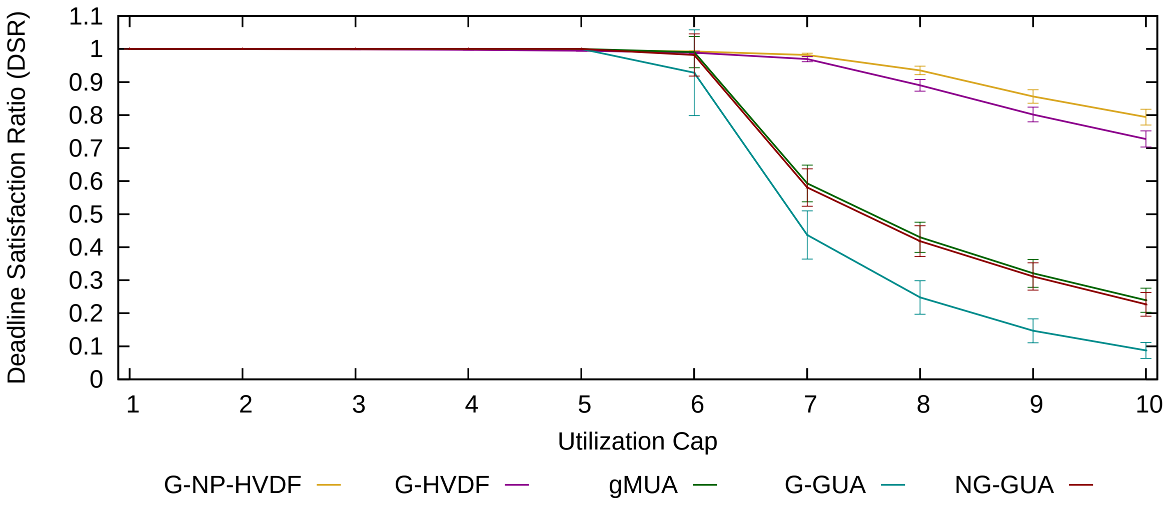


(c)

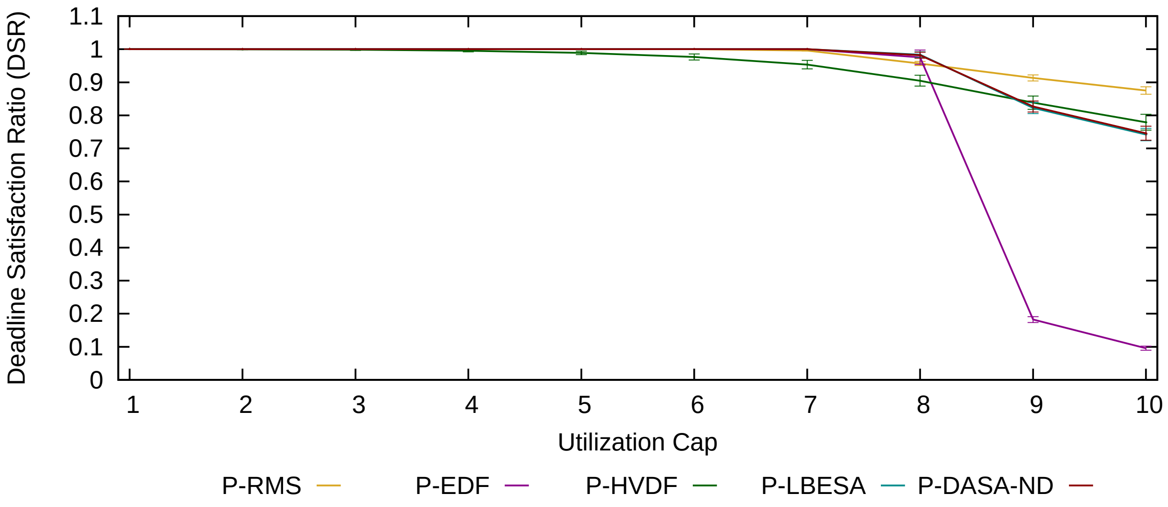
Figure B.5: 8-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

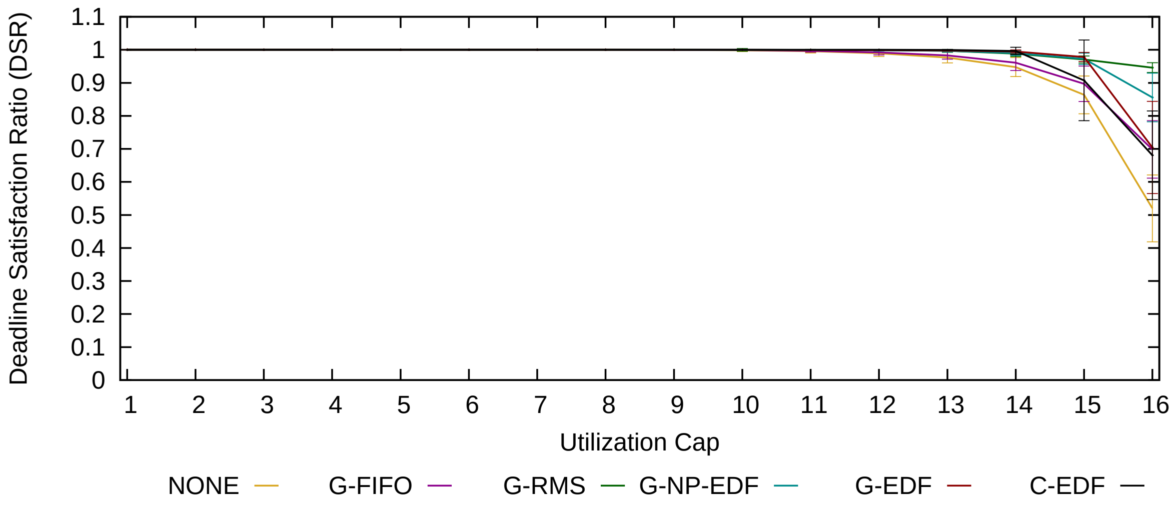


(b)

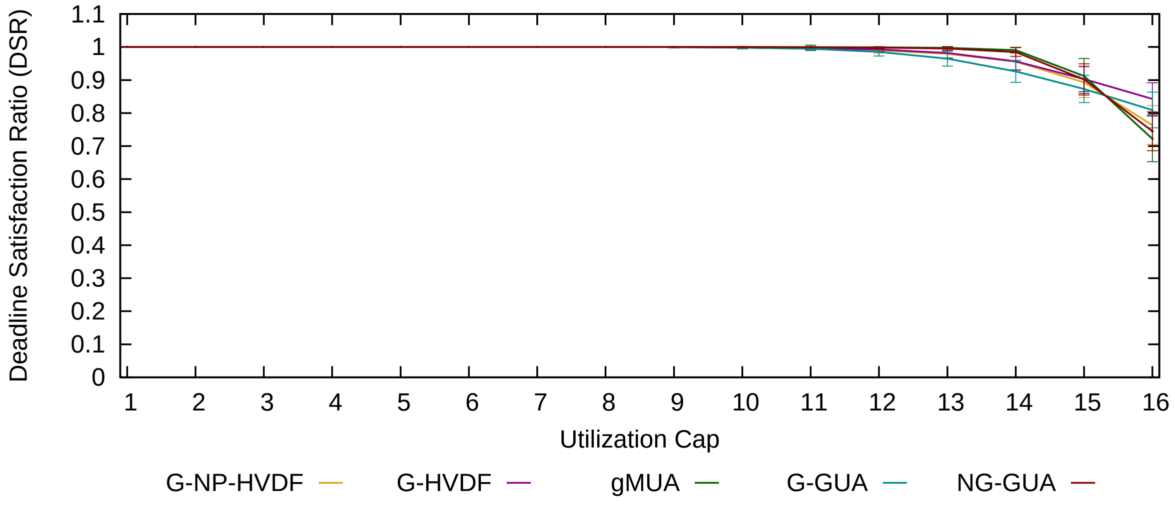


(c)

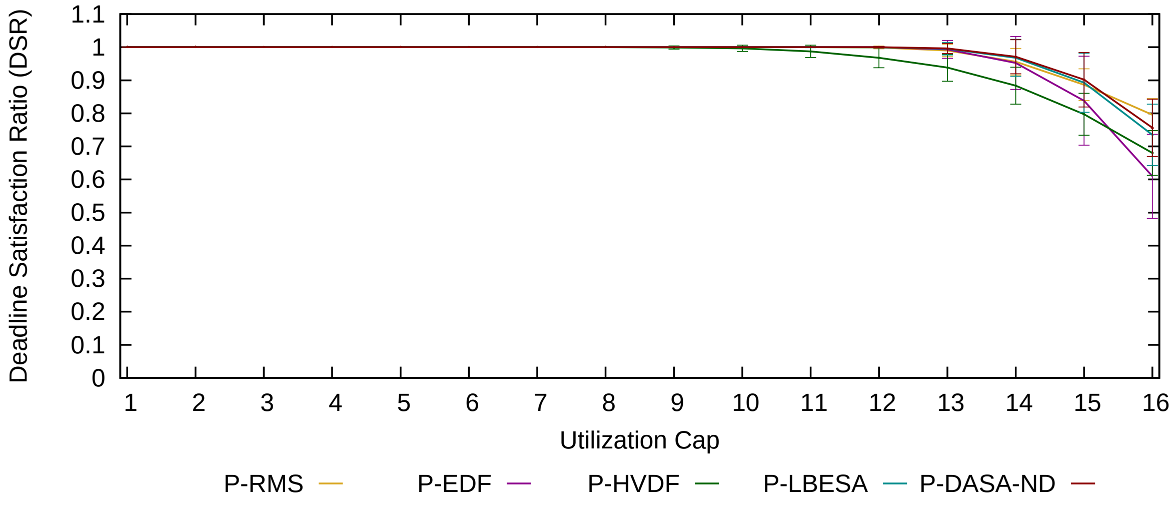
Figure B.6: 8-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

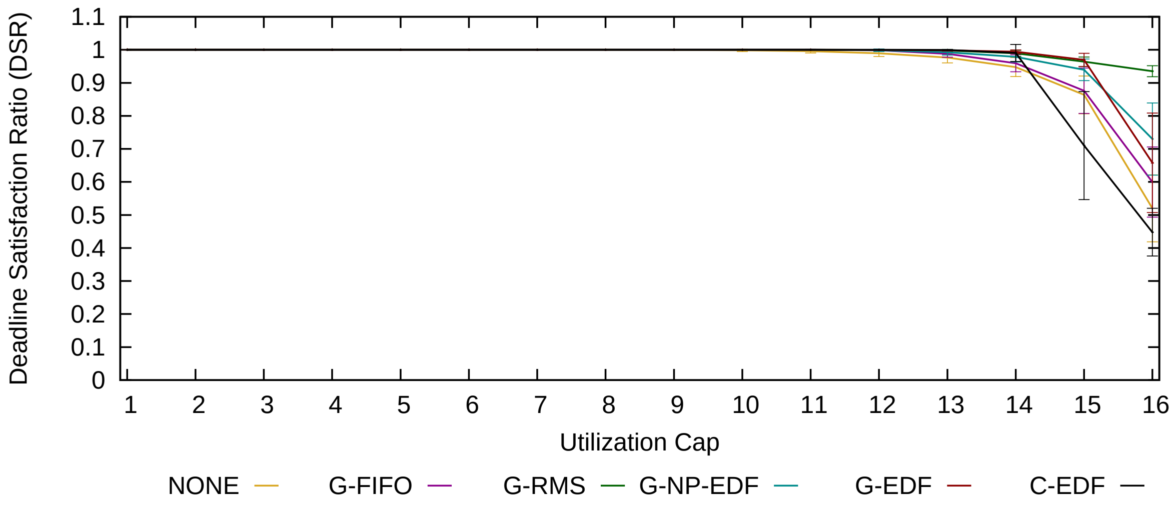


(b)

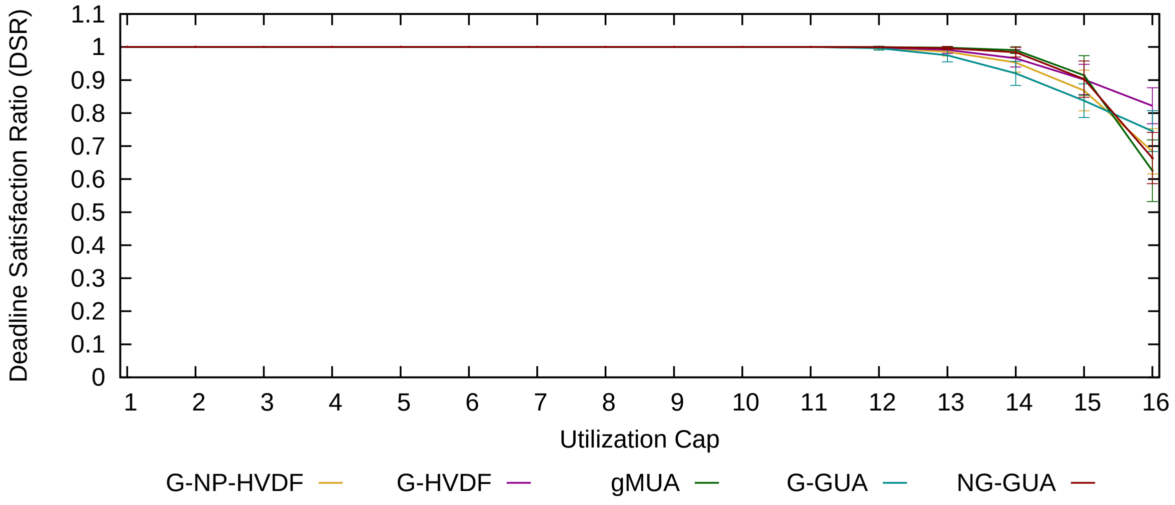


(c)

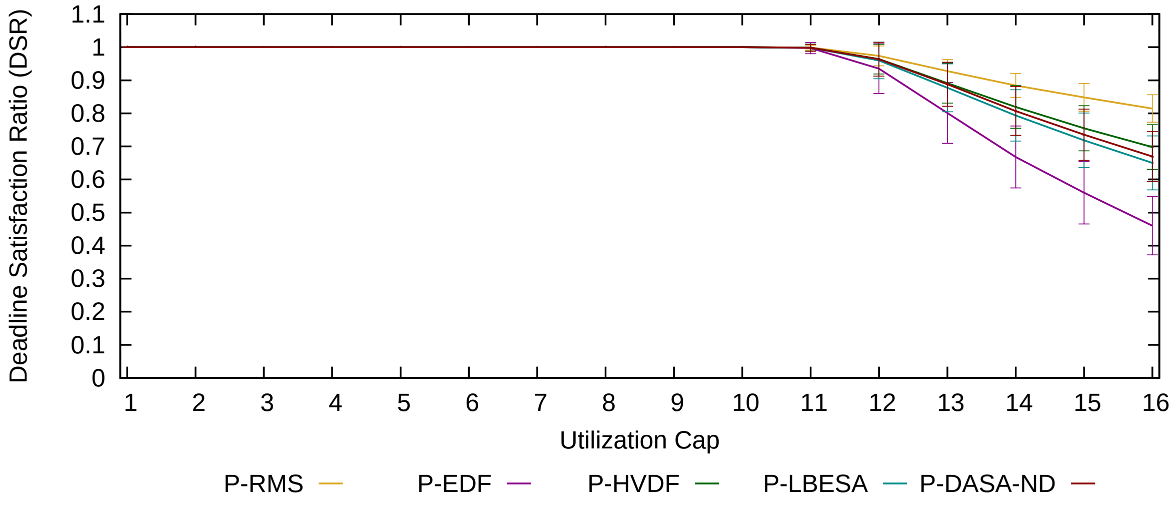
Figure B.7: 16-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

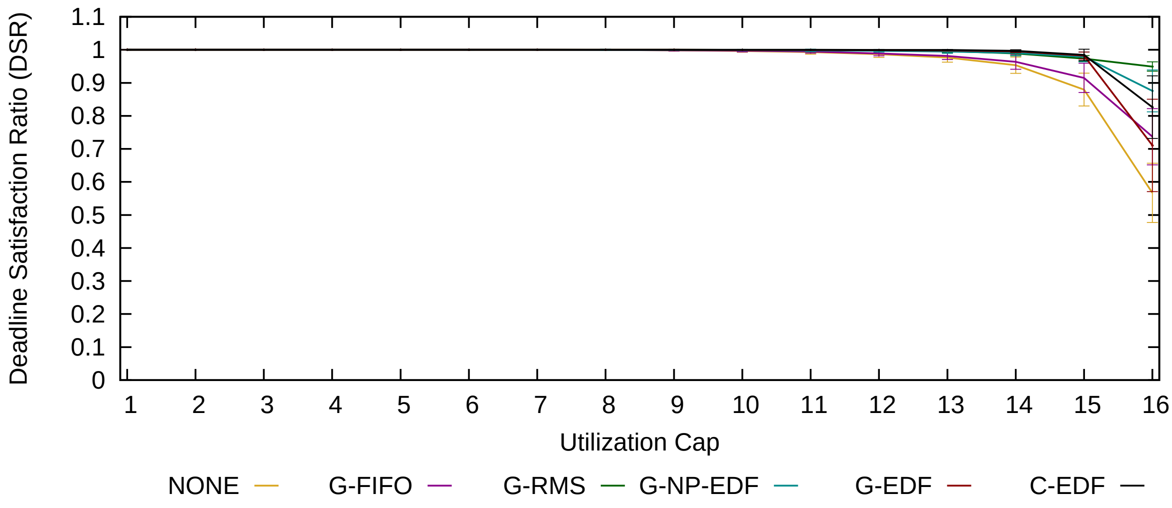


(b)

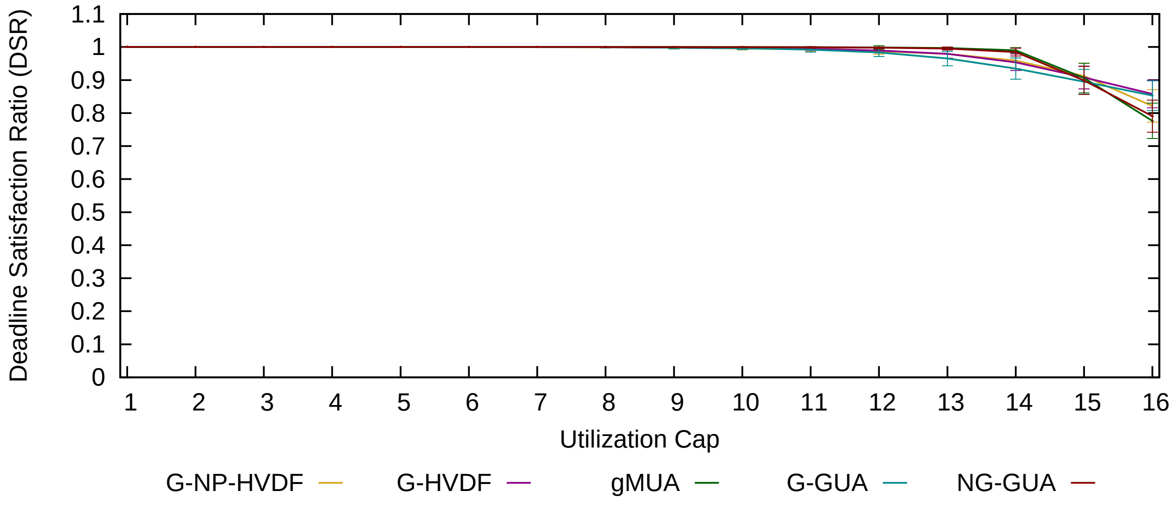


(c)

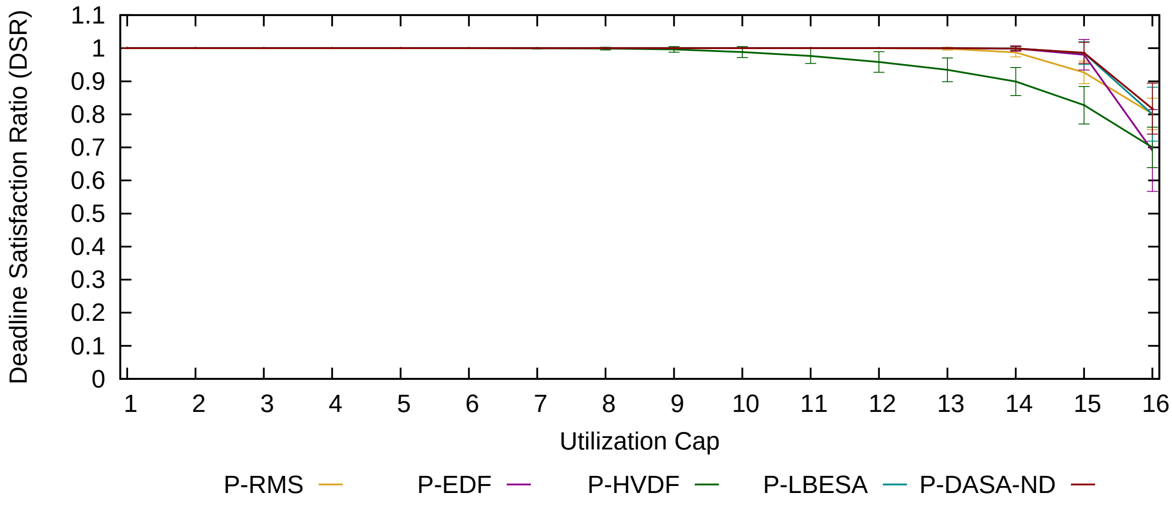
Figure B.8: 16-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

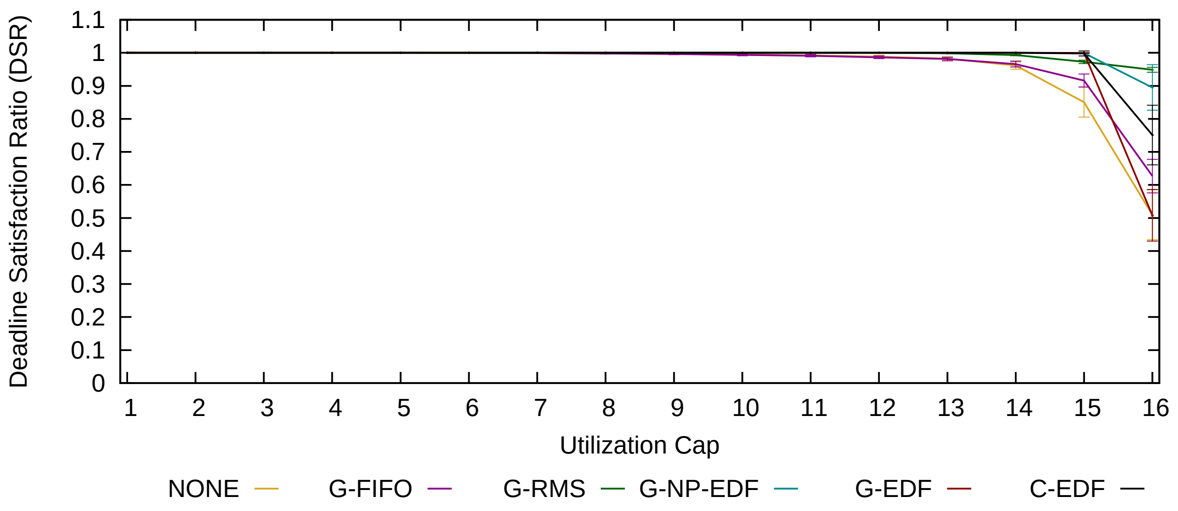


(b)

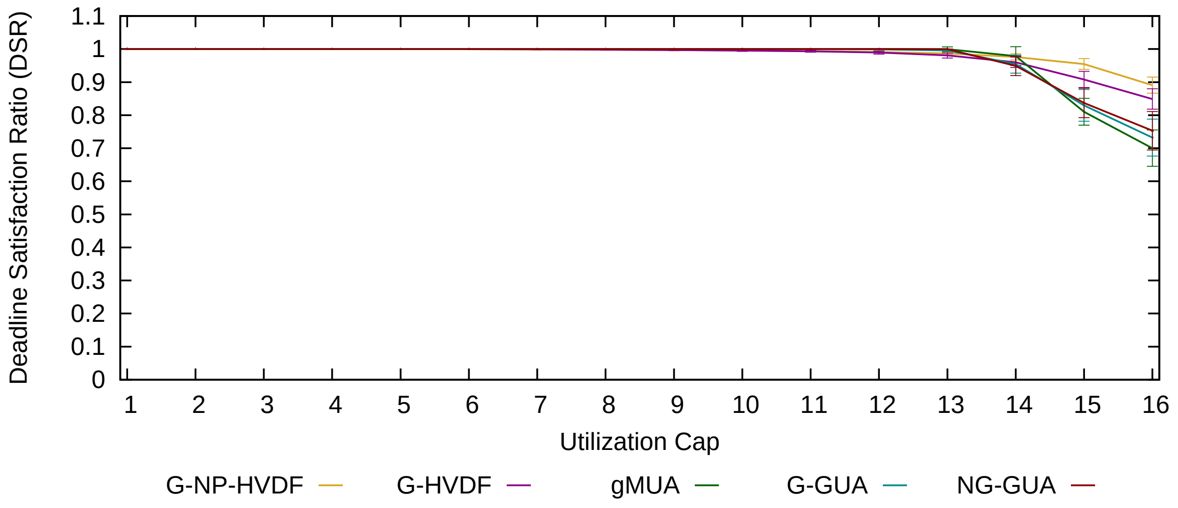


(c)

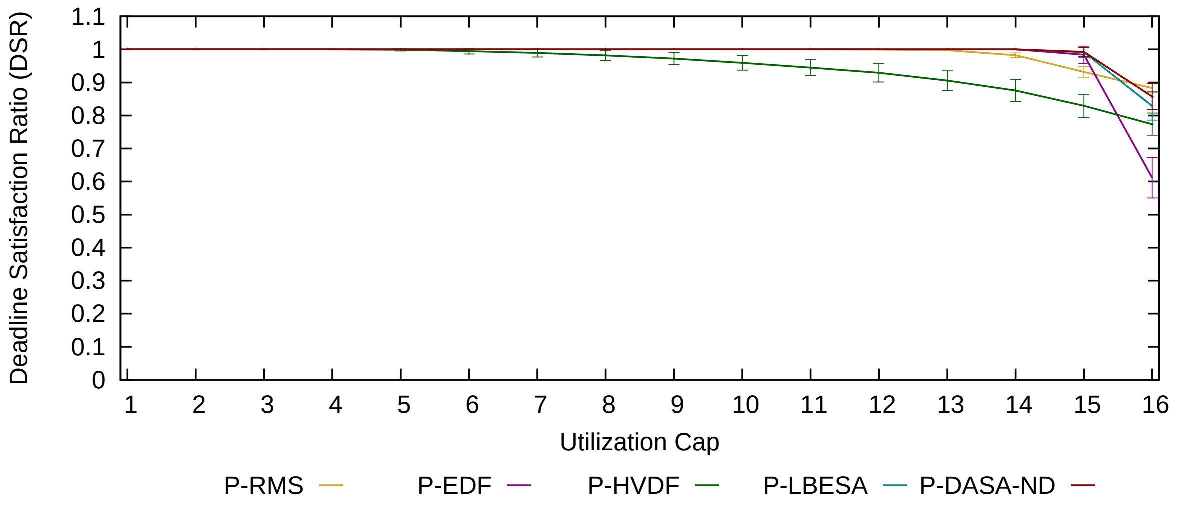
Figure B.9: 16-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

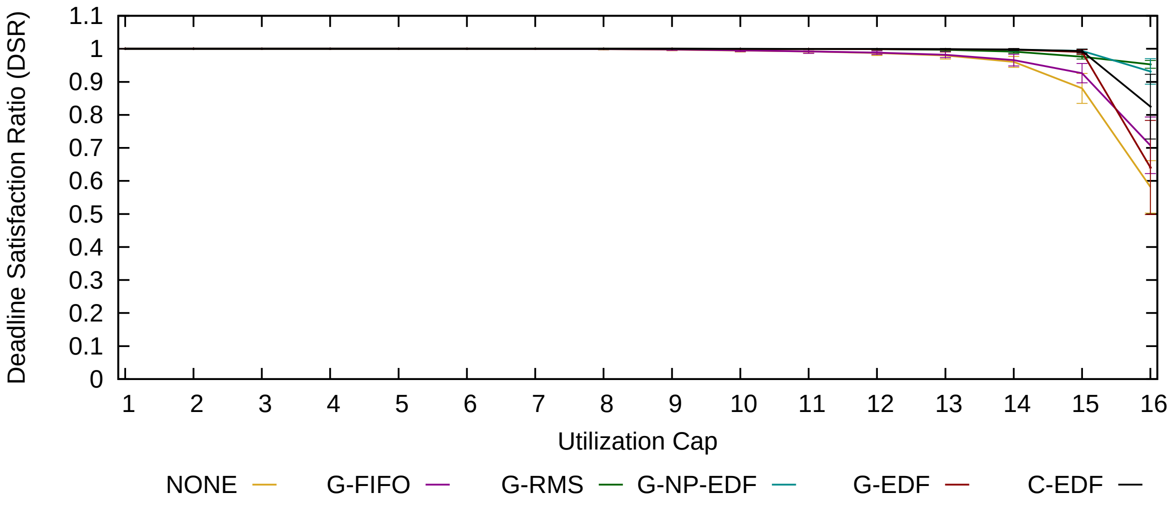


(b)

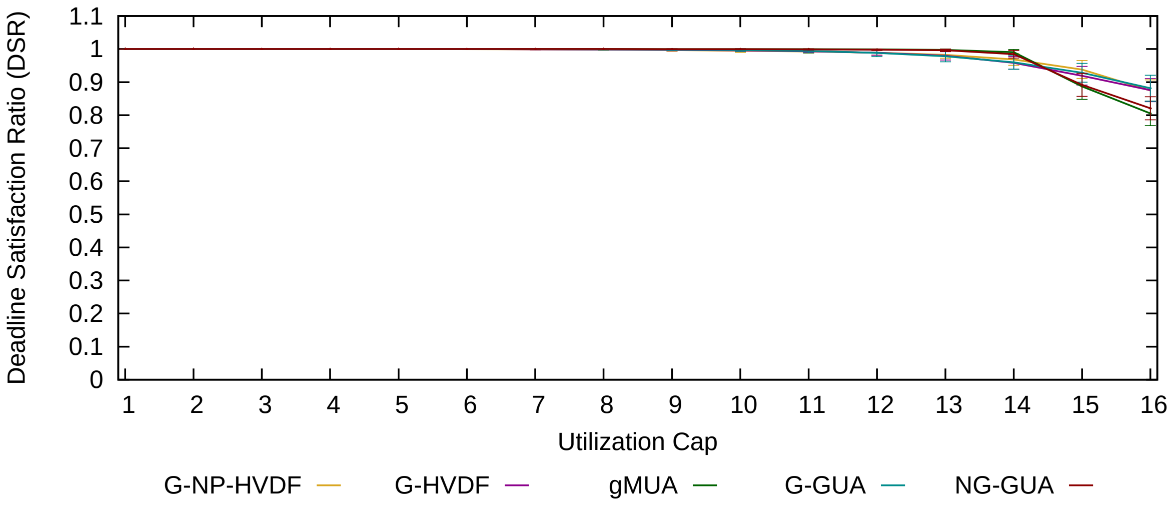


(c)

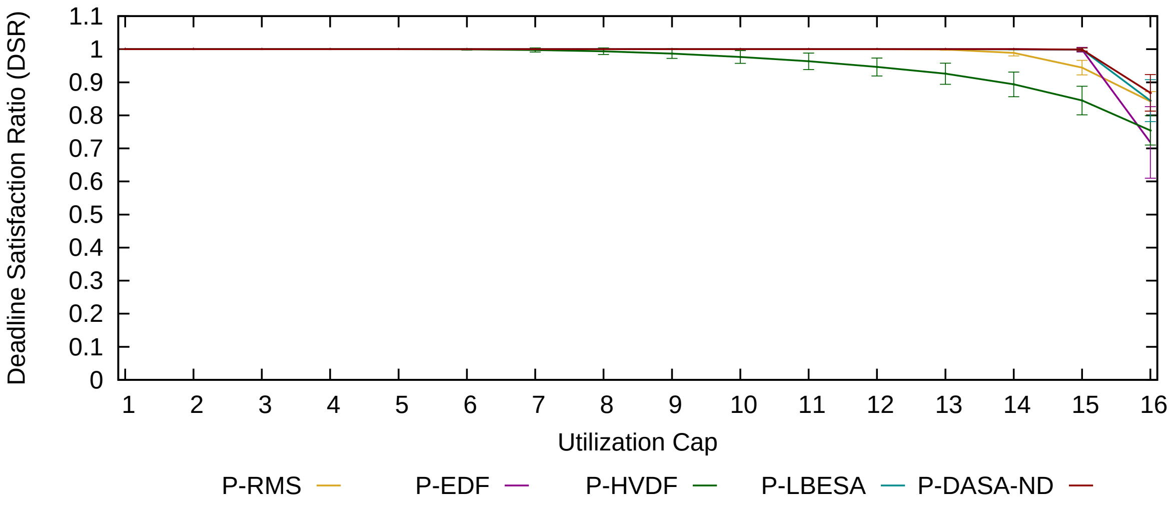
Figure B.10: 16-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

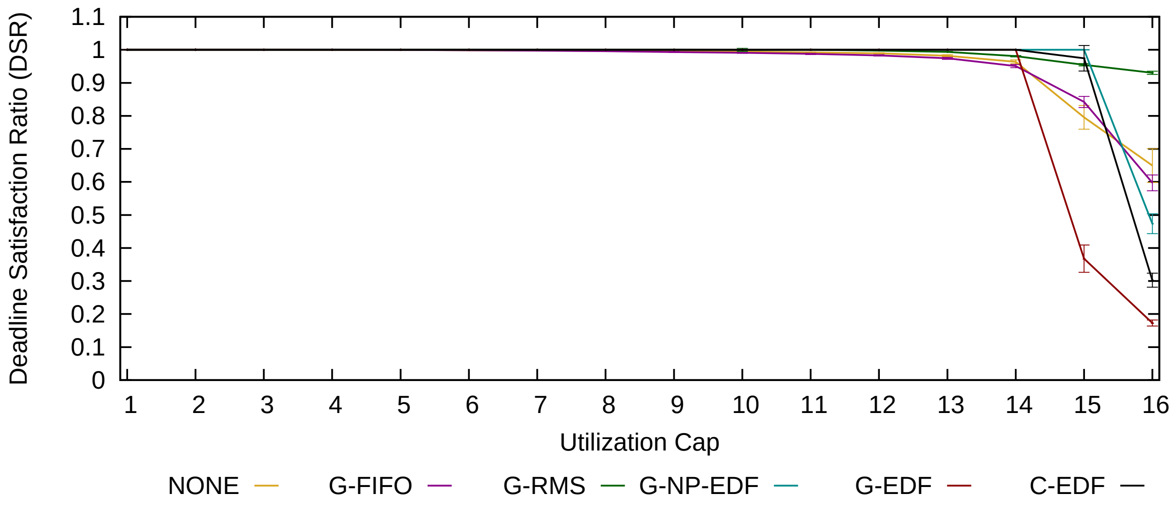


(b)

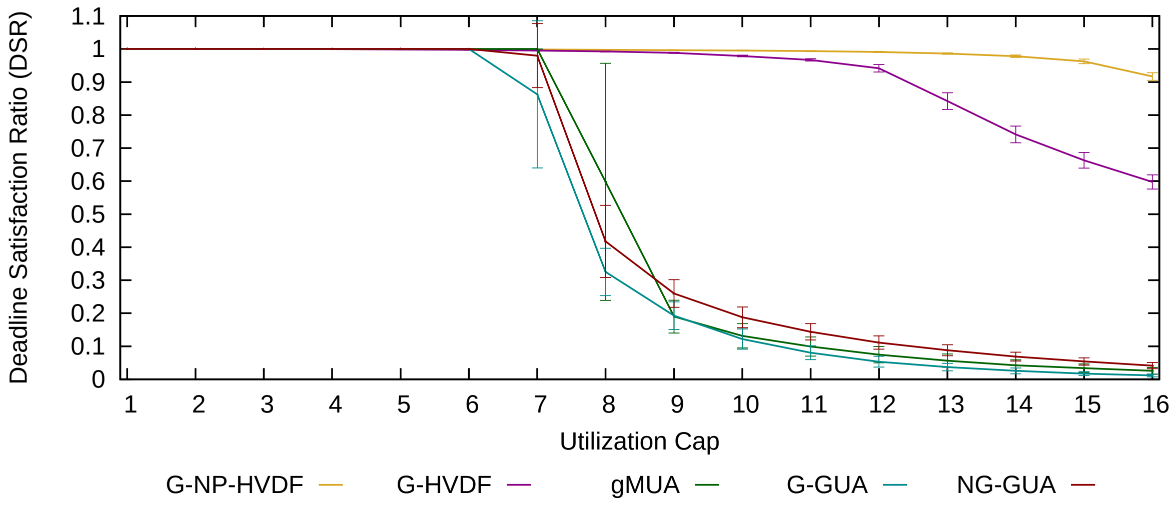


(c)

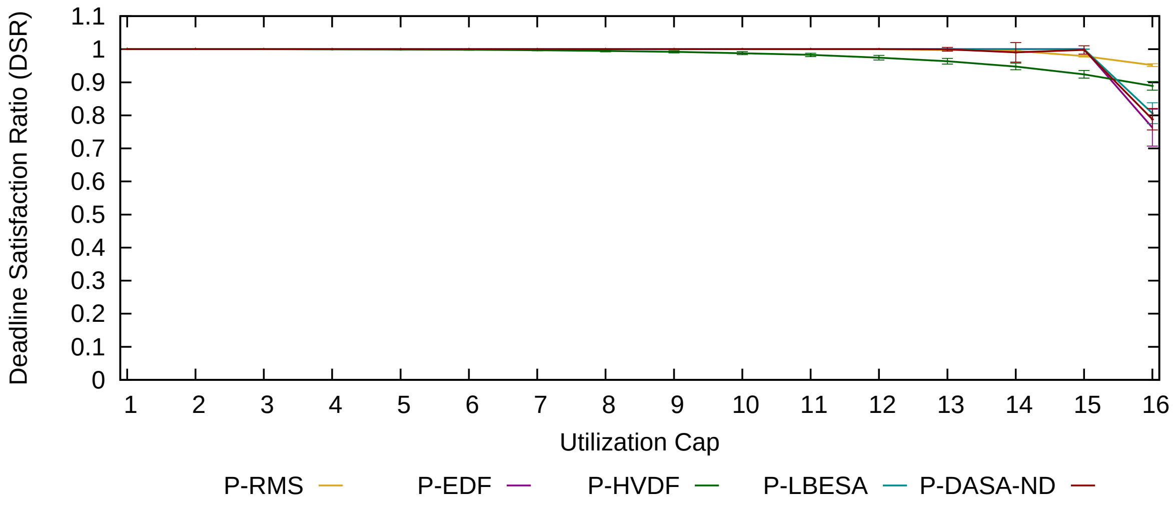
Figure B.11: 16-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

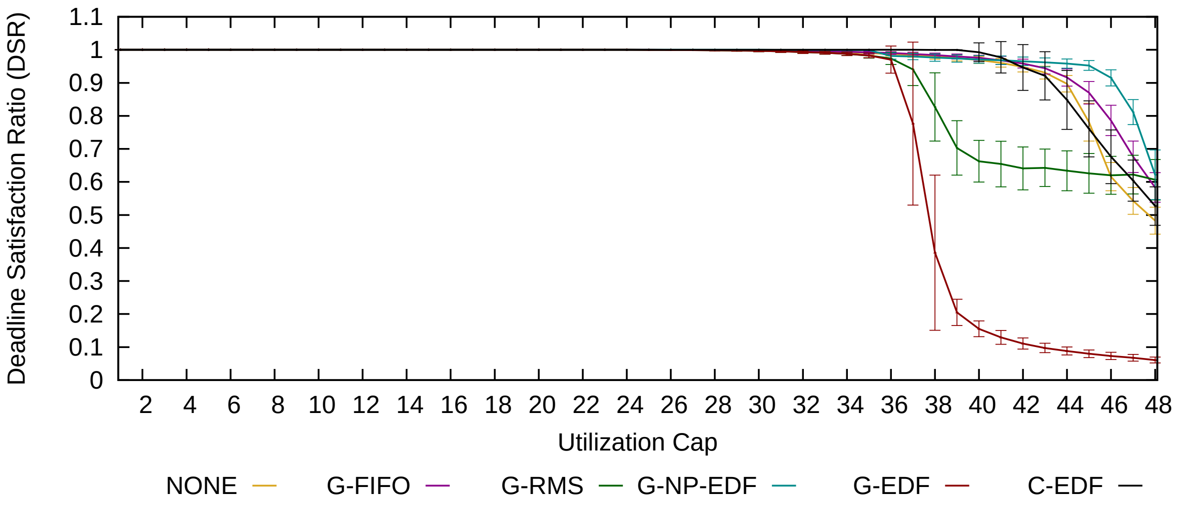


(b)

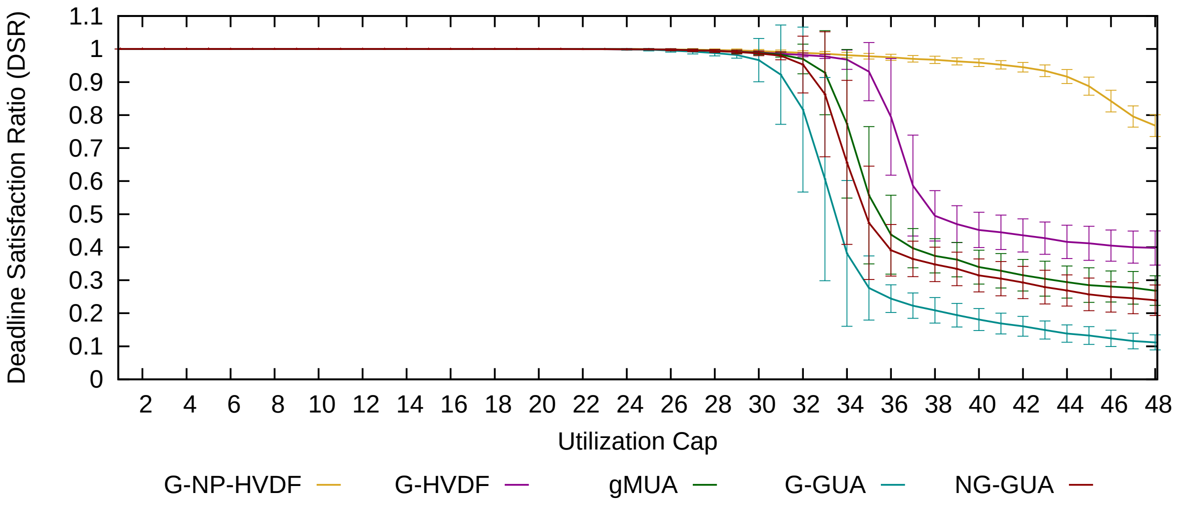


(c)

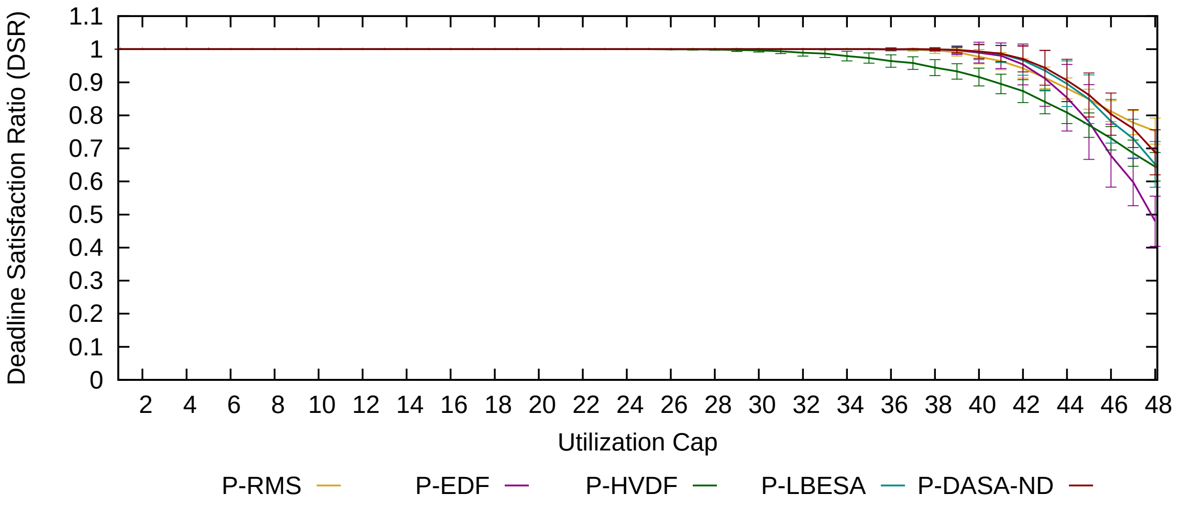
Figure B.12: 16-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

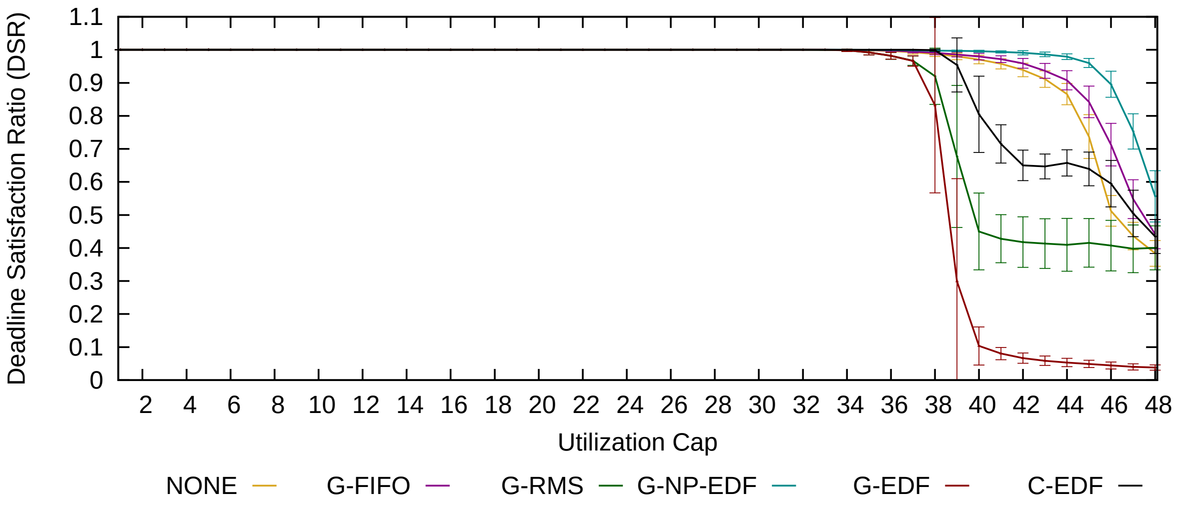


(b)

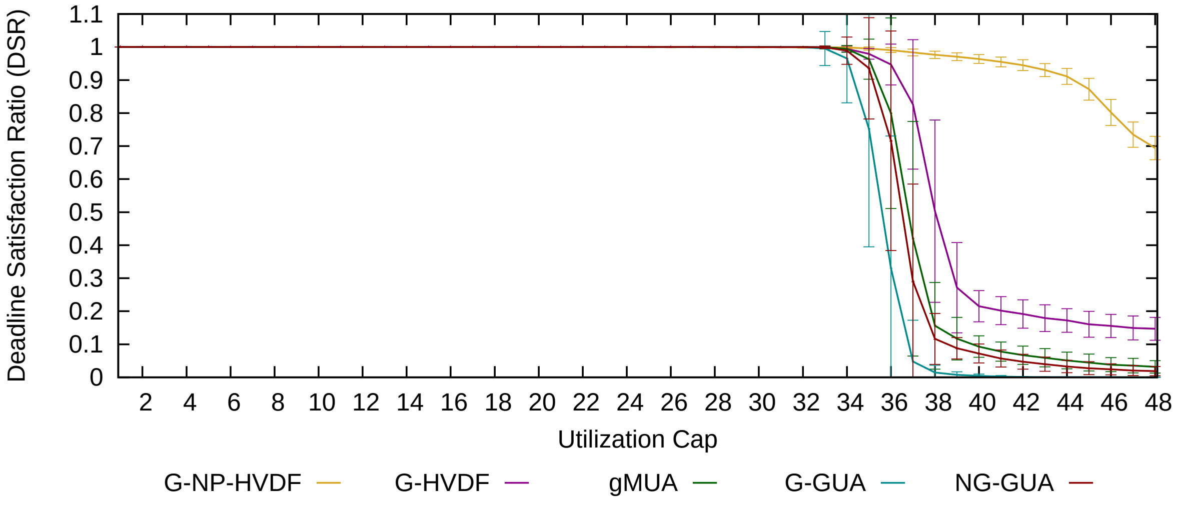


(c)

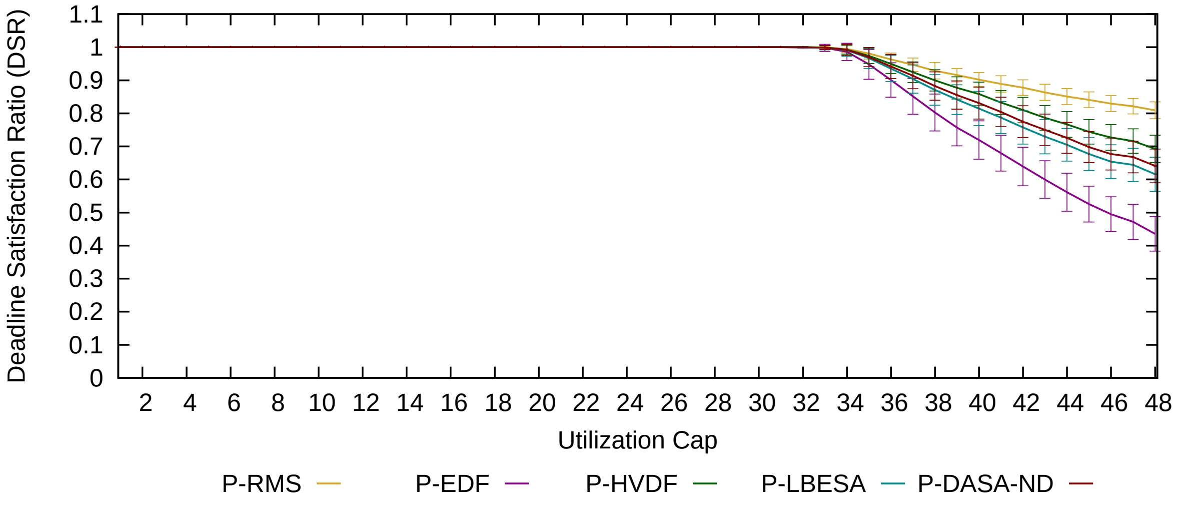
Figure B.13: 48-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

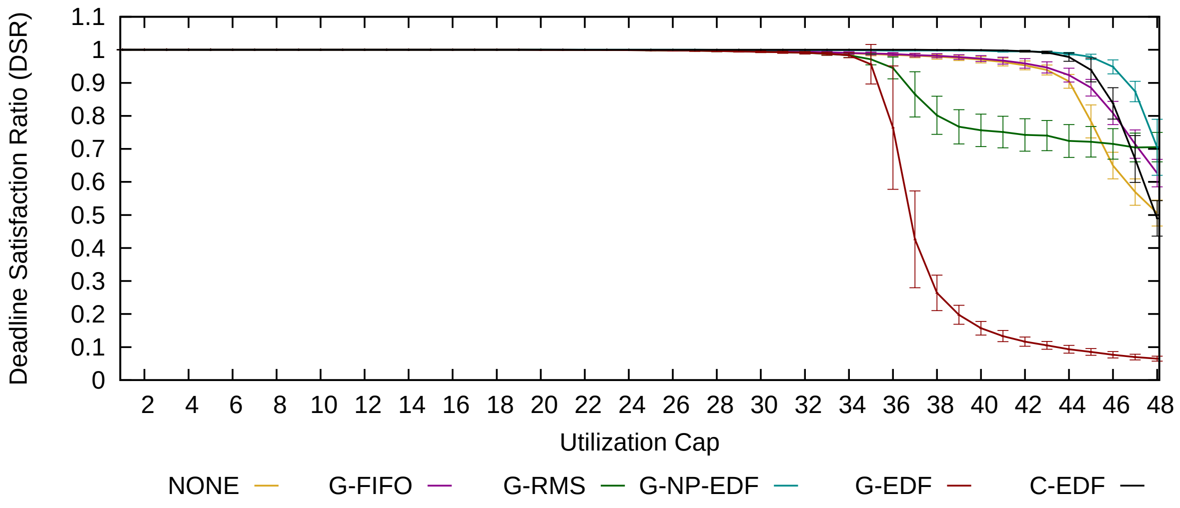


(b)

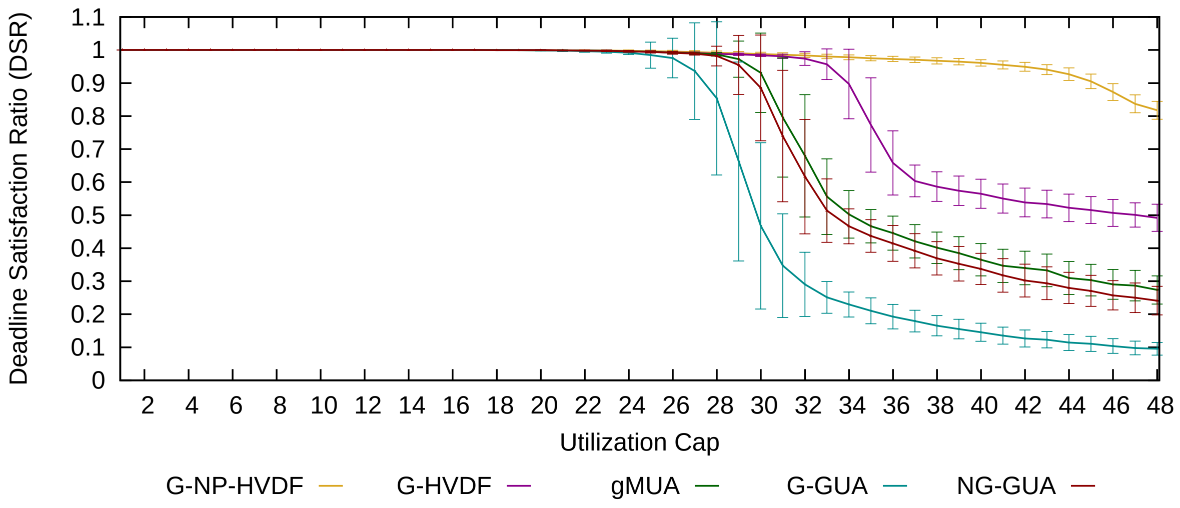


(c)

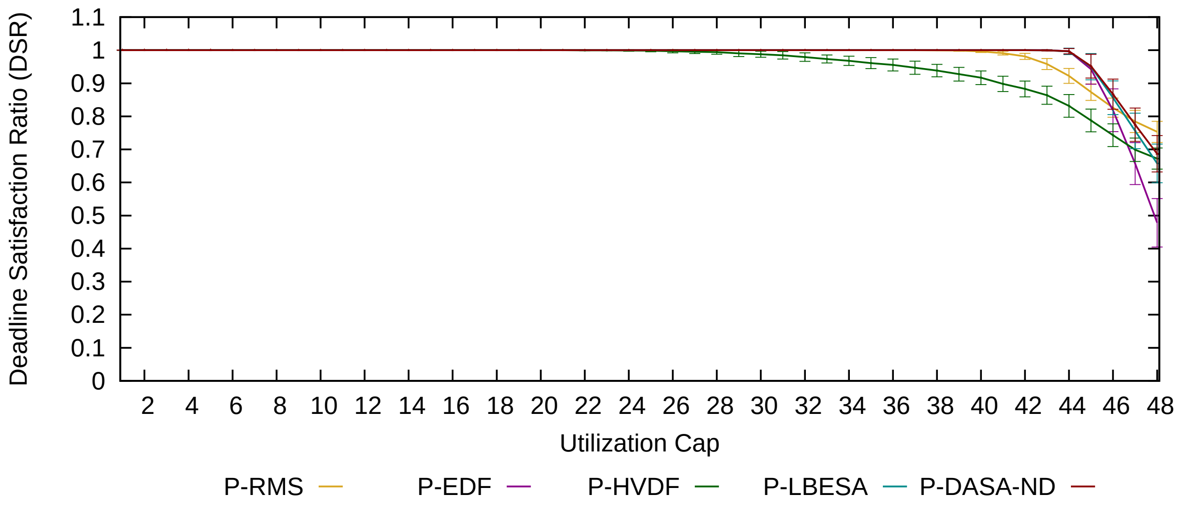
Figure B.14: 48-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

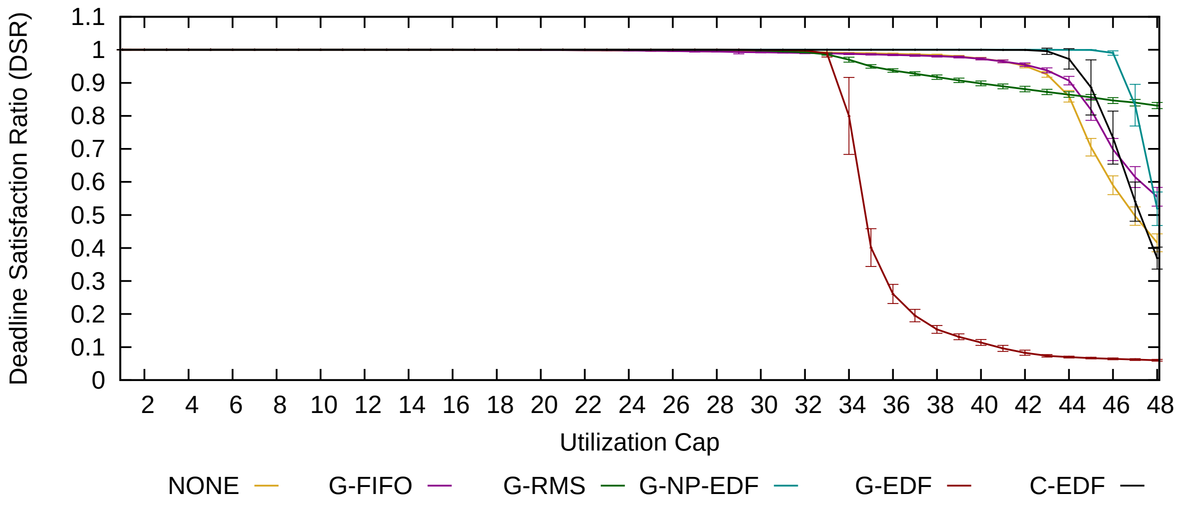


(b)

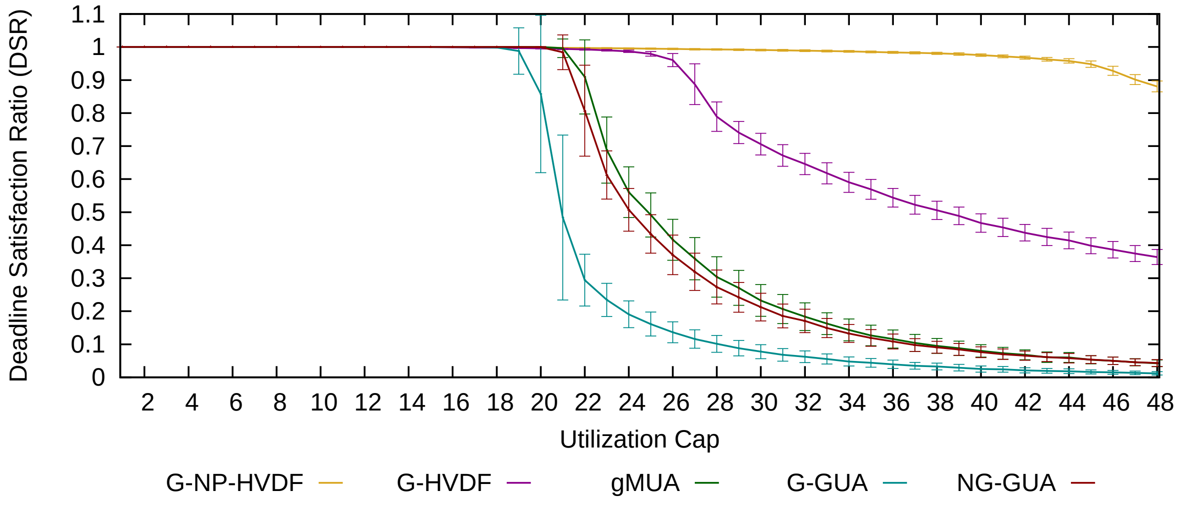


(c)

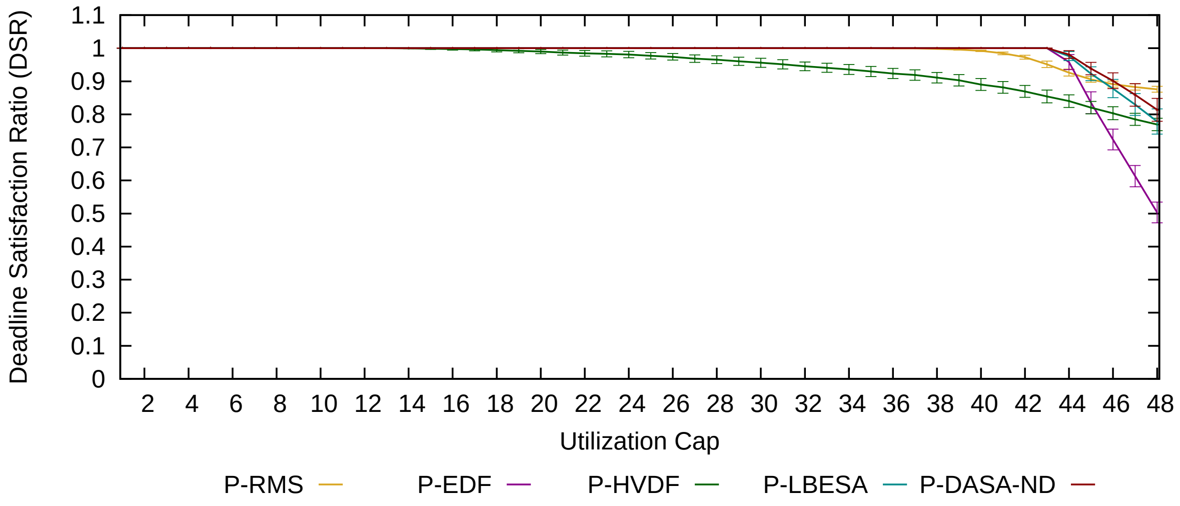
Figure B.15: 48-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

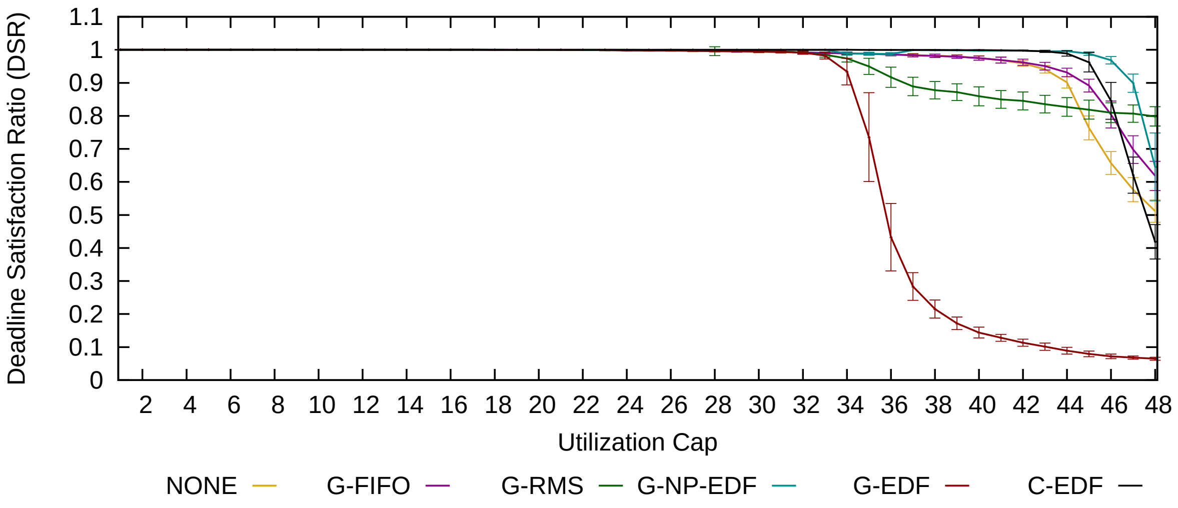


(b)

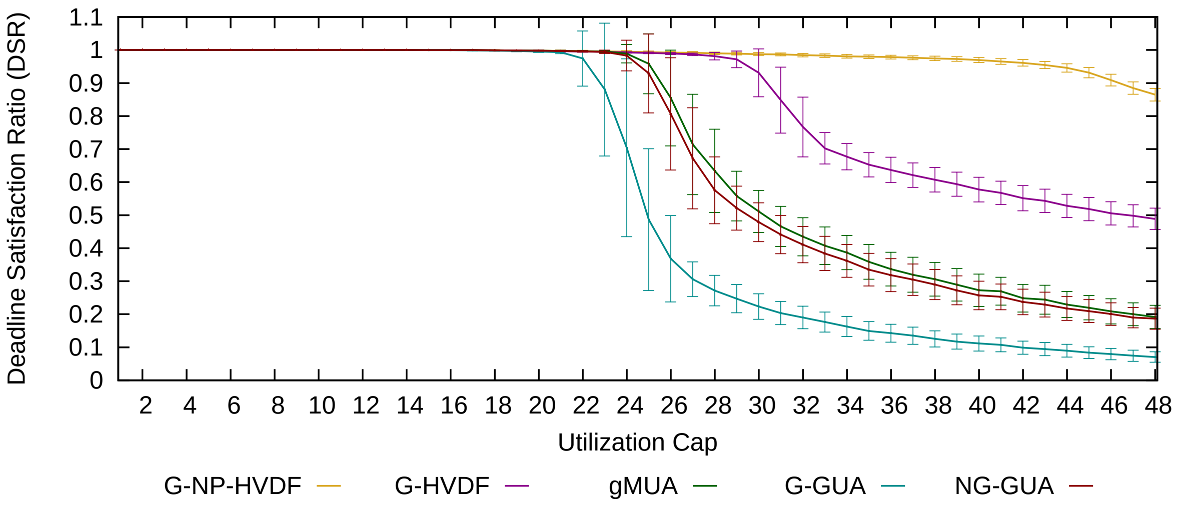


(c)

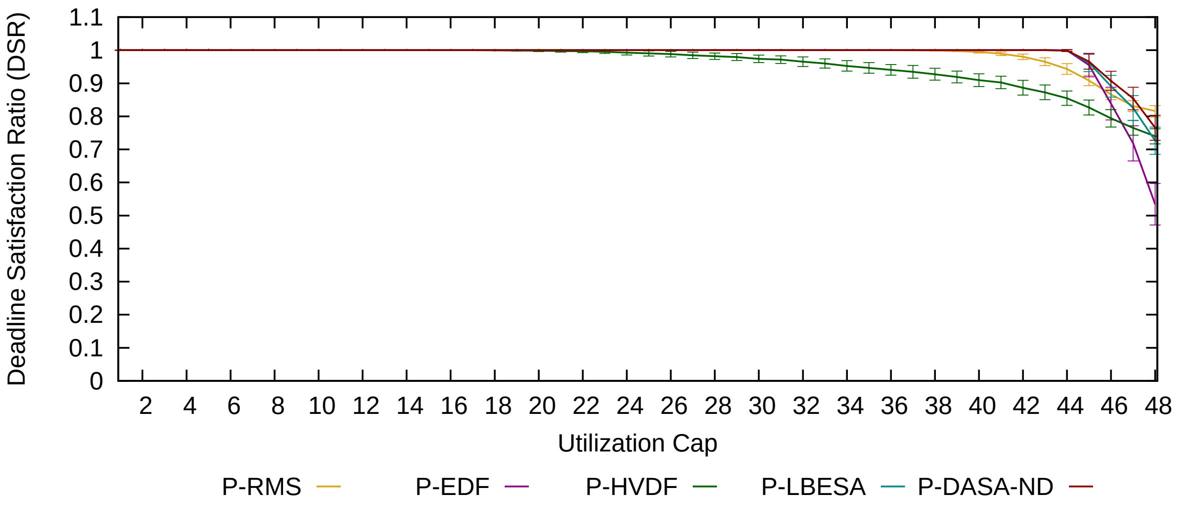
Figure B.16: 48-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium per-task weight distributions



(a)

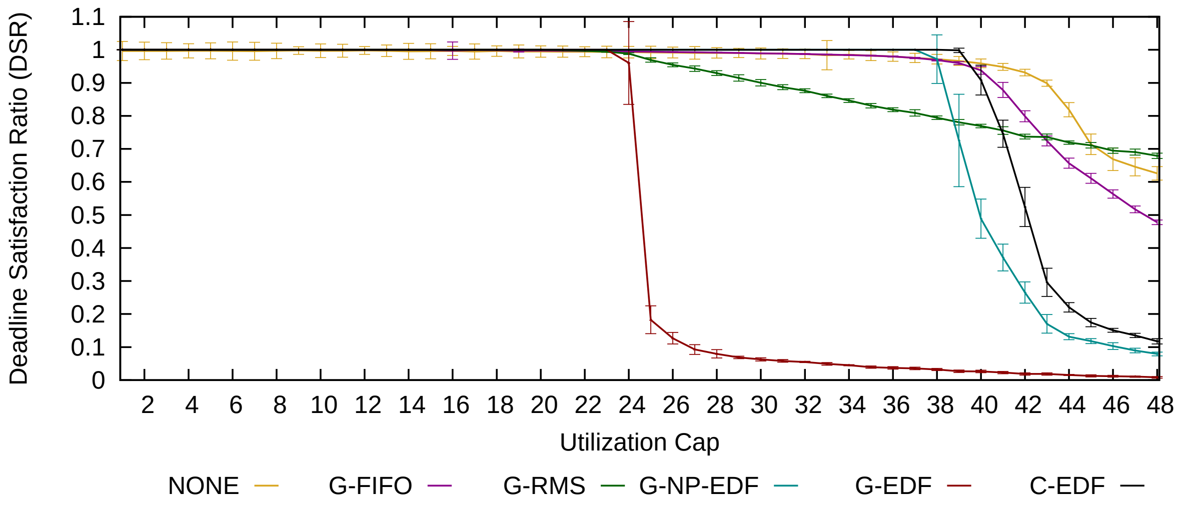


(b)

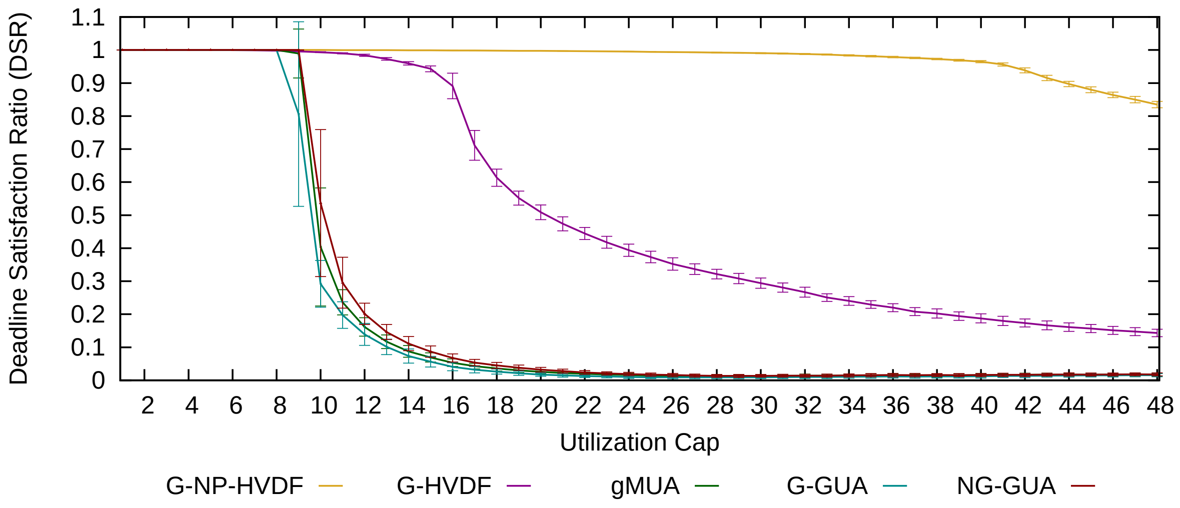


(c)

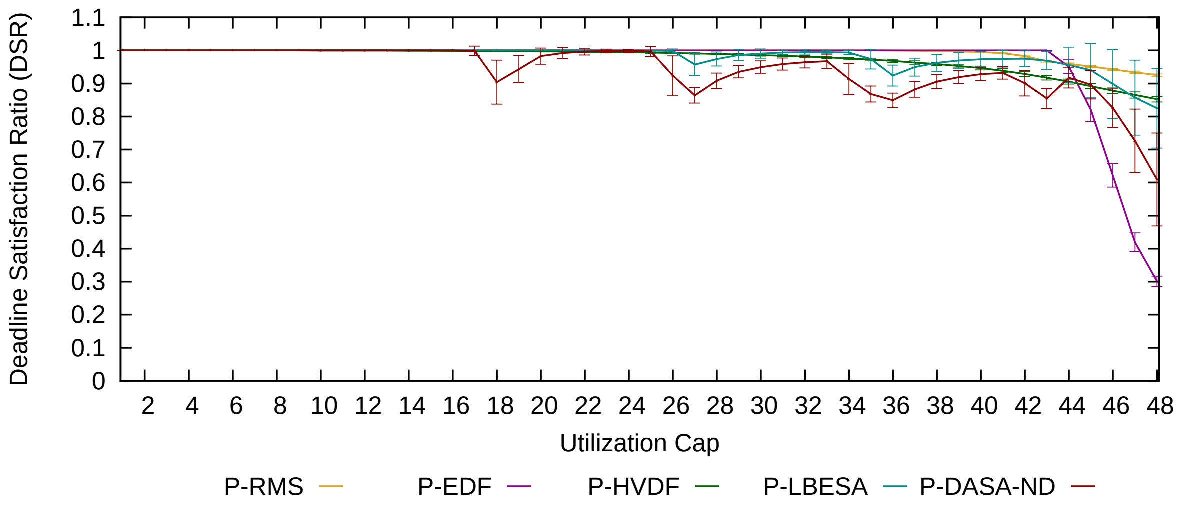
Figure B.17: 48-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)



(b)



(c)

Figure B.18: 48-Core DSR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions

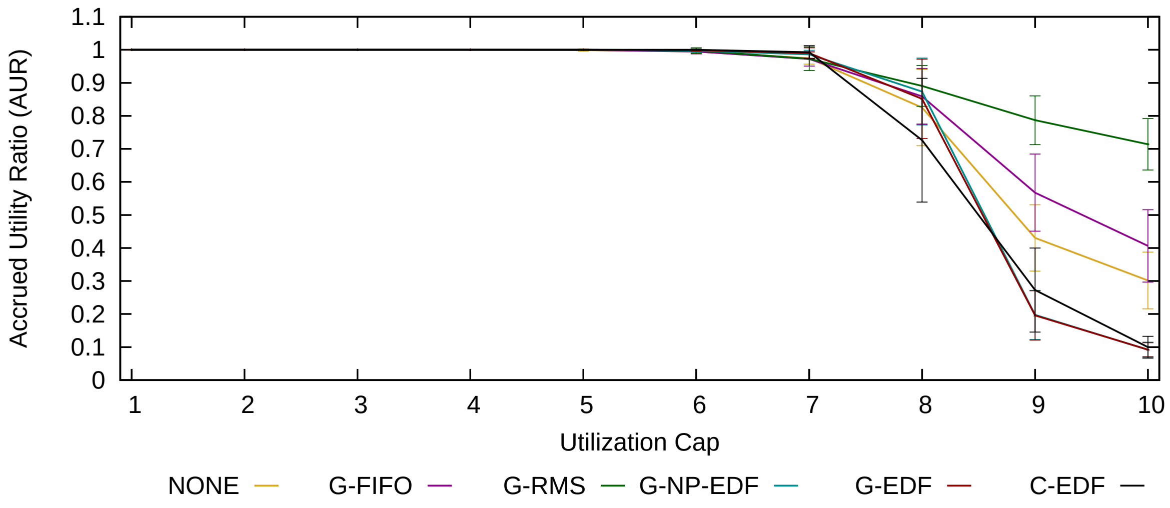
Appendix C

Complete Utility Accrual Results

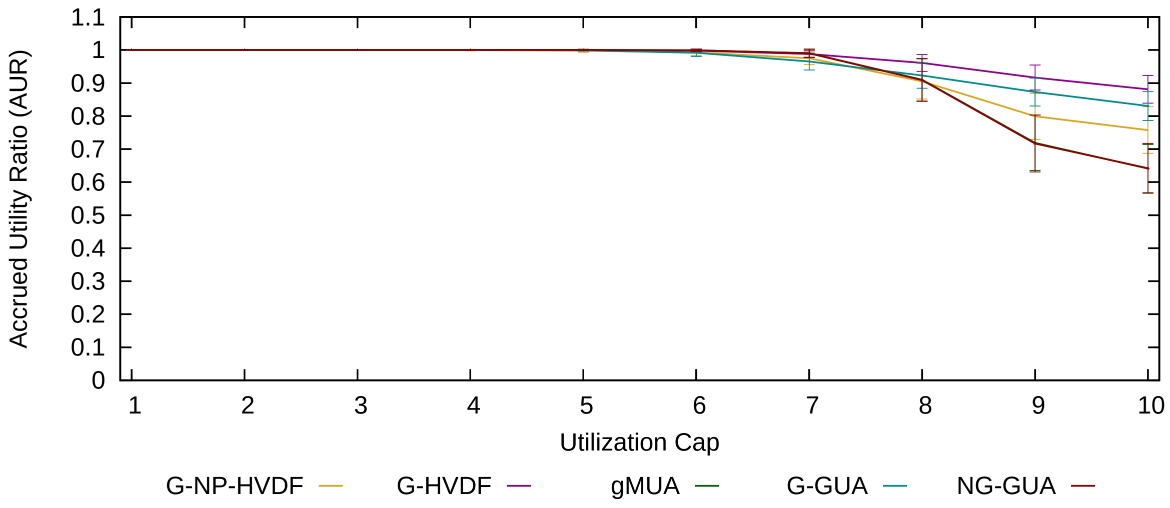
This appendix provides our complete accrued utility ratio results. It contains 36 plots grouped in 12 figures by taskset distribution and machine. It is organized as follows:

- Figure C.1 shows AUR results for our 8-core platform under heavy bimodal load
- Figure C.2 shows AUR results for our 8-core platform under heavy uniform load
- Figure C.3 shows AUR results for our 8-core platform under medium bimodal load
- Figure C.4 shows AUR results for our 8-core platform under medium uniform load
- Figure C.5 shows AUR results for our 8-core platform under light bimodal load
- Figure C.6 shows AUR results for our 8-core platform under light uniform load
- Figure C.7 shows AUR results for our 16-core platform under heavy bimodal load
- Figure C.8 shows AUR results for our 16-core platform under heavy uniform load
- Figure C.9 shows AUR results for our 16-core platform under medium bimodal load
- Figure C.10 shows AUR results for our 16-core platform under medium uniform load
- Figure C.11 shows AUR results for our 16-core platform under light bimodal load
- Figure C.12 shows AUR results for our 16-core platform under light uniform load
- Figure C.13 shows AUR results for our 48-core platform under heavy bimodal load
- Figure C.14 shows AUR results for our 48-core platform under heavy uniform load
- Figure C.15 shows AUR results for our 48-core platform under medium bimodal load

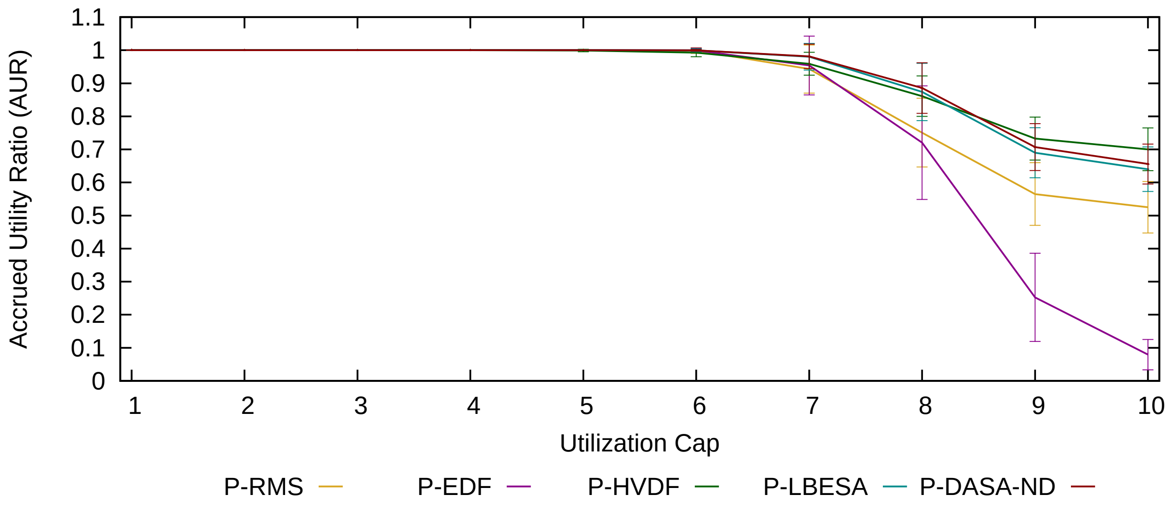
- Figure C.16 shows AUR results for our 48-core platform under medium uniform load
- Figure C.17 shows AUR results for our 48-core platform under light bimodal load
- Figure C.18 shows AUR results for our 48-core platform under light uniform load



(a)

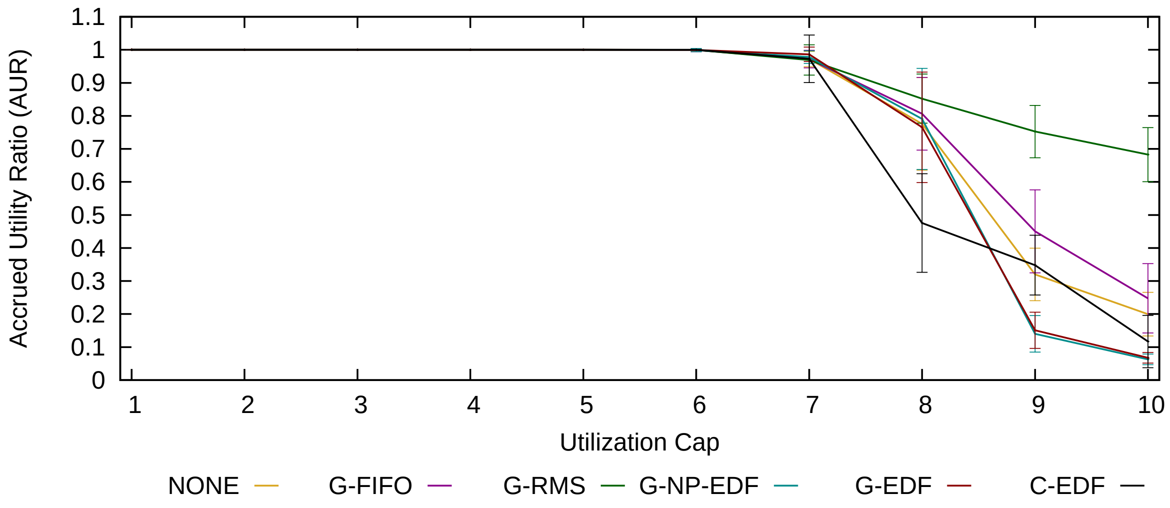


(b)

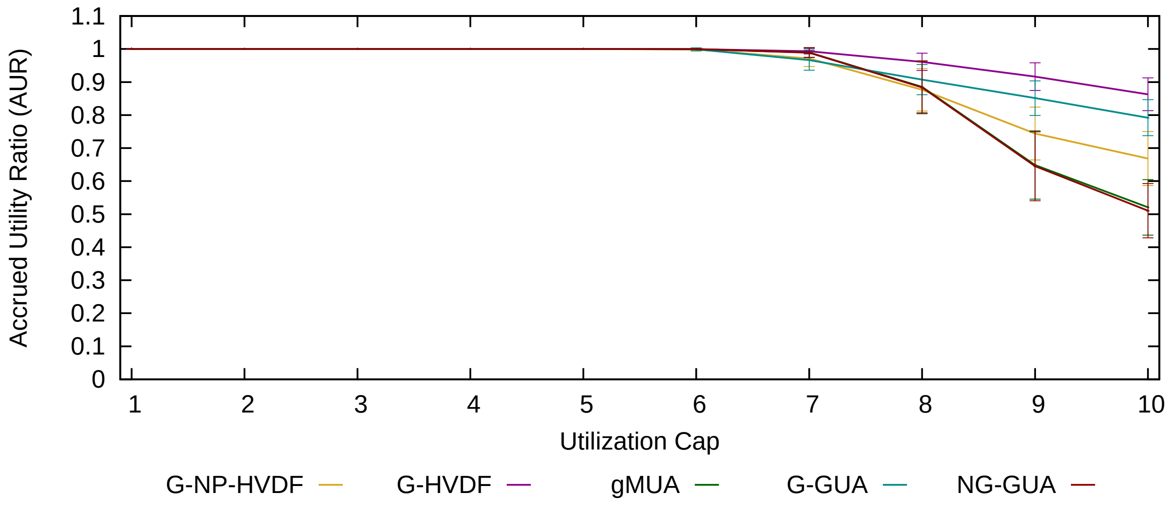


(c)

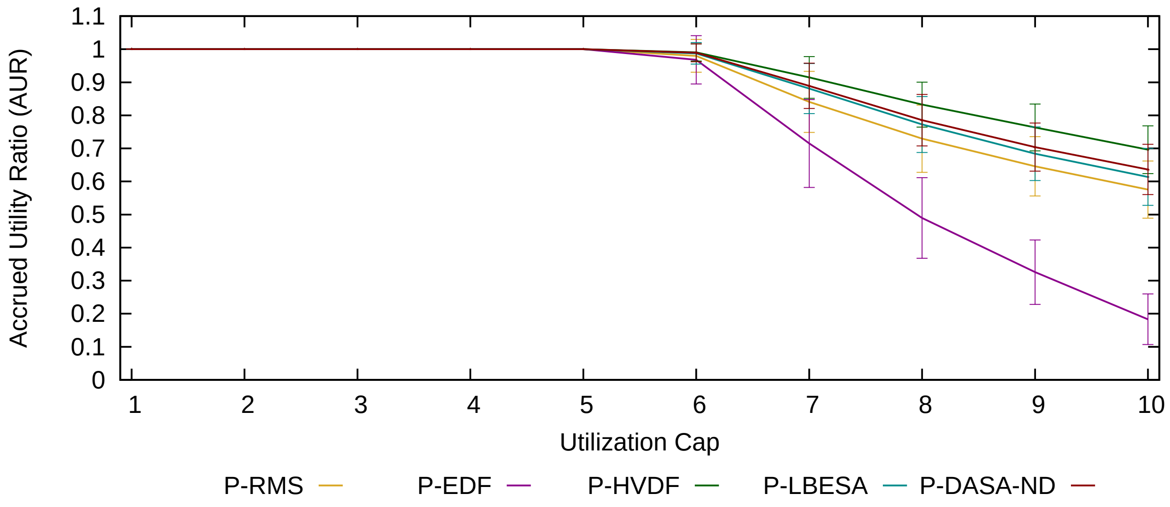
Figure C.1: 8-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

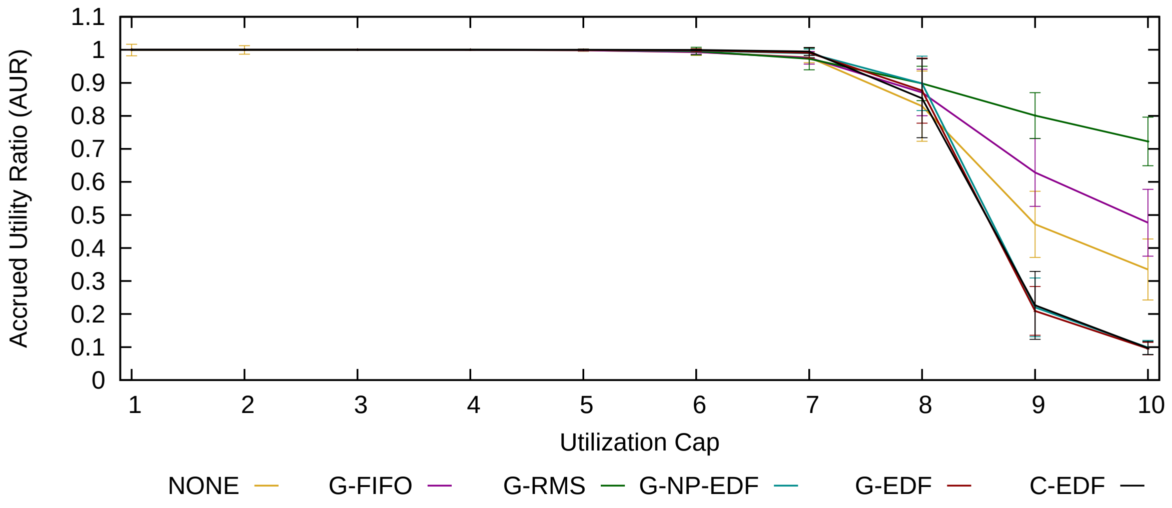


(b)

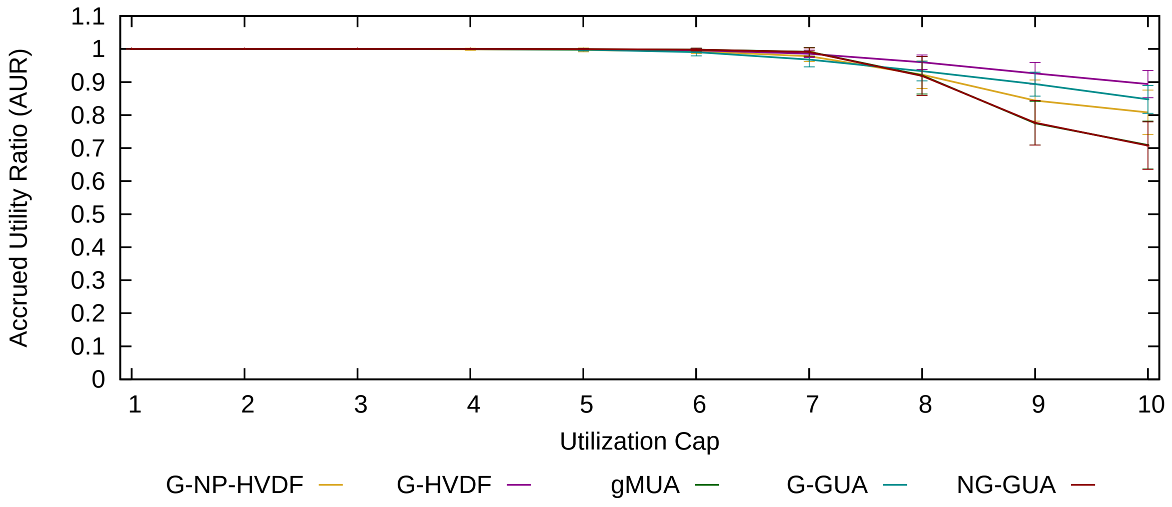


(c)

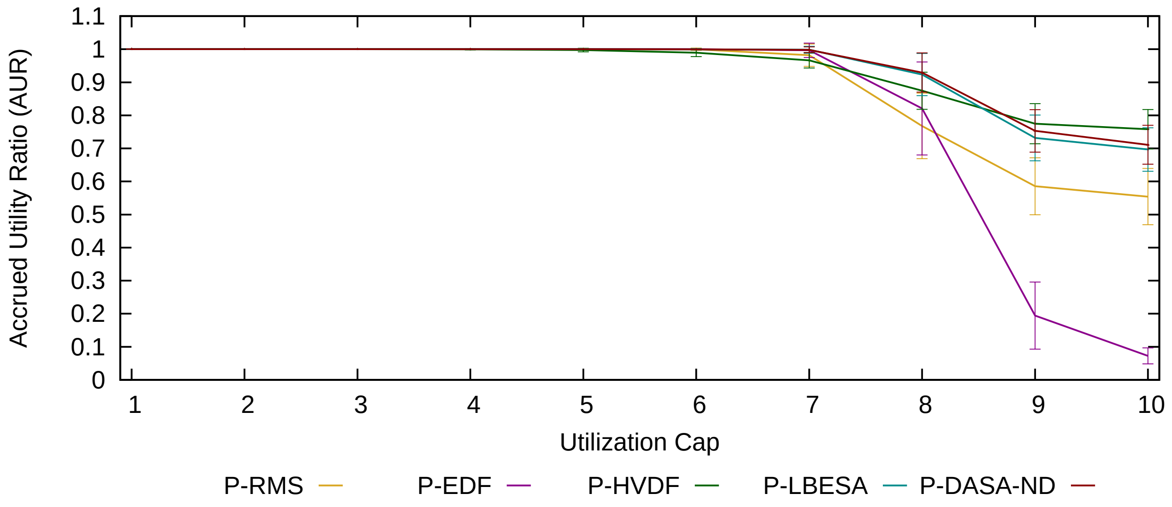
Figure C.2: 8-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

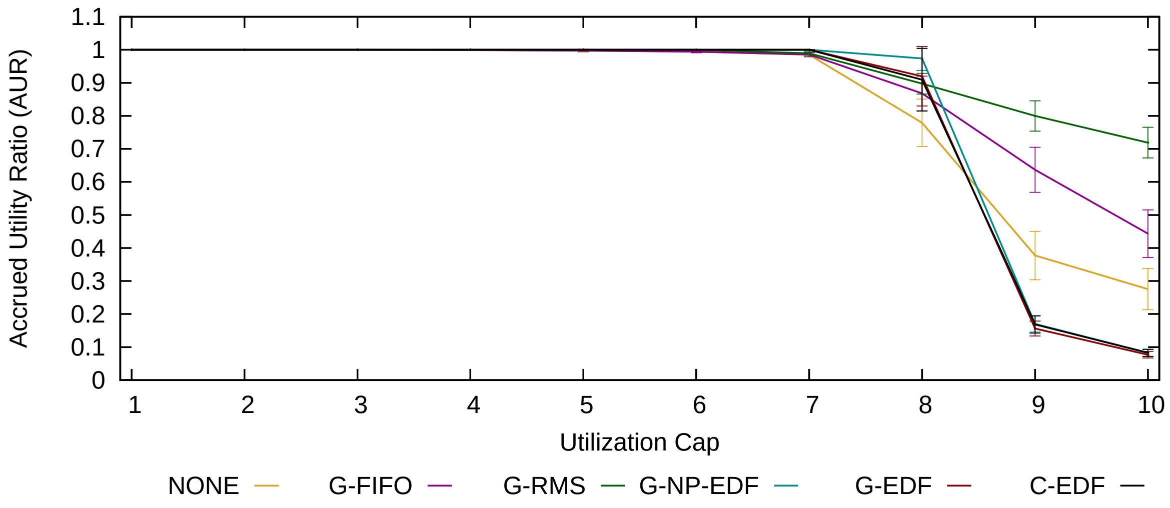


(b)

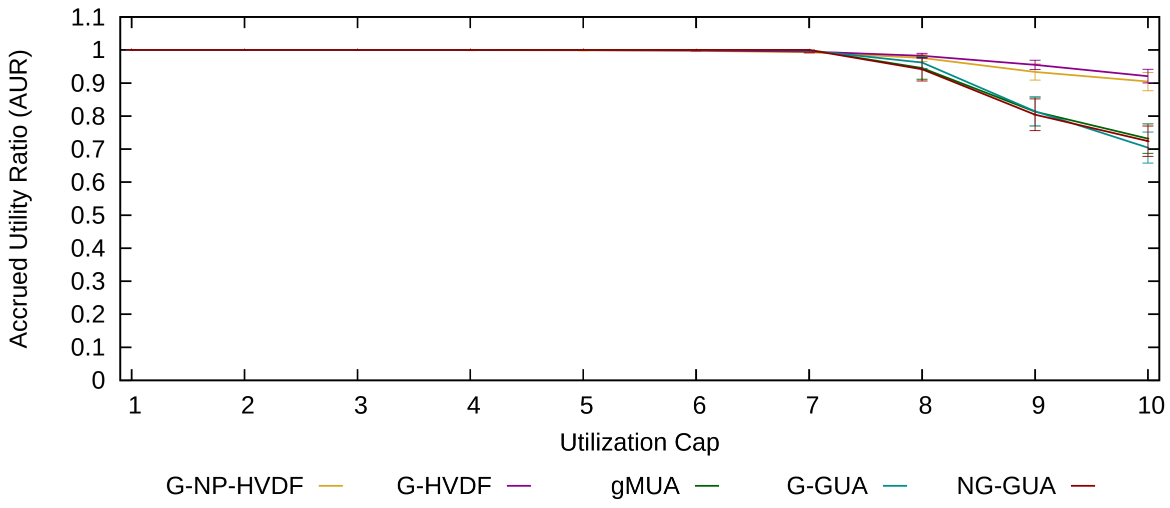


(c)

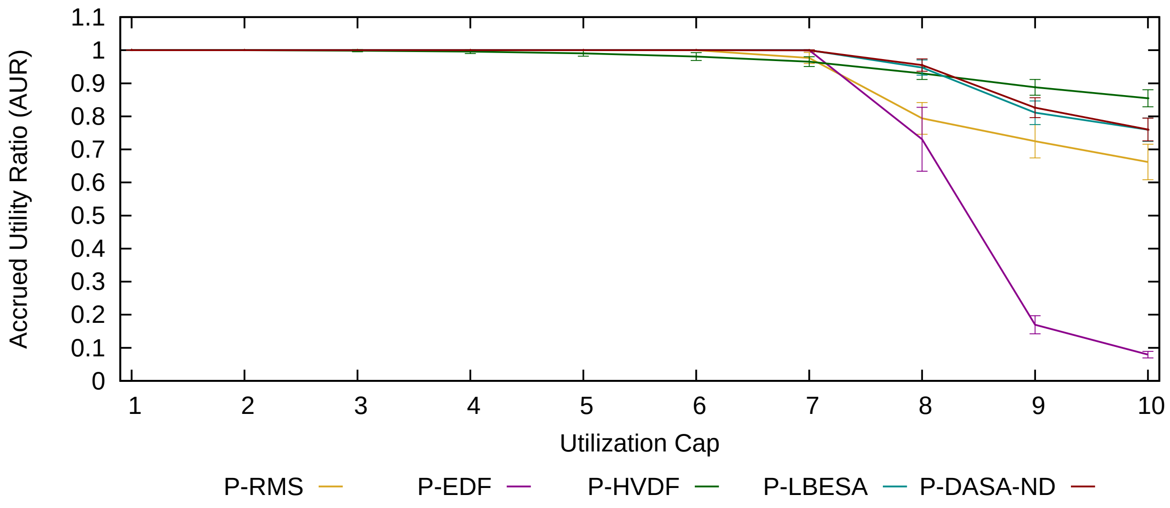
Figure C.3: 8-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

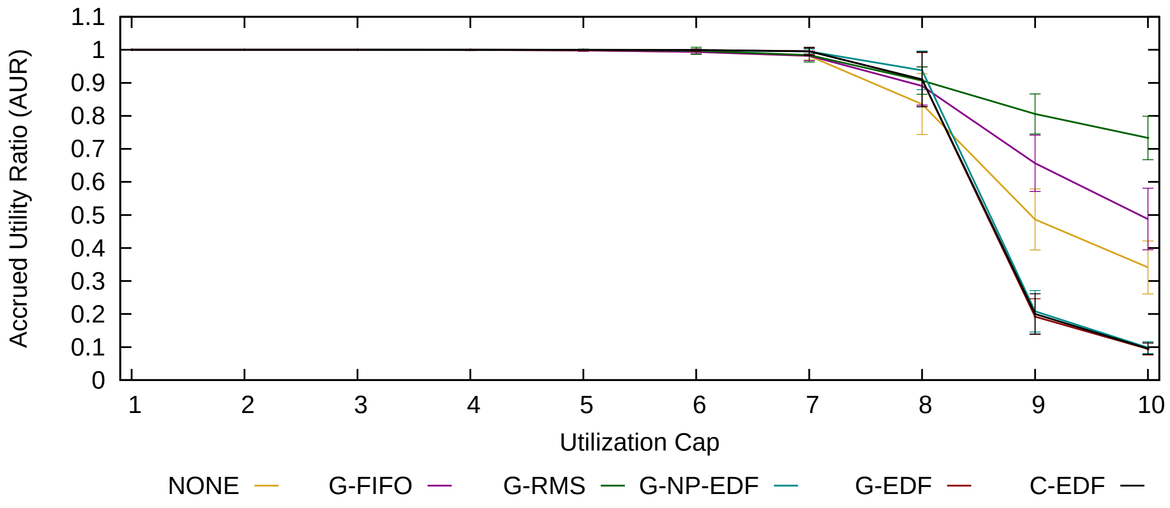


(b)

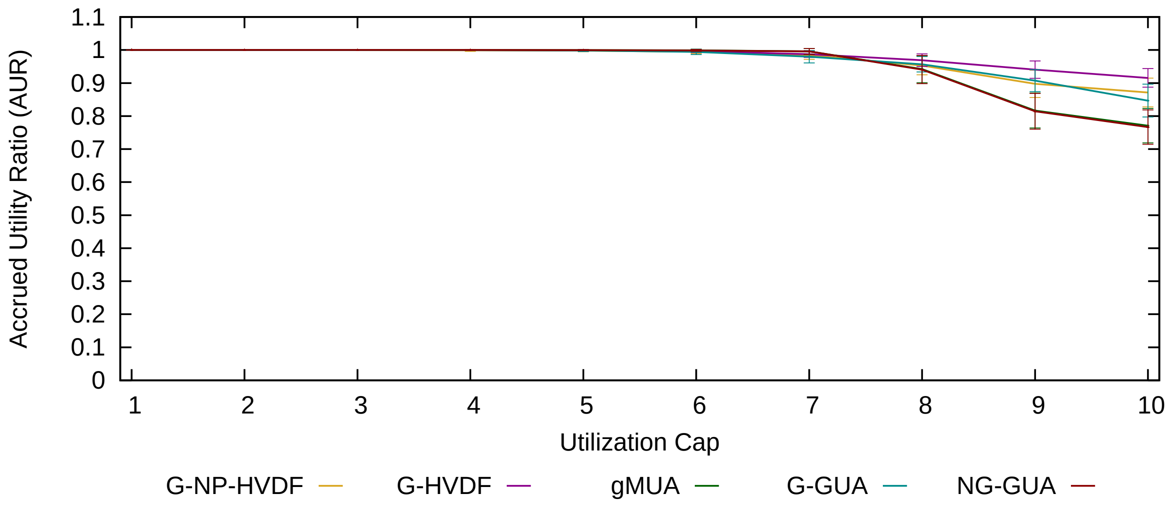


(c)

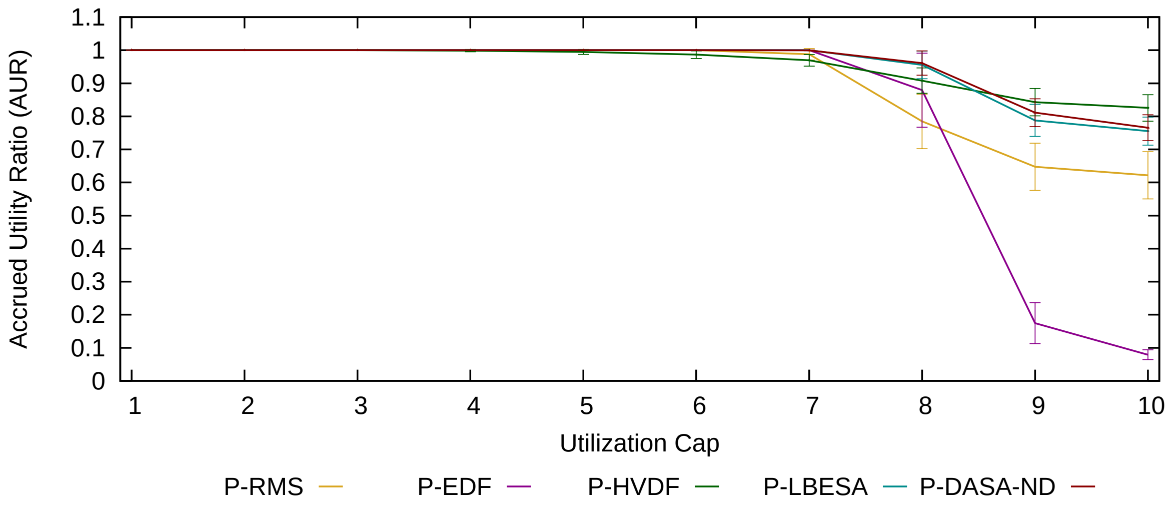
Figure C.4: 8-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

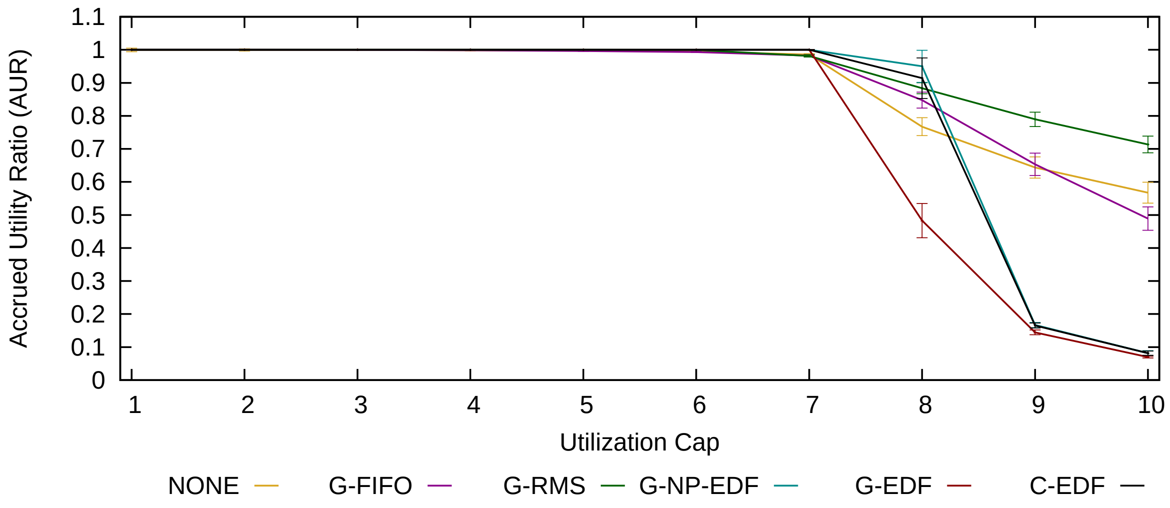


(b)

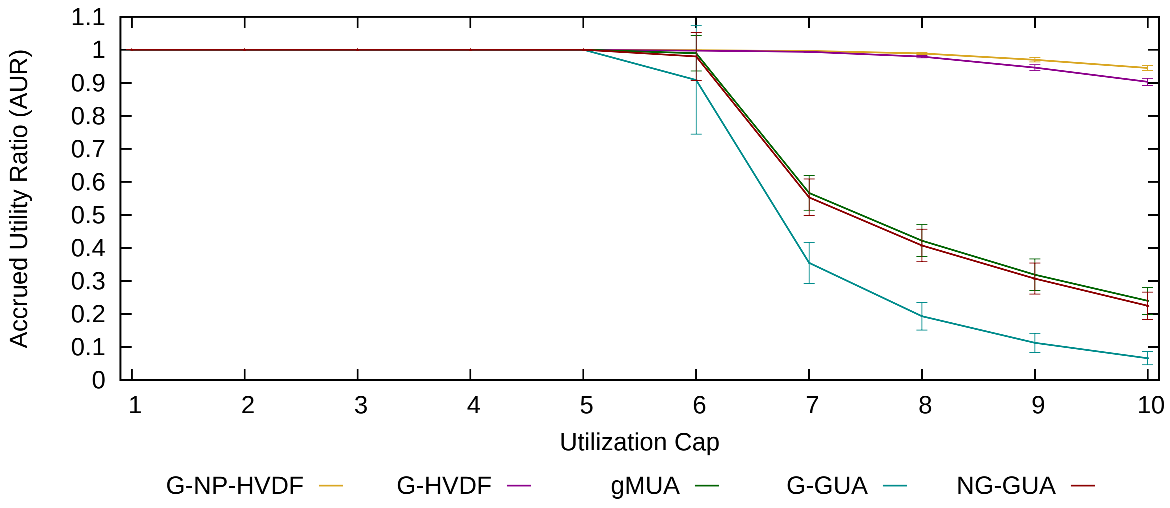


(c)

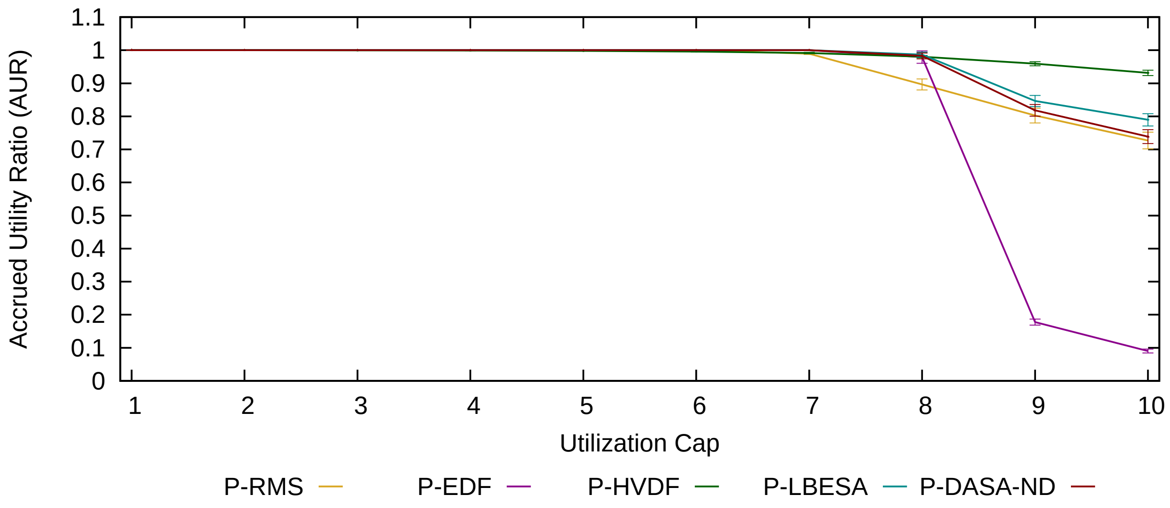
Figure C.5: 8-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

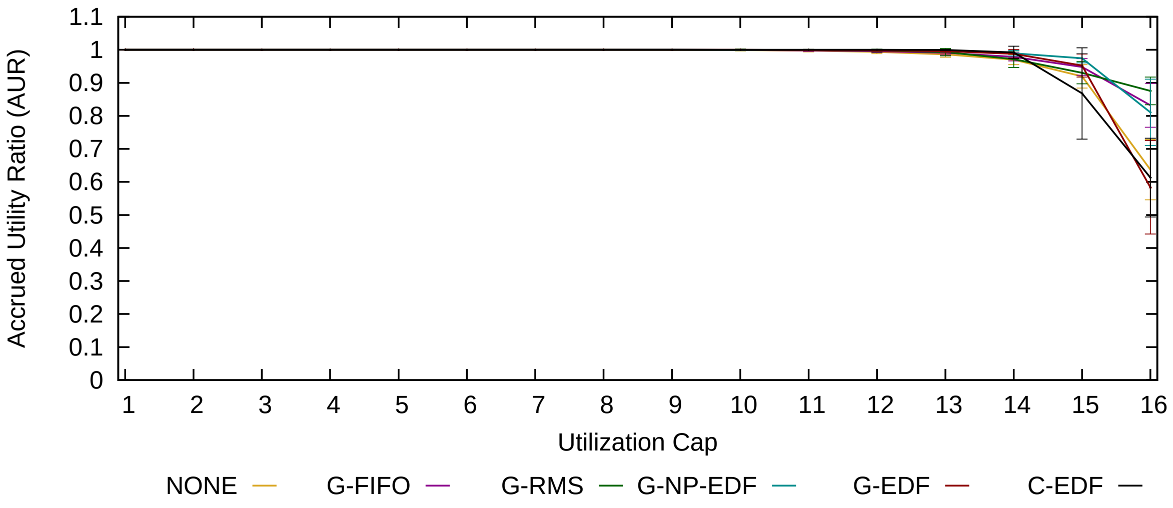


(b)

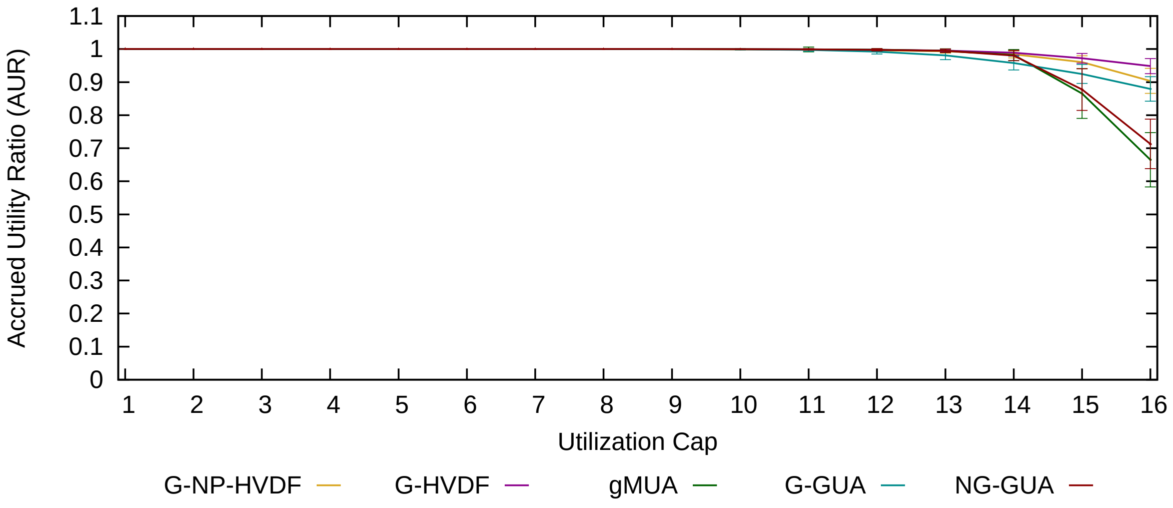


(c)

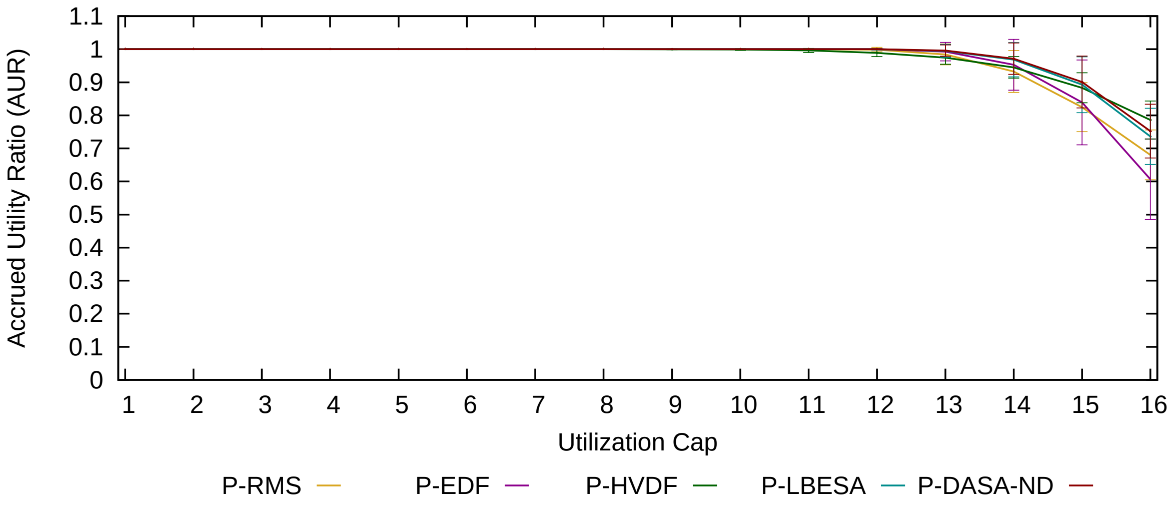
Figure C.6: 8-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

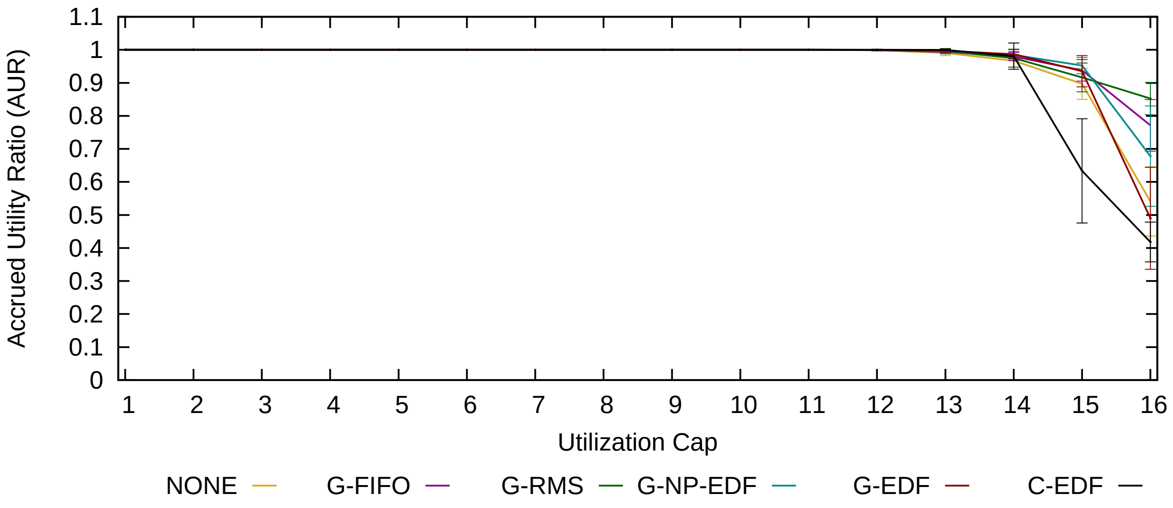


(b)

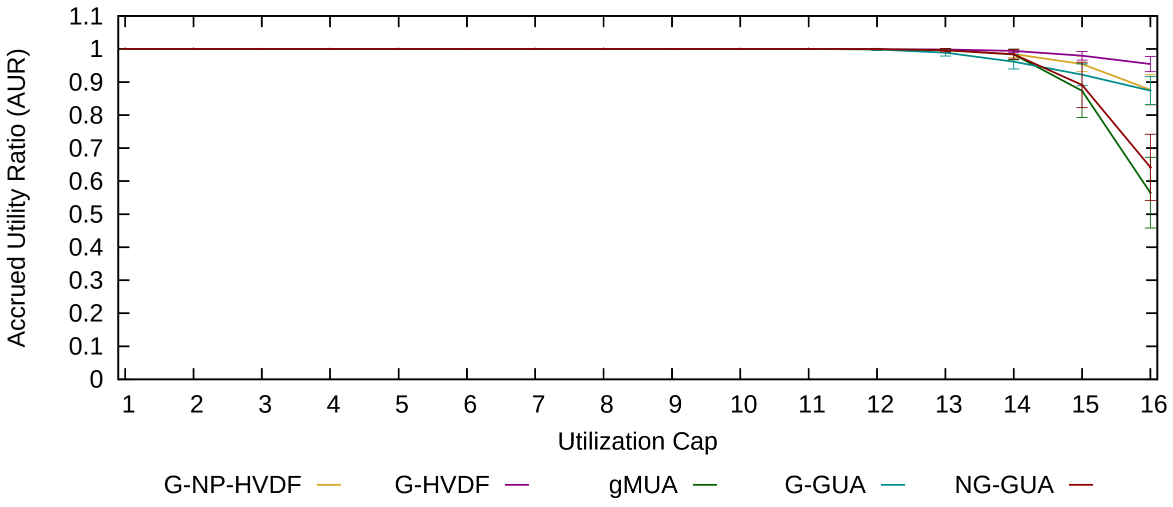


(c)

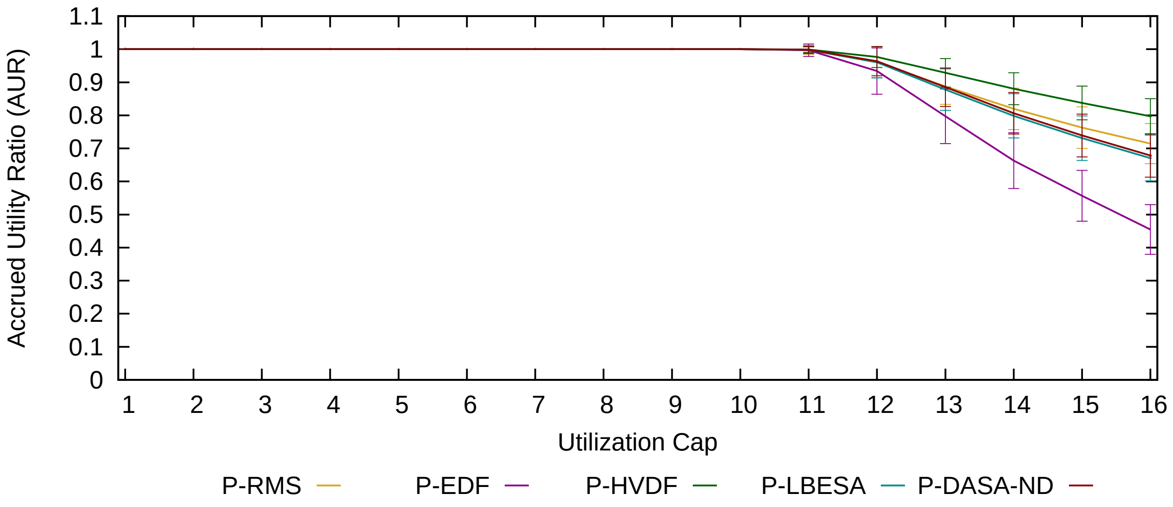
Figure C.7: 16-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

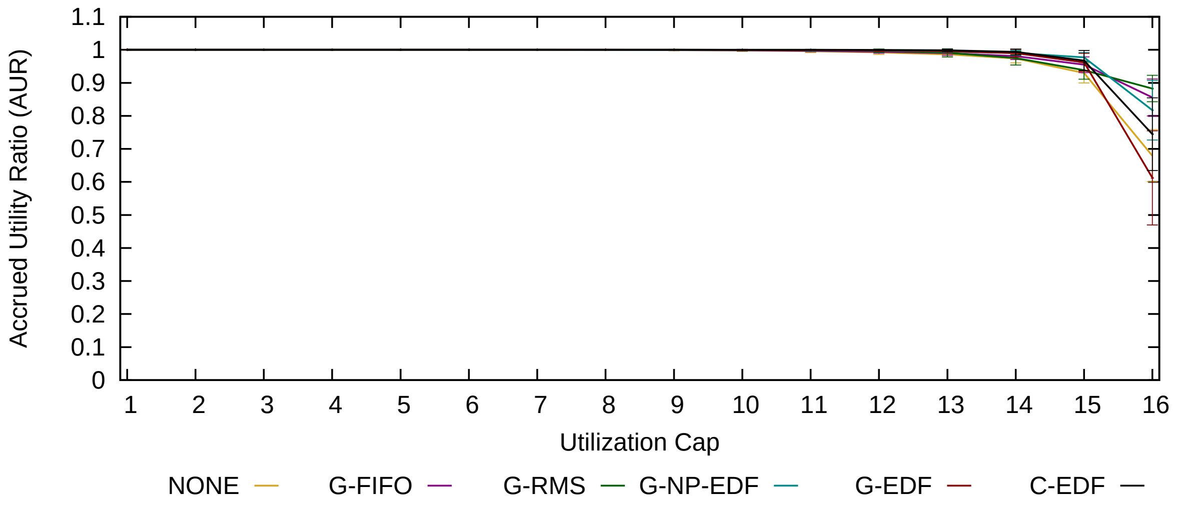


(b)

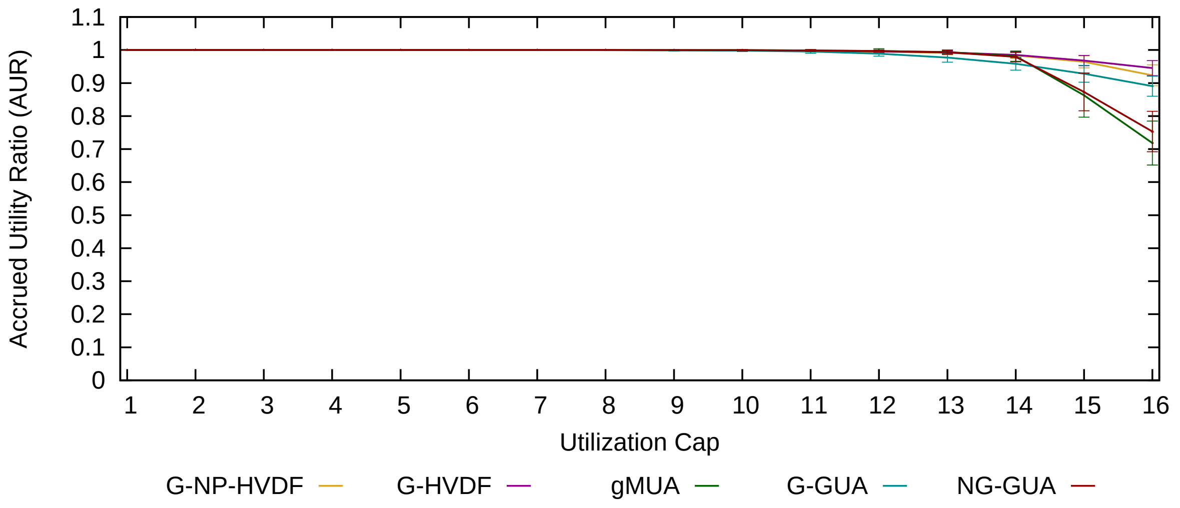


(c)

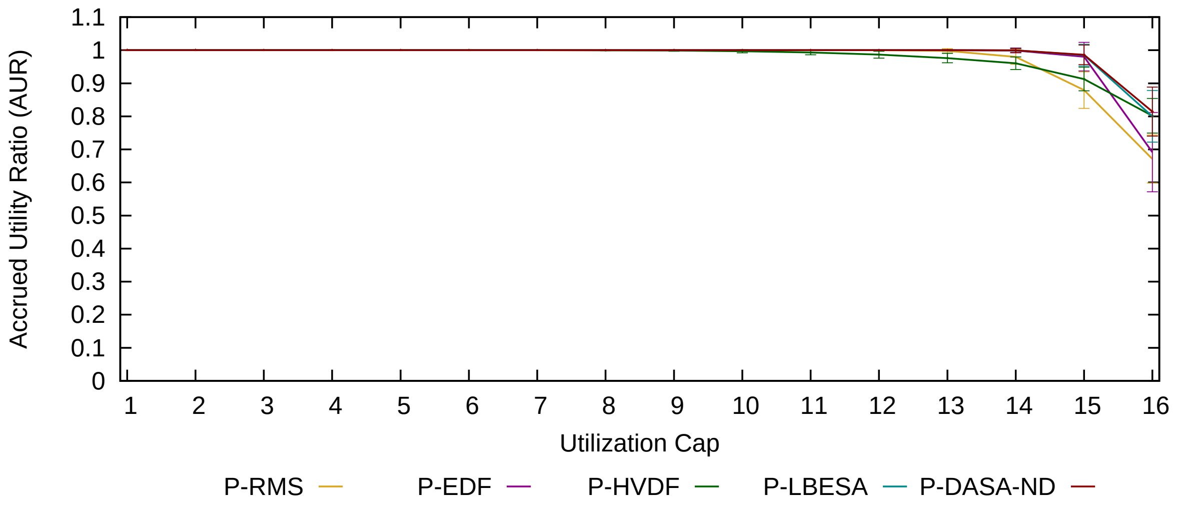
Figure C.8: 16-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

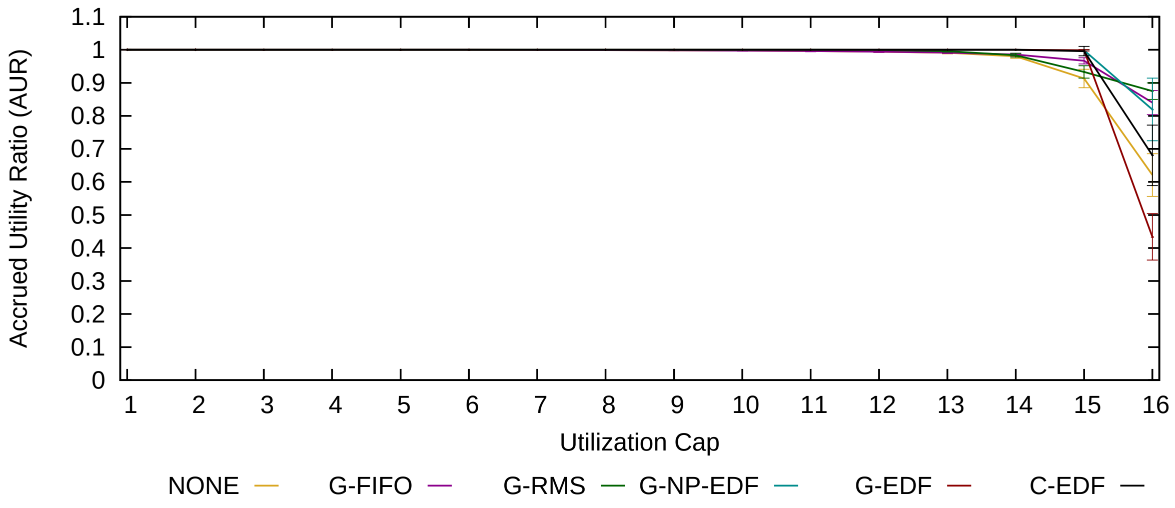


(b)

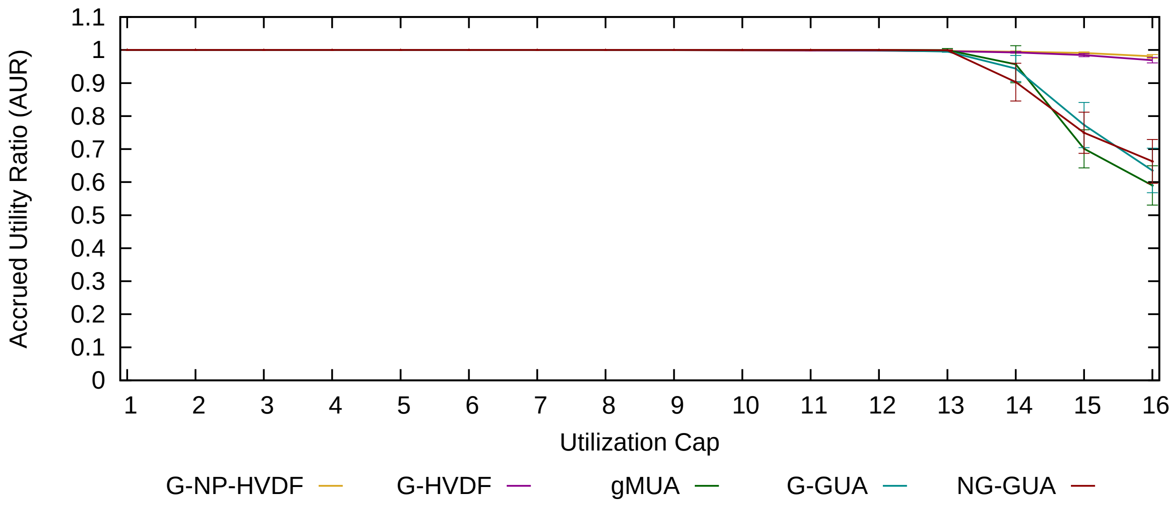


(c)

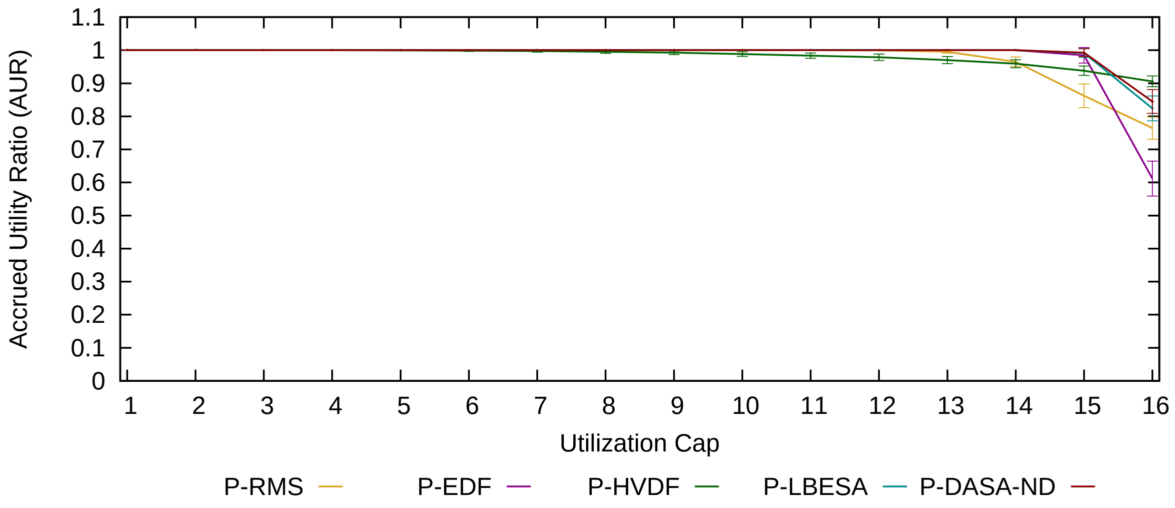
Figure C.9: 16-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

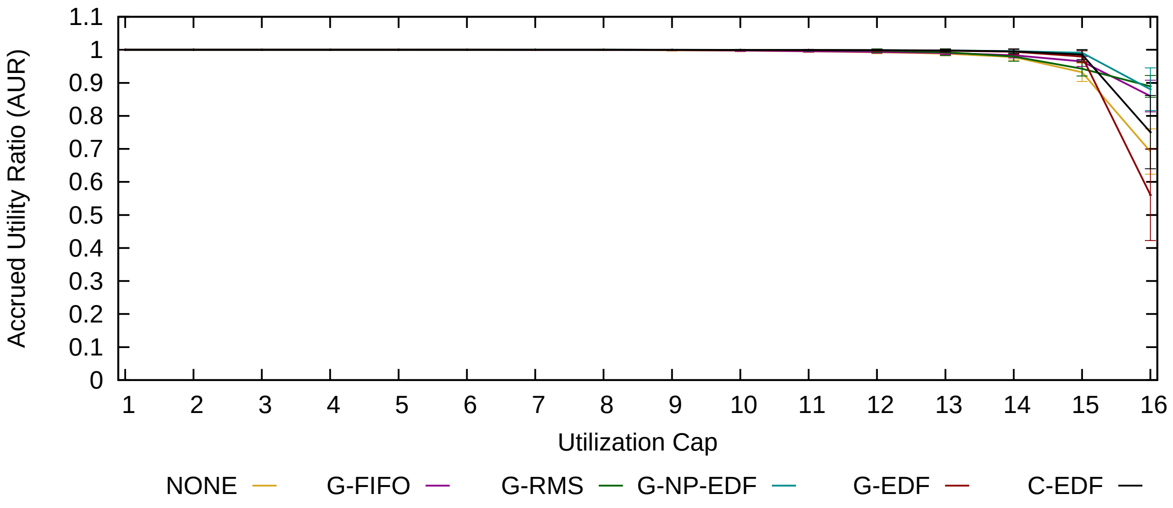


(b)

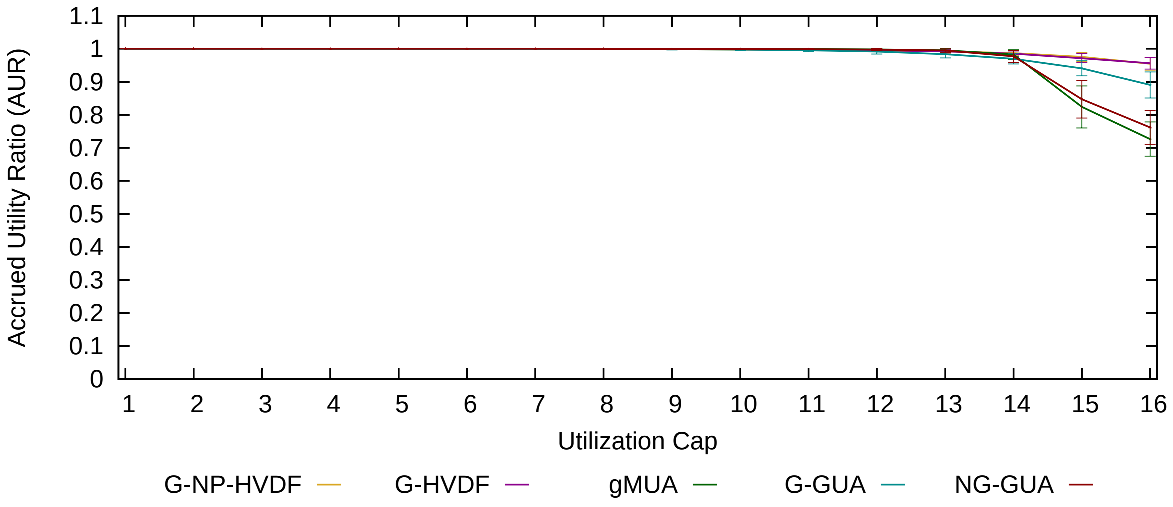


(c)

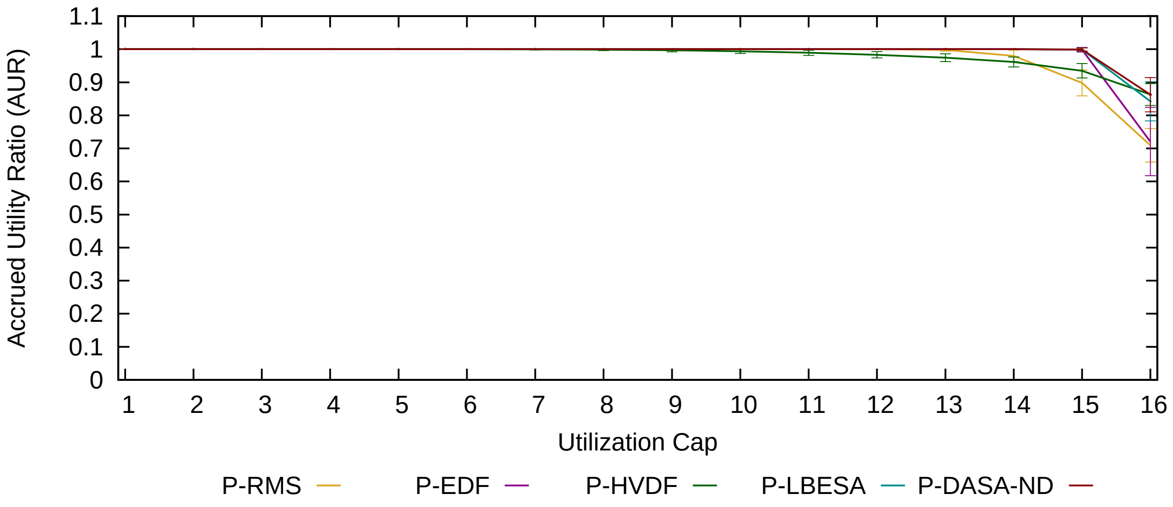
Figure C.10: 16-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

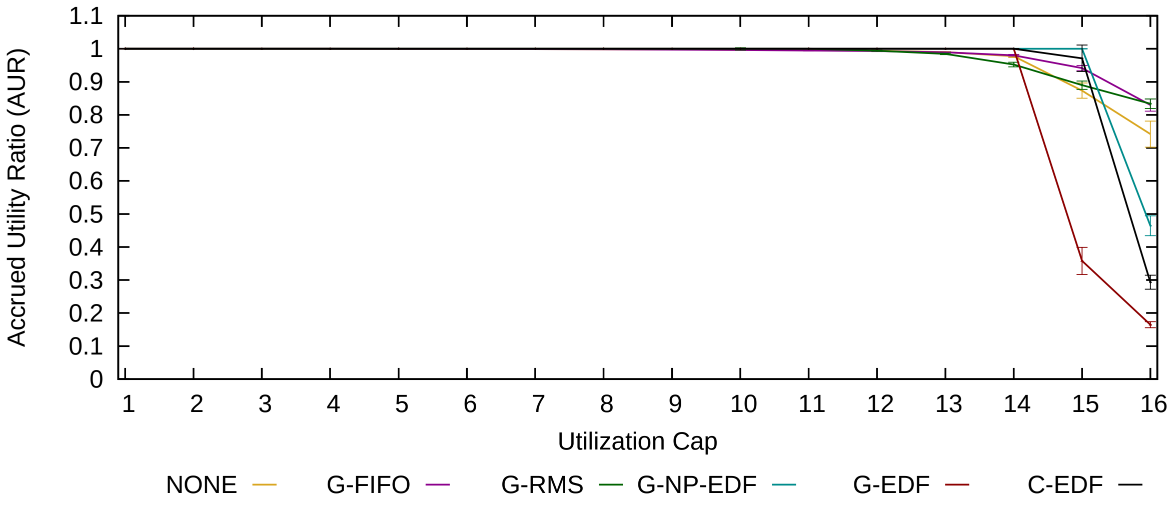


(b)

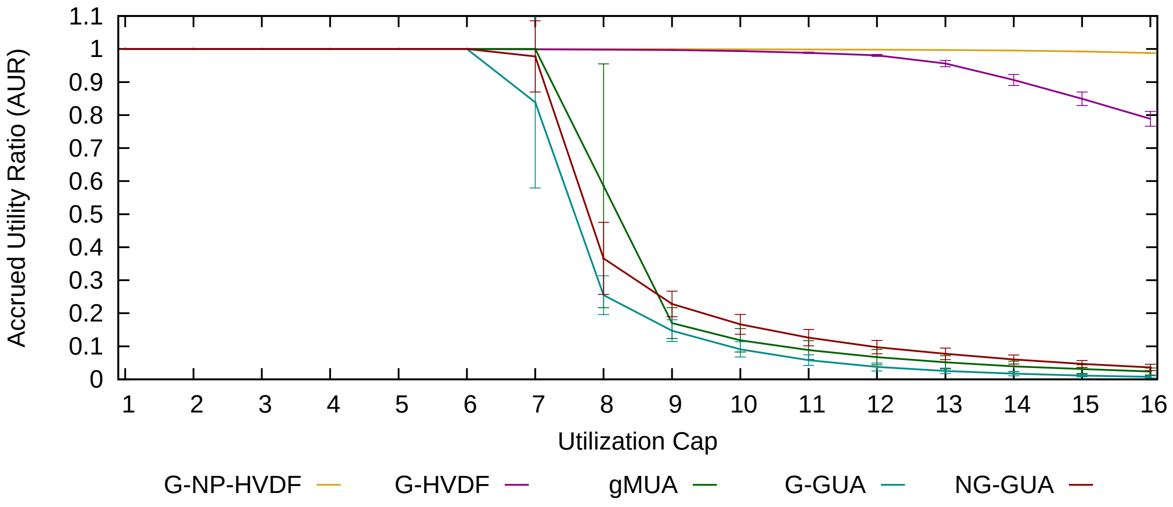


(c)

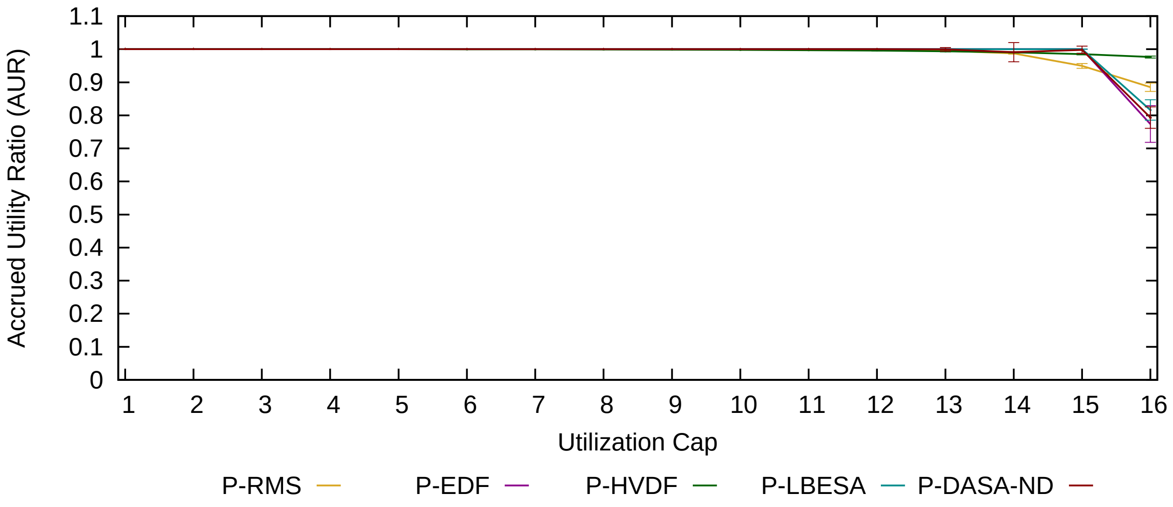
Figure C.11: 16-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

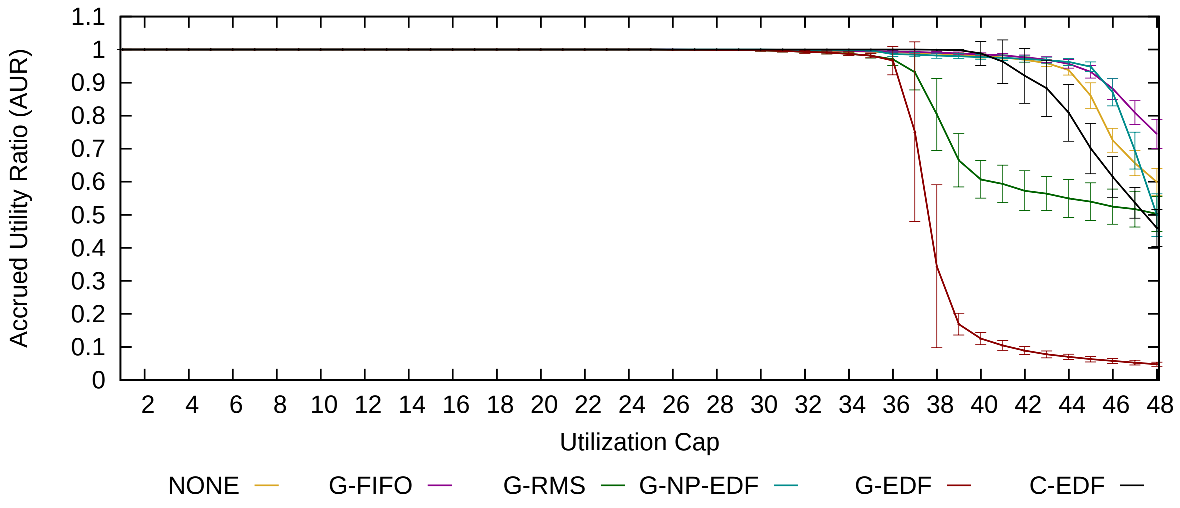


(b)

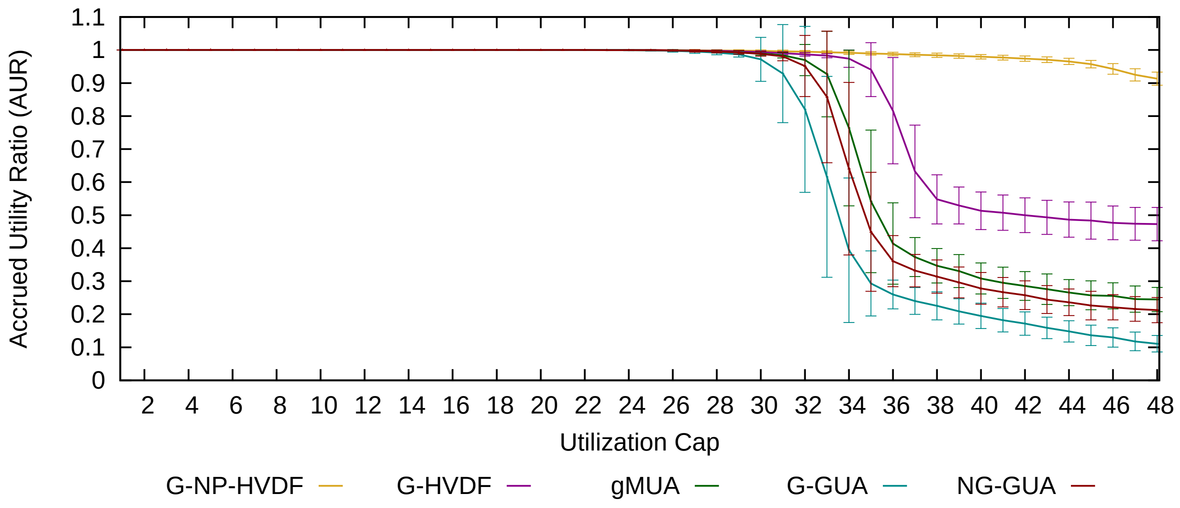


(c)

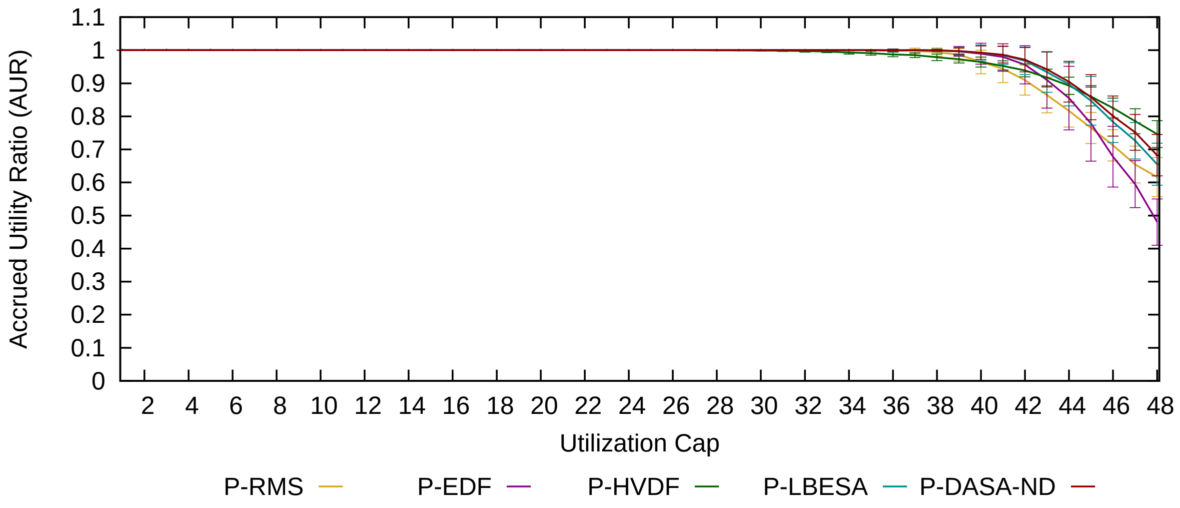
Figure C.12: 16-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

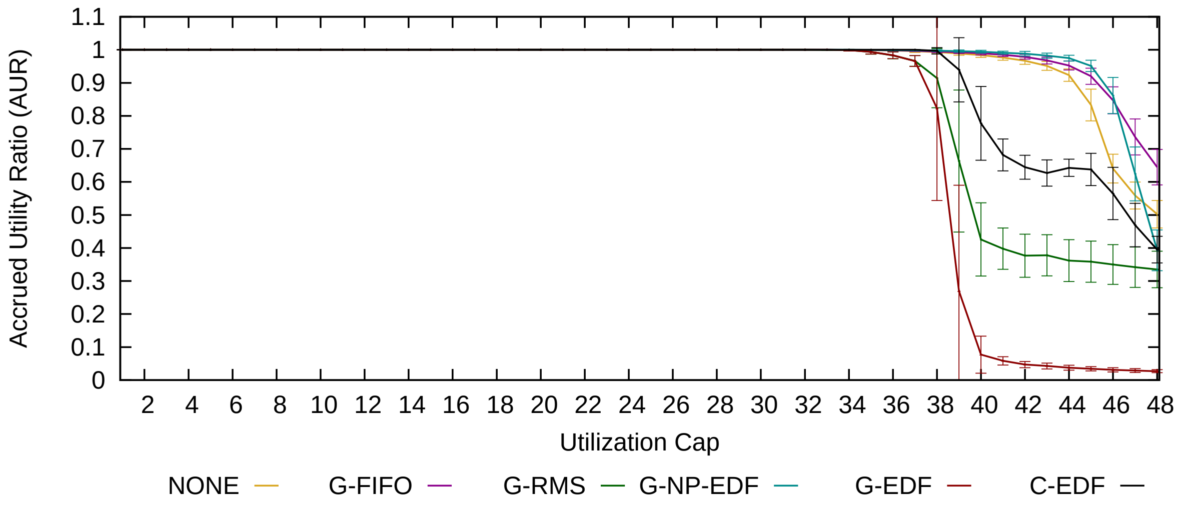


(b)

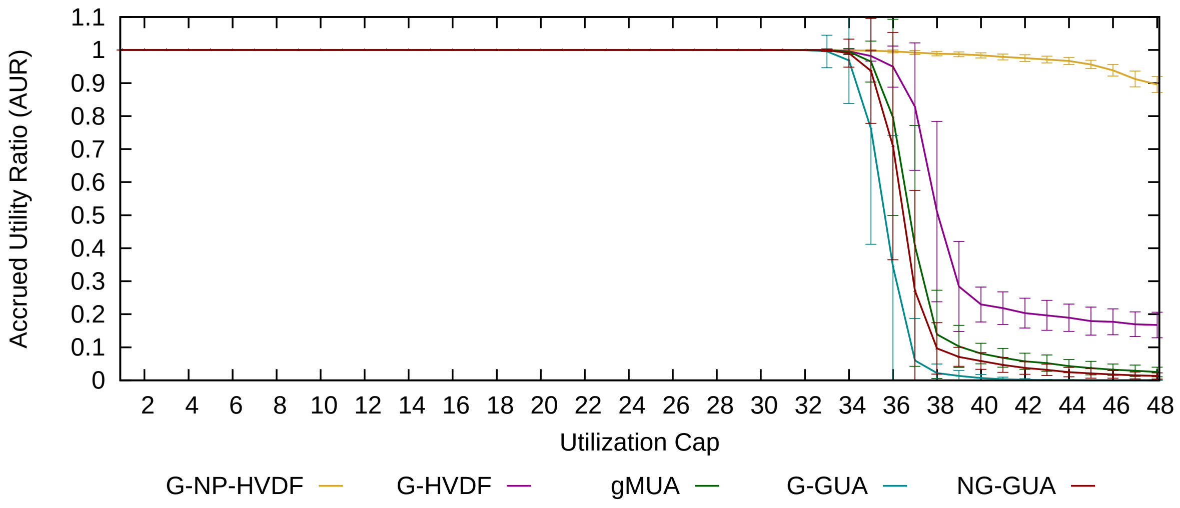


(c)

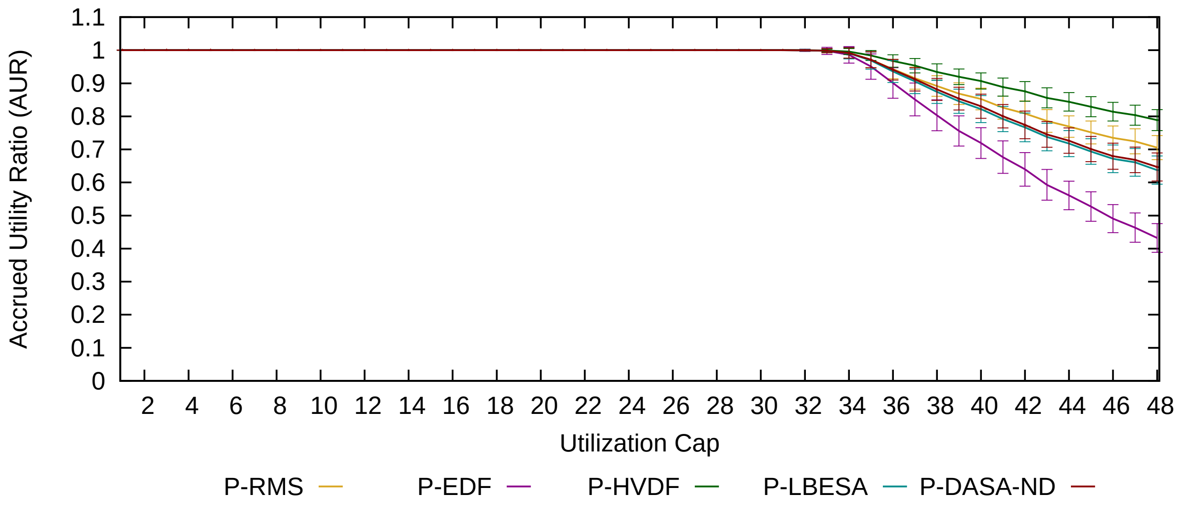
Figure C.13: 48-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

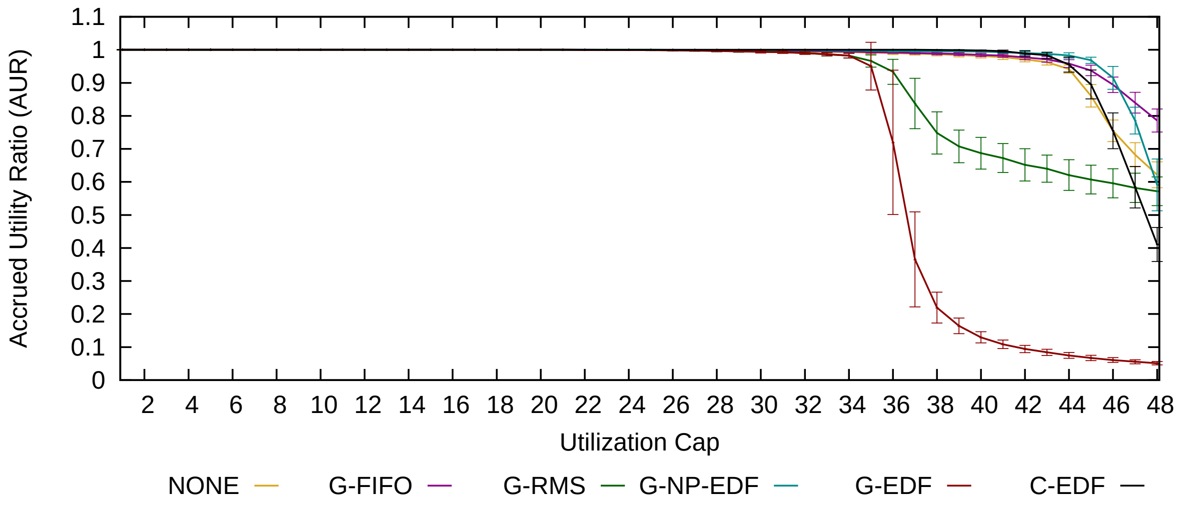


(b)

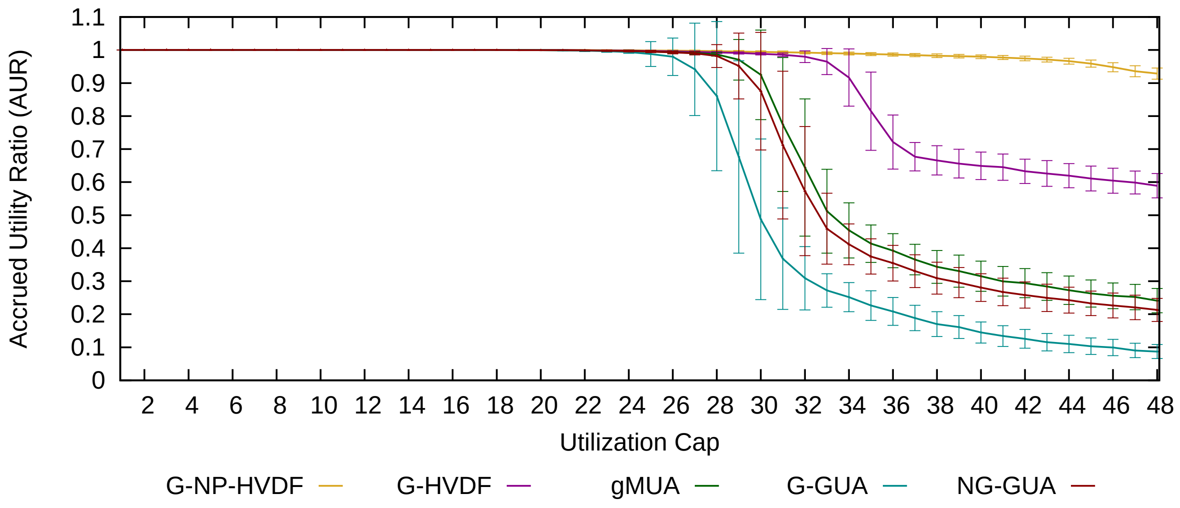


(c)

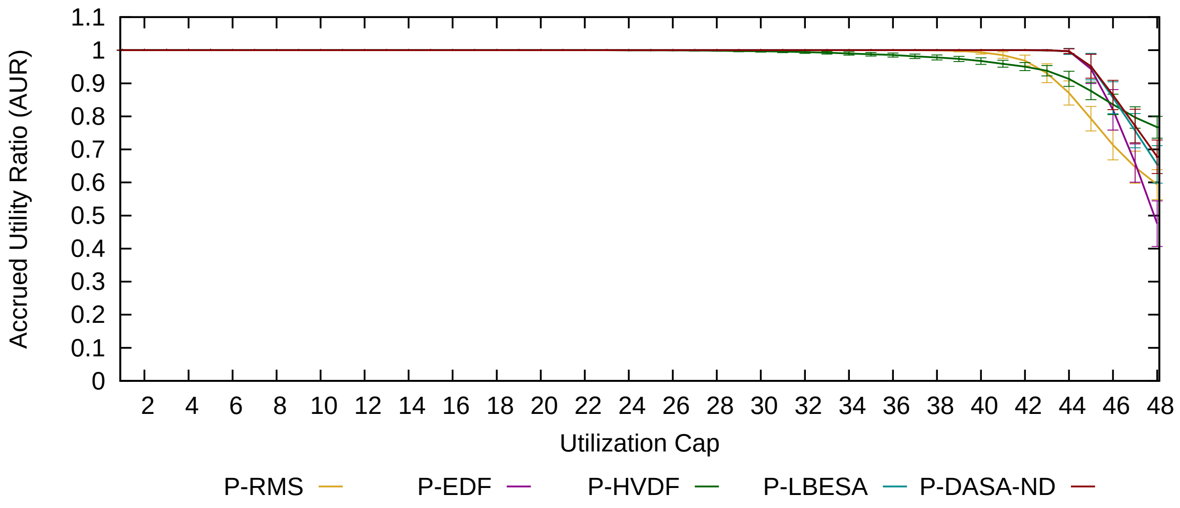
Figure C.14: 48-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

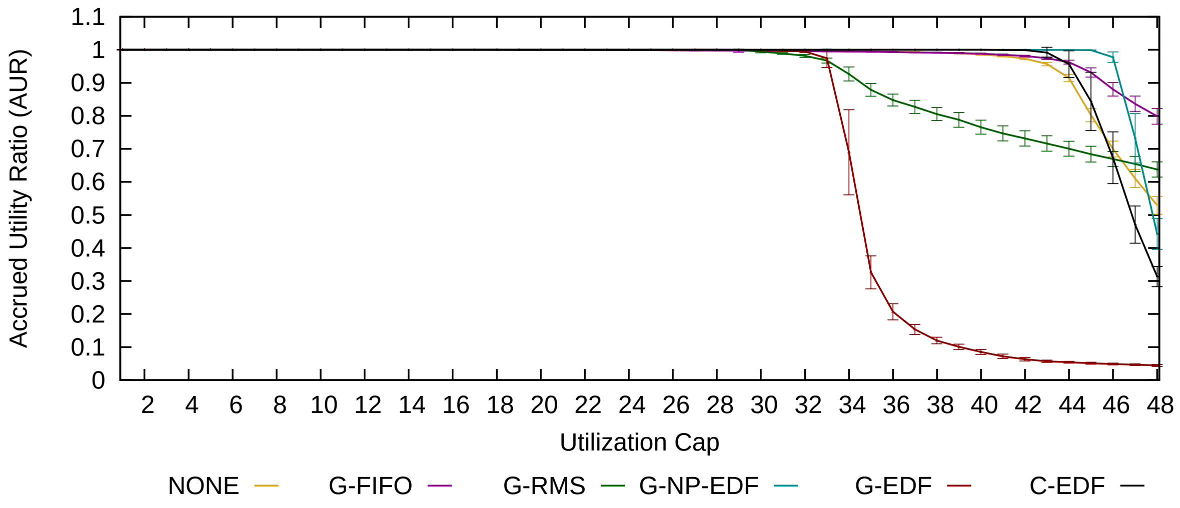


(b)

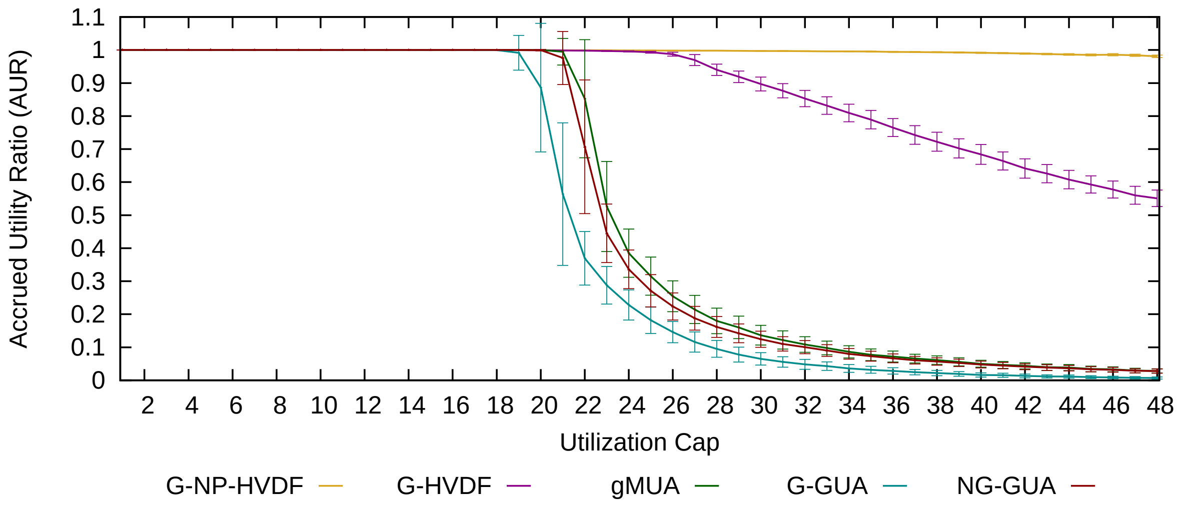


(c)

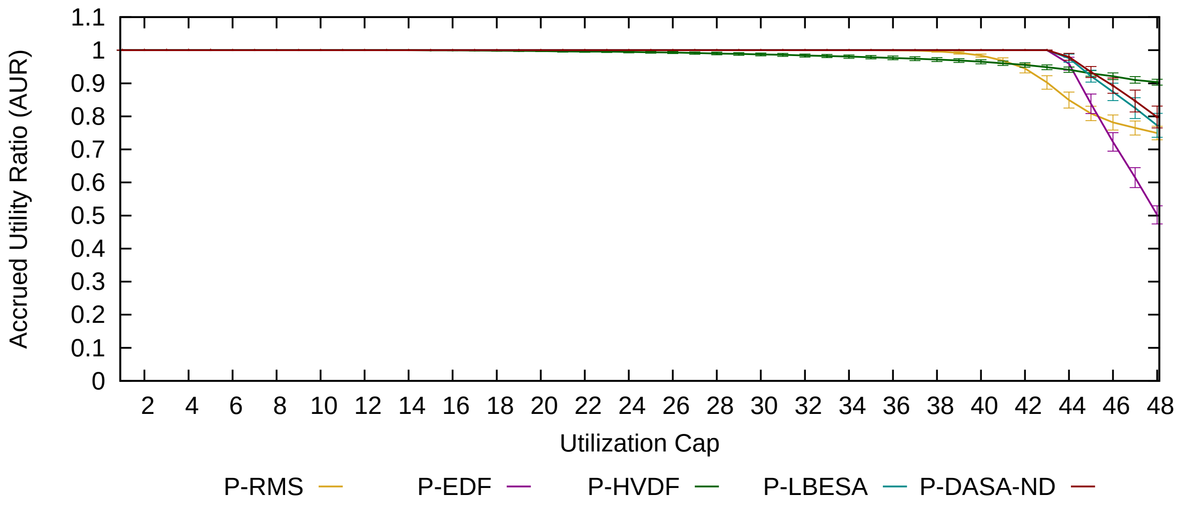
Figure C.15: 48-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

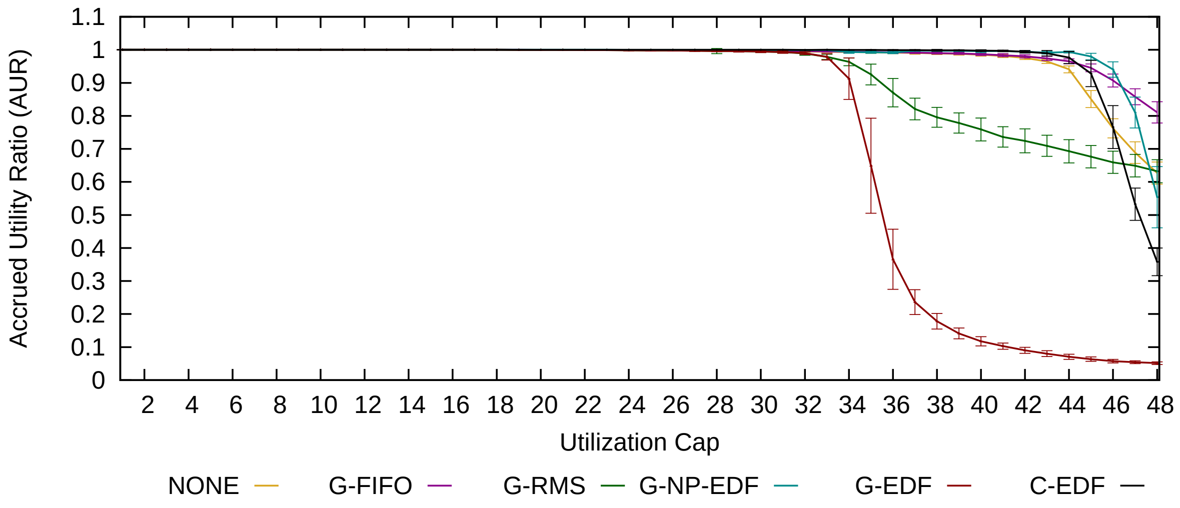


(b)

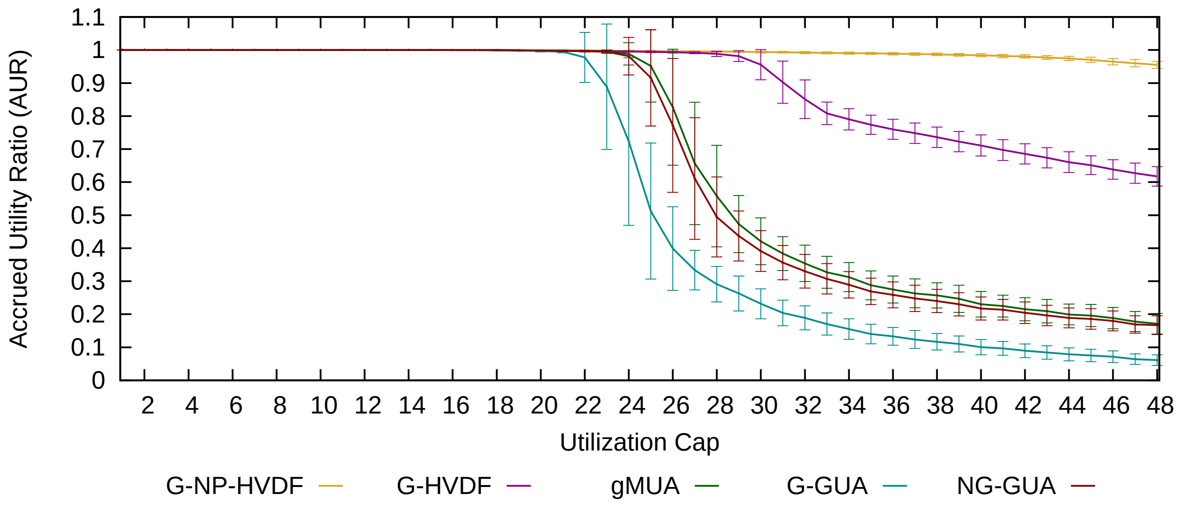


(c)

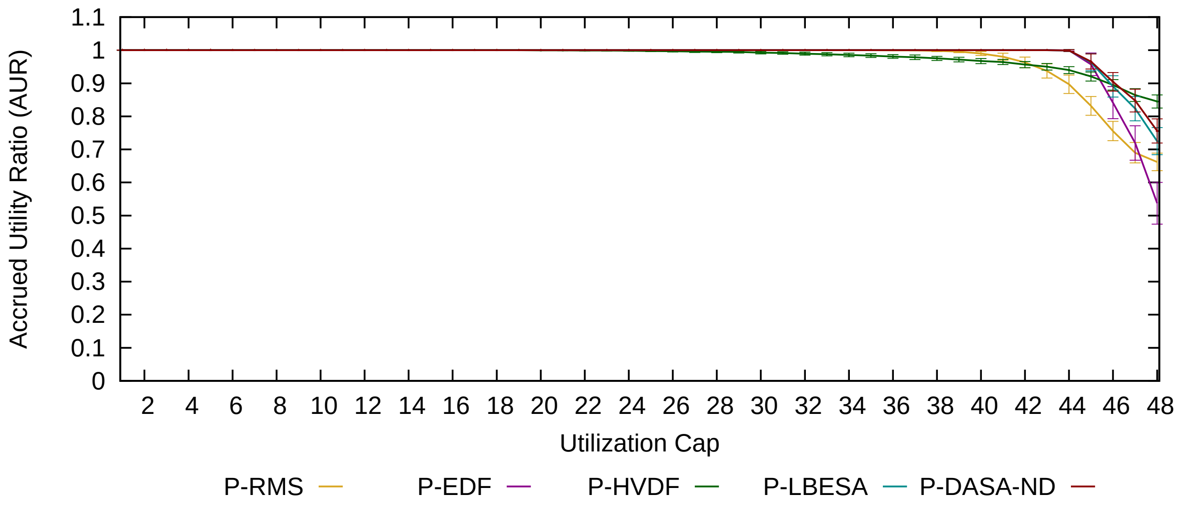
Figure C.16: 48-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

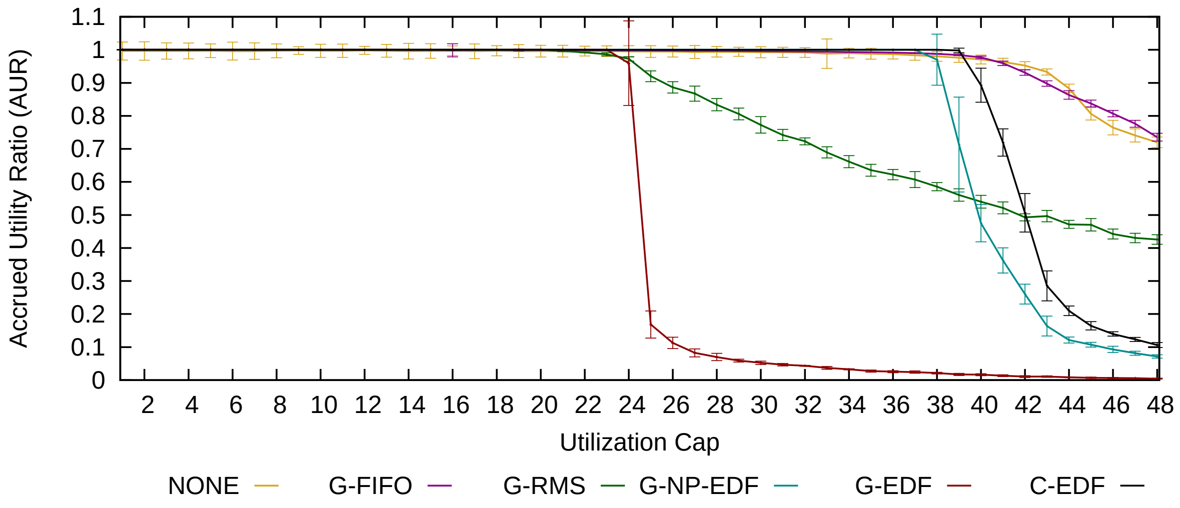


(b)

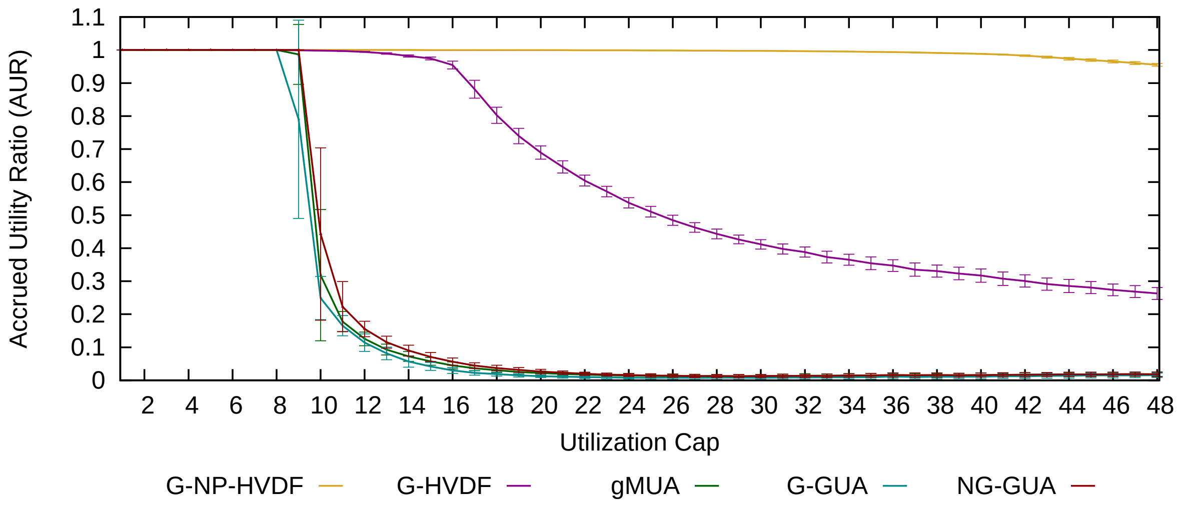


(c)

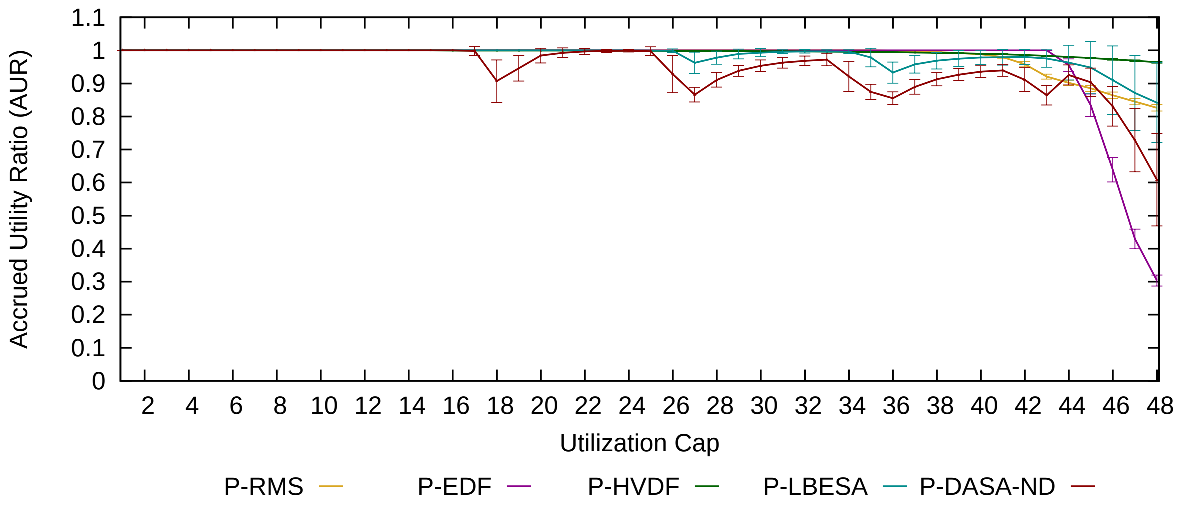
Figure C.17: 48-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)



(b)



(c)

Figure C.18: 48-Core AUR results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions

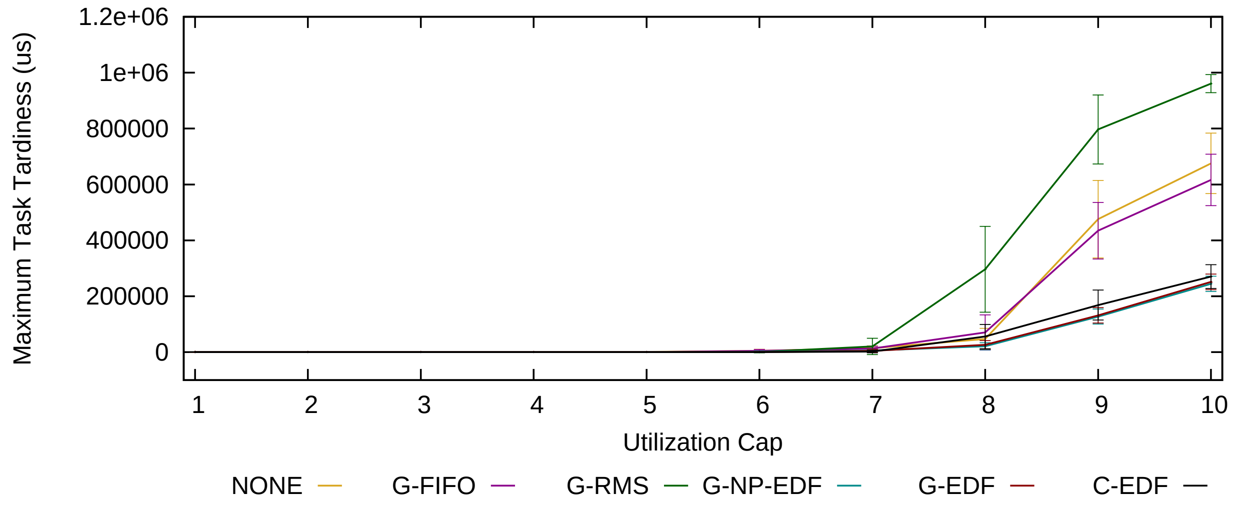
Appendix D

Complete Tardiness Results

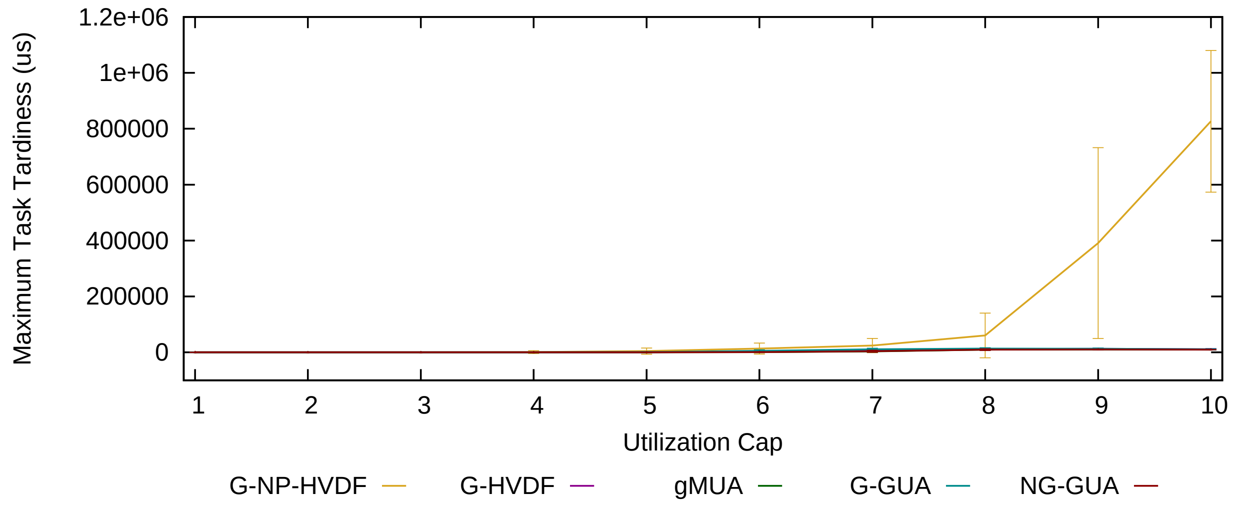
This appendix provides our complete tardiness results. It contains 36 plots grouped in 12 figures by taskset distribution and machine. It is organized as follows:

- Figure D.1 shows tardiness results for our 8-core platform under heavy bimodal load
- Figure D.2 shows tardiness results for our 8-core platform under heavy uniform load
- Figure D.3 shows tardiness results for our 8-core platform under medium bimodal load
- Figure D.4 shows tardiness results for our 8-core platform under medium uniform load
- Figure D.5 shows tardiness results for our 8-core platform under light bimodal load
- Figure D.6 shows tardiness results for our 8-core platform under light uniform load
- Figure D.7 shows tardiness results for our 16-core platform under heavy bimodal load
- Figure D.8 shows tardiness results for our 16-core platform under heavy uniform load
- Figure D.9 shows tardiness results for our 16-core platform under medium bimodal load
- Figure D.10 shows tardiness results for our 16-core platform under medium uniform load
- Figure D.11 shows tardiness results for our 16-core platform under light bimodal load
- Figure D.12 shows tardiness results for our 16-core platform under light uniform load
- Figure D.13 shows tardiness results for our 48-core platform under heavy bimodal load
- Figure D.14 shows tardiness results for our 48-core platform under heavy uniform load

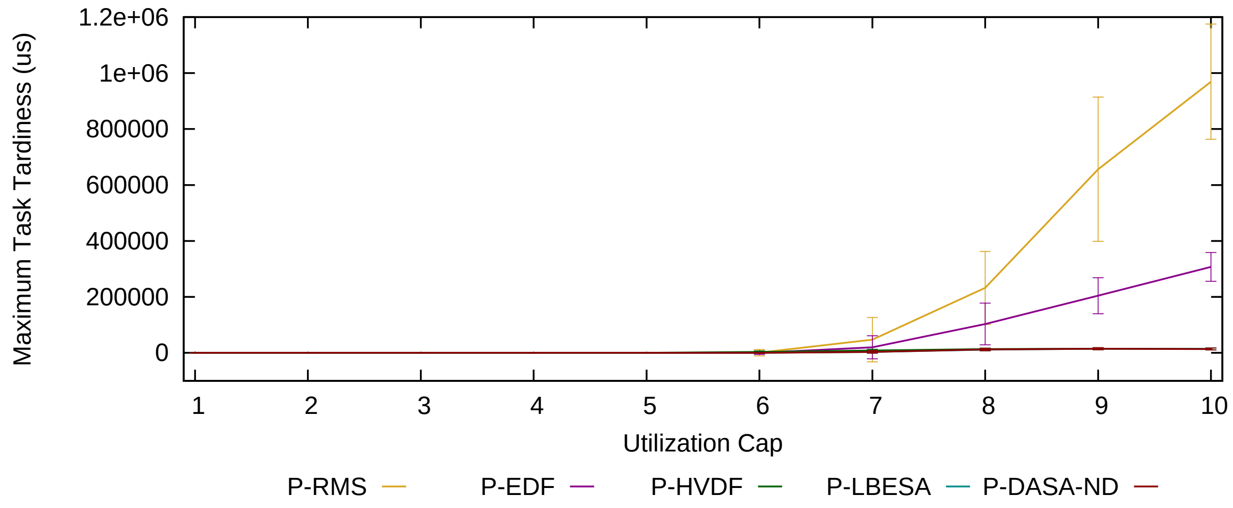
- Figure D.15 shows tardiness results for our 48-core platform under medium bimodal load
- Figure D.16 shows tardiness results for our 48-core platform under medium uniform load
- Figure D.17 shows tardiness results for our 48-core platform under light bimodal load
- Figure D.18 shows tardiness results for our 48-core platform under light uniform load



(a)

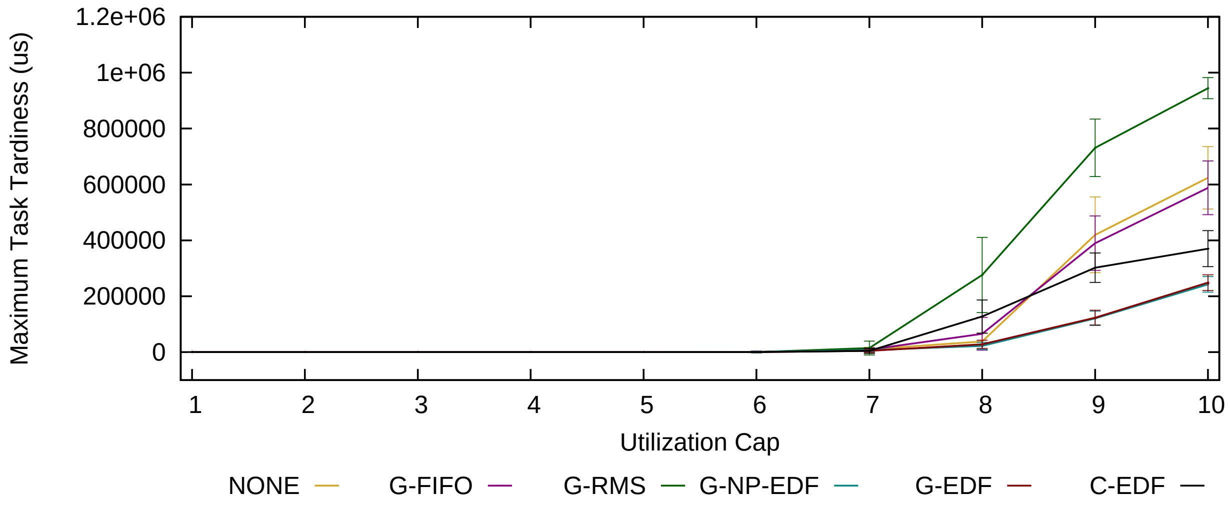


(b)

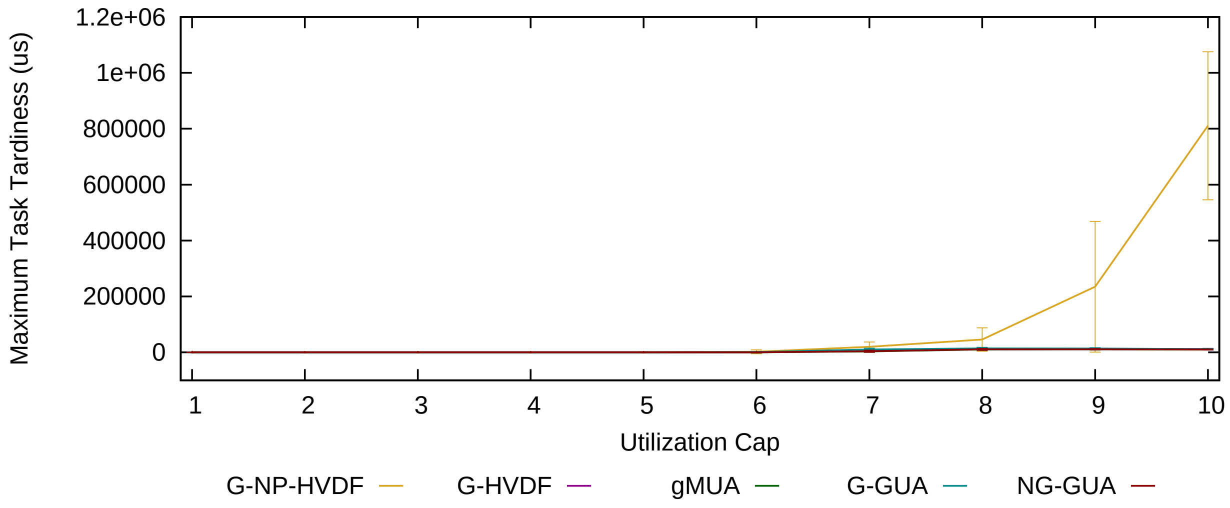


(c)

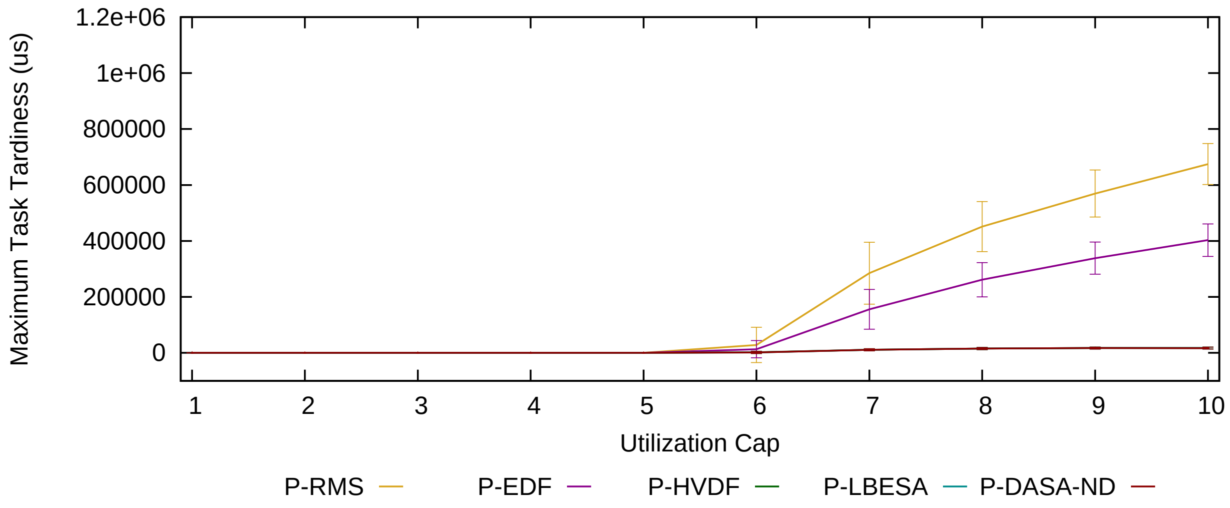
Figure D.1: 8-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

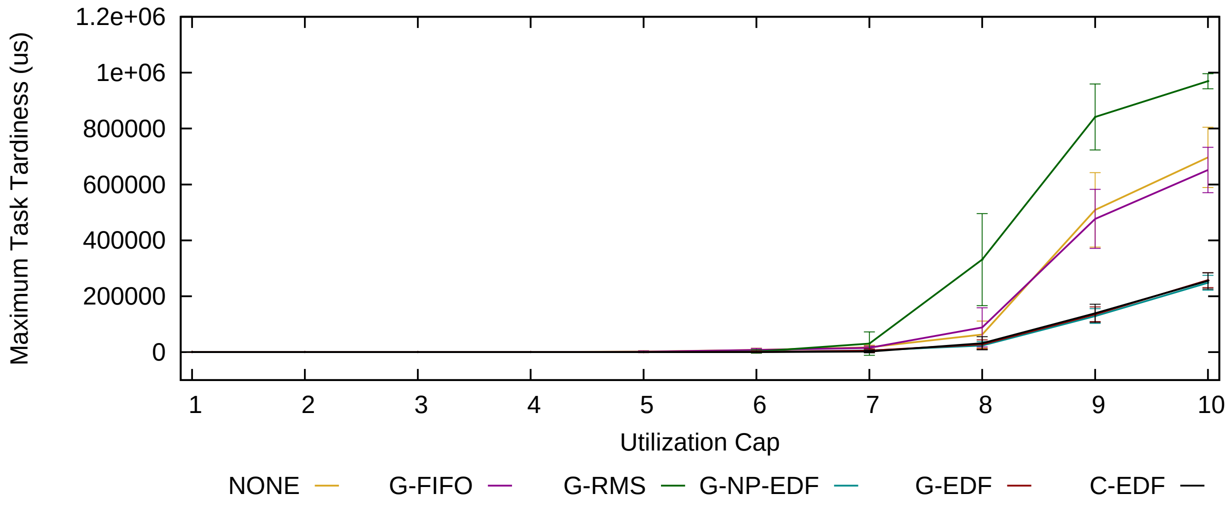


(b)

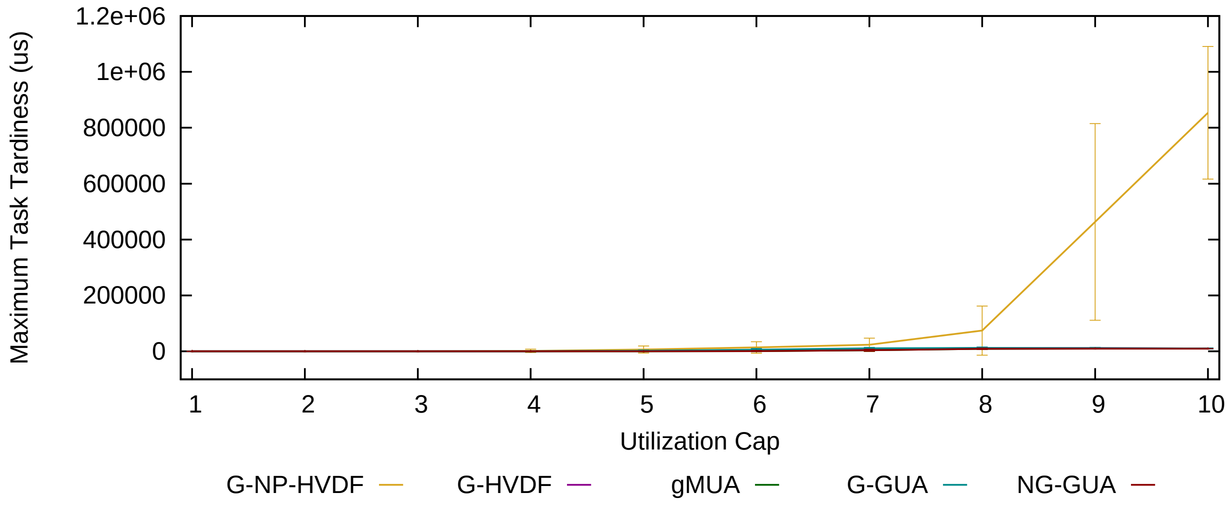


(c)

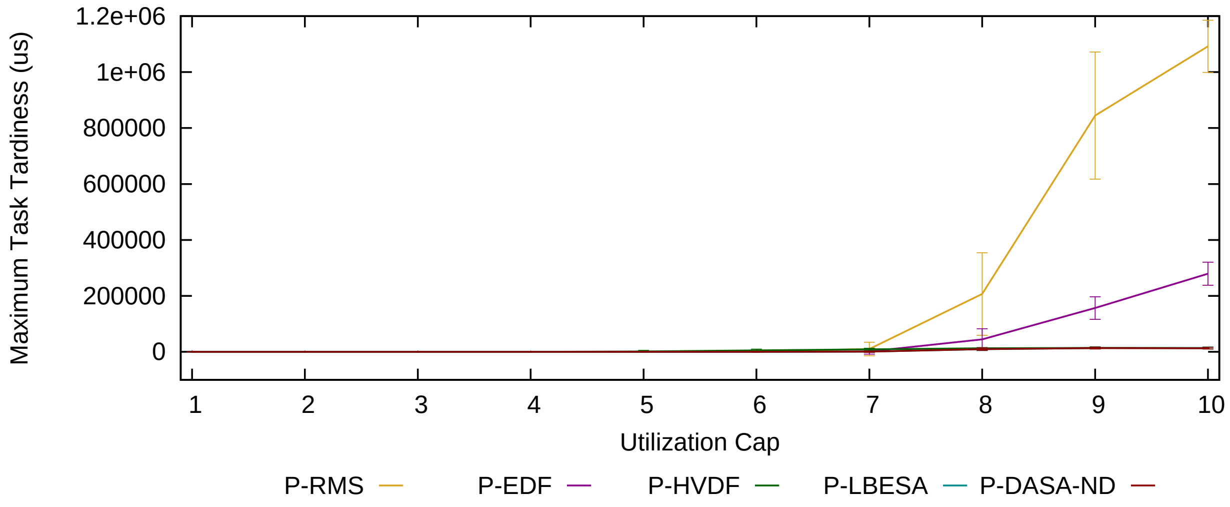
Figure D.2: 8-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

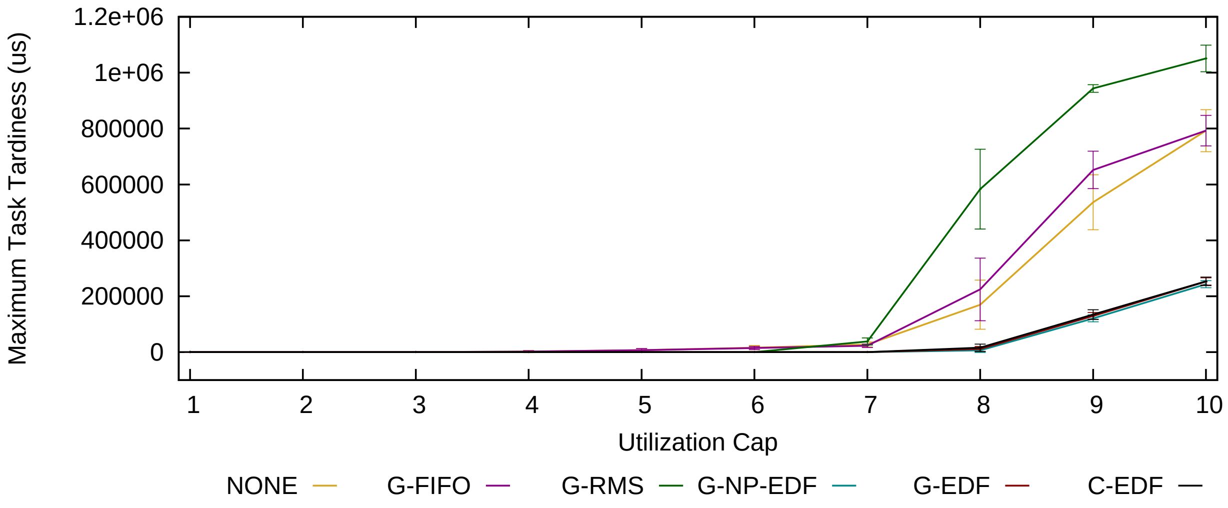


(b)

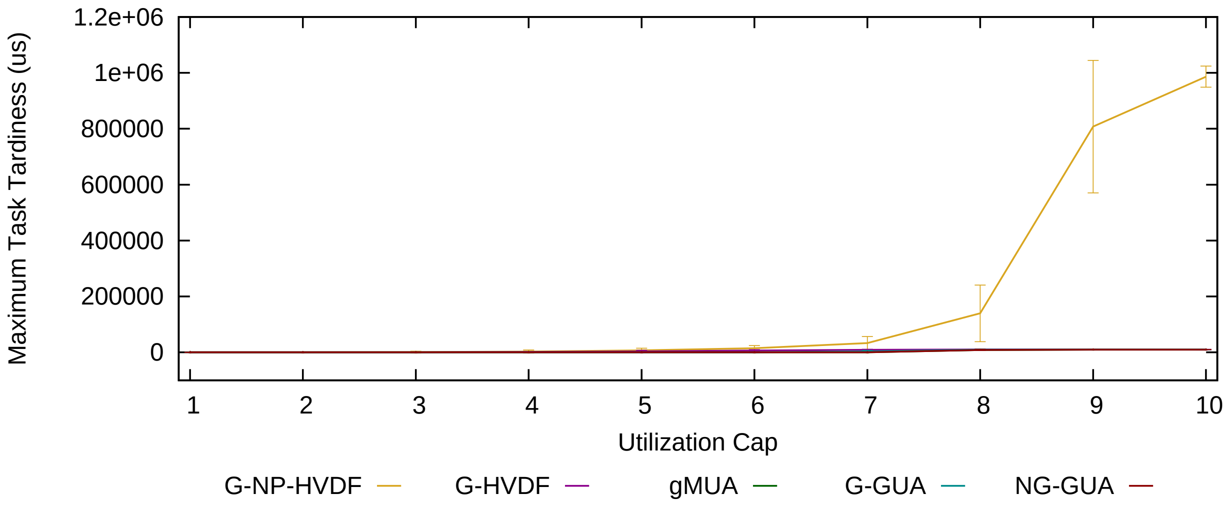


(c)

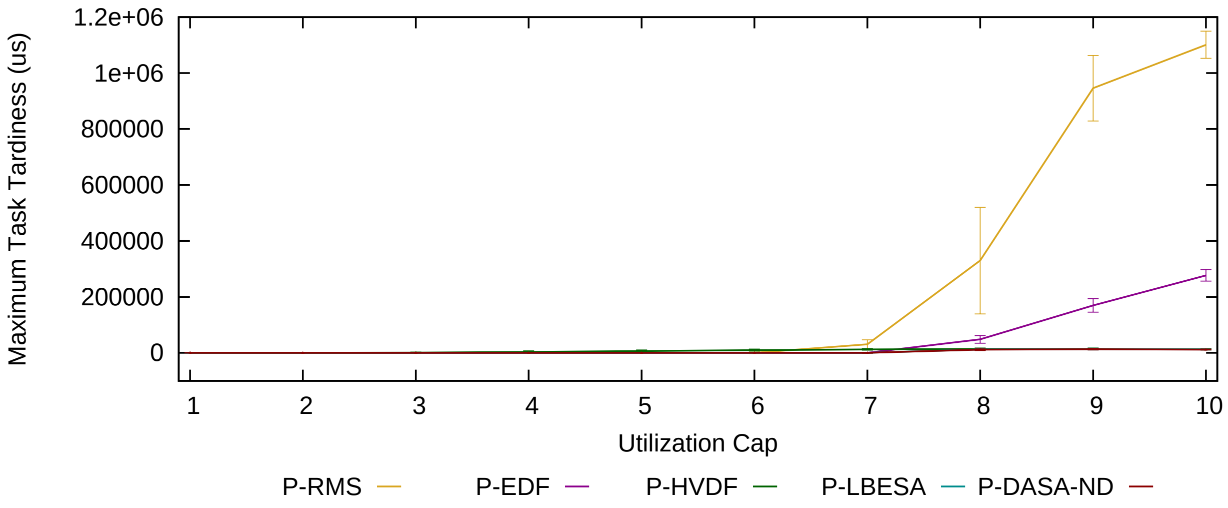
Figure D.3: 8-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

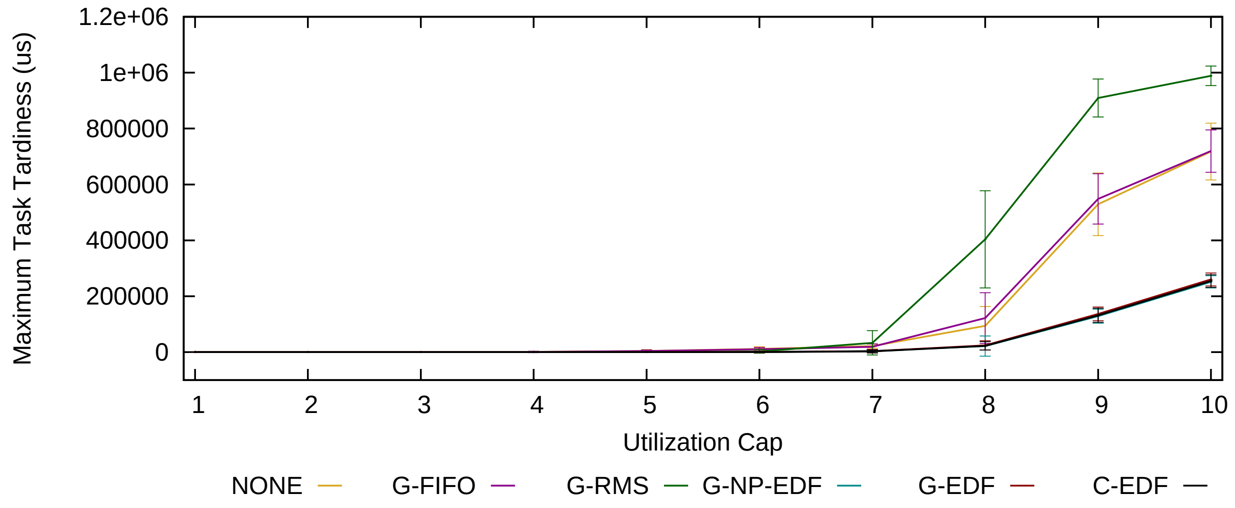


(b)

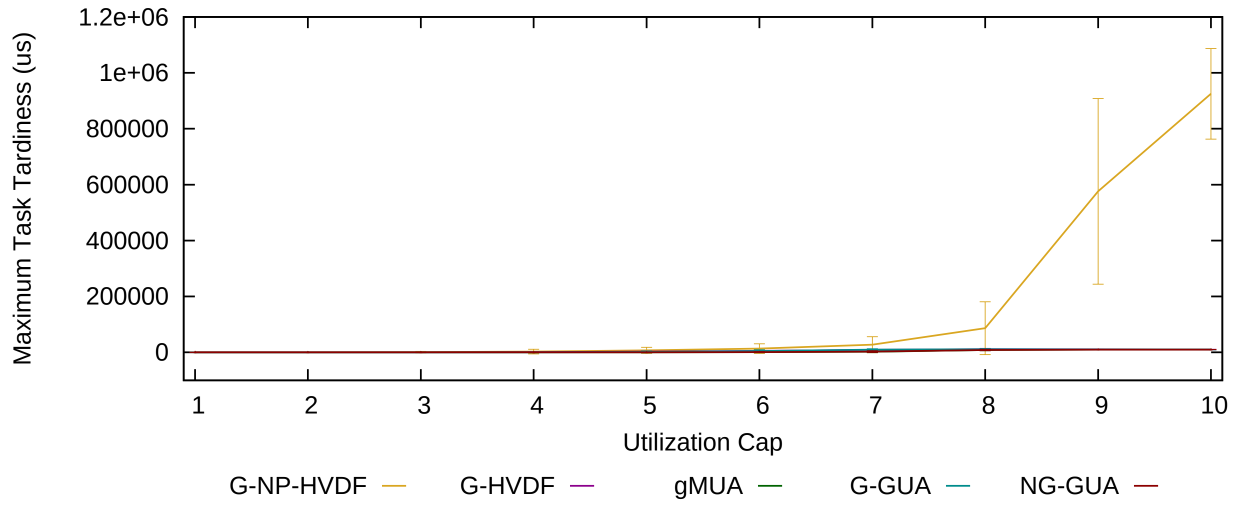


(c)

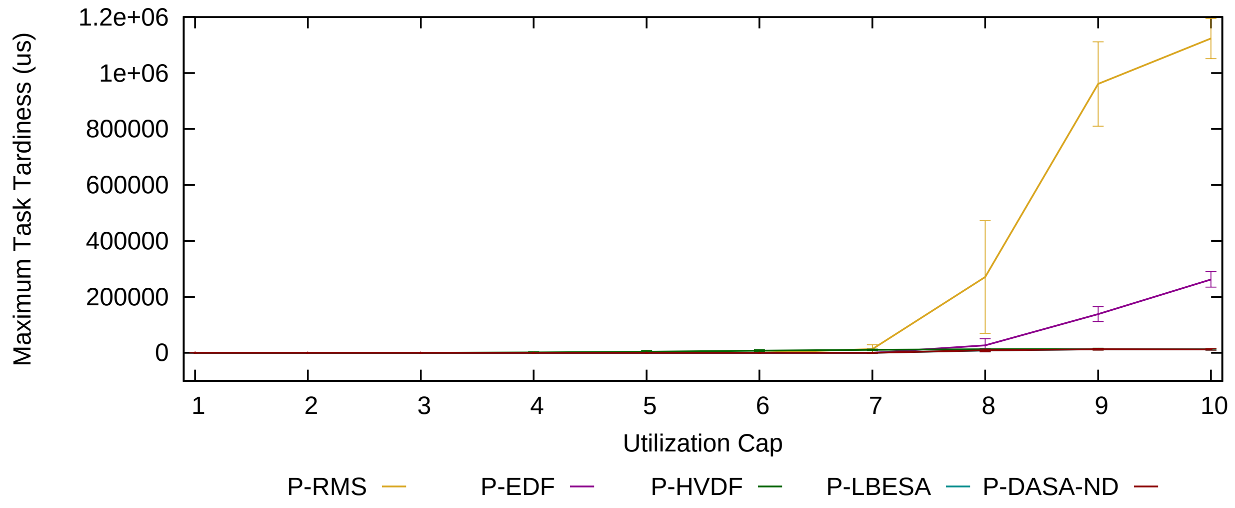
Figure D.4: 8-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

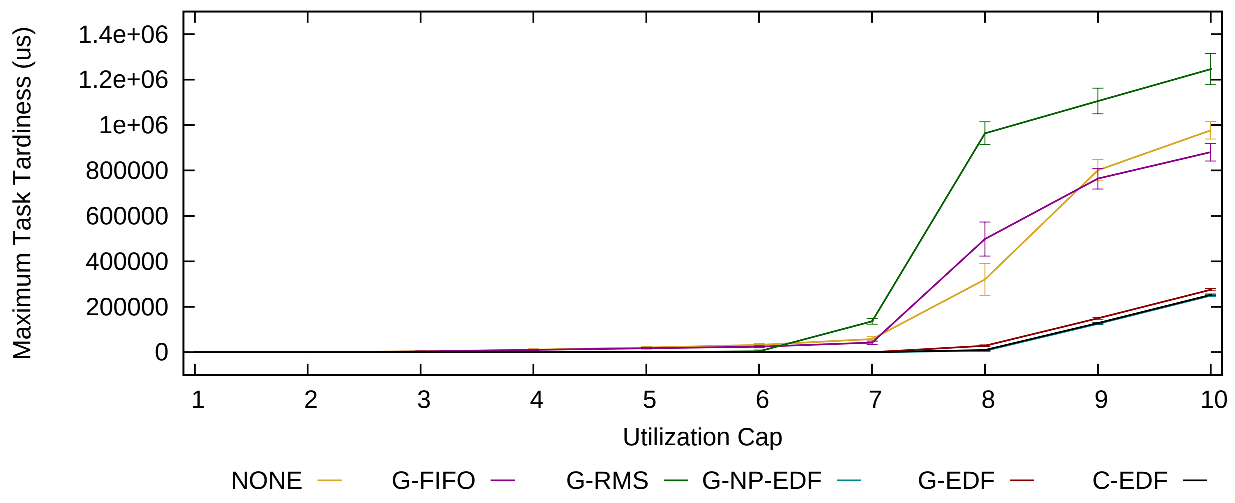


(b)

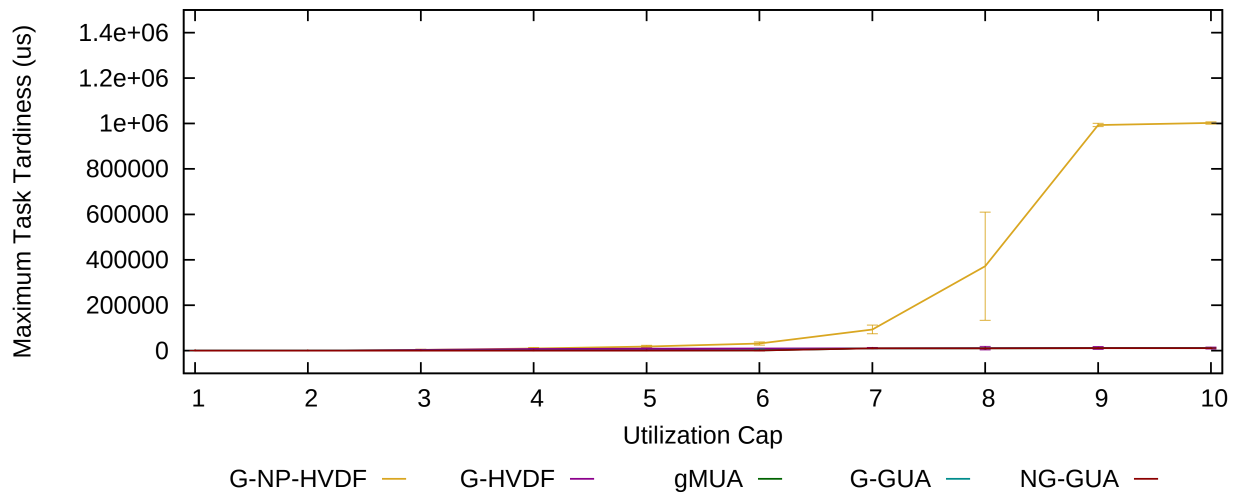


(c)

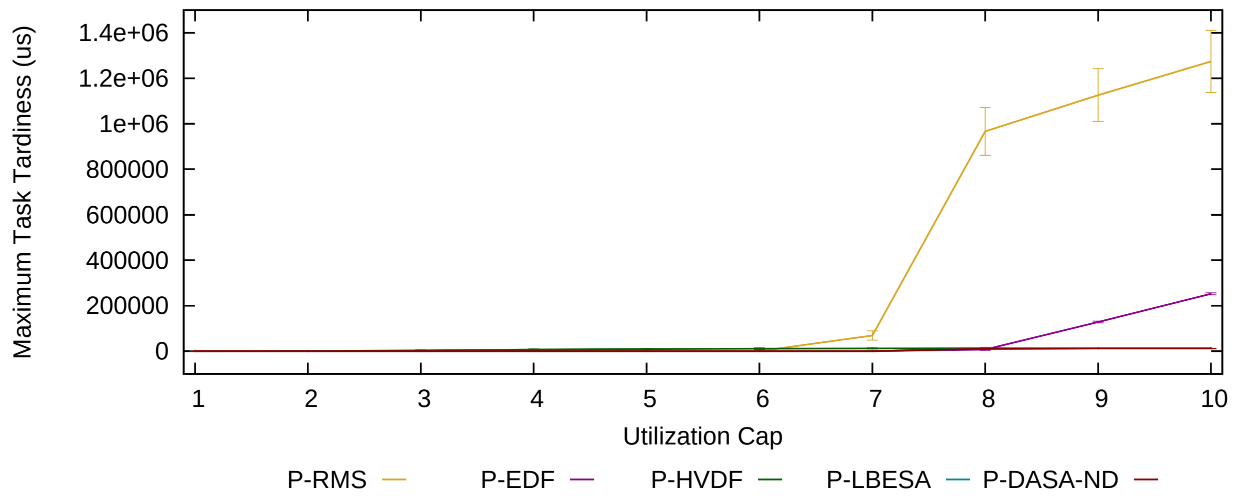
Figure D.5: 8-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

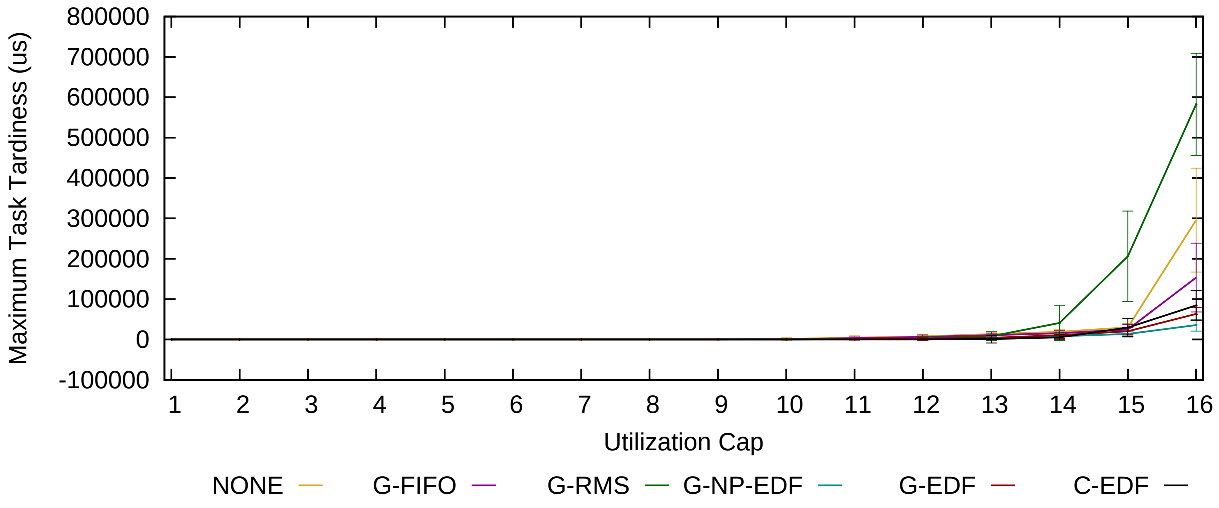


(b)

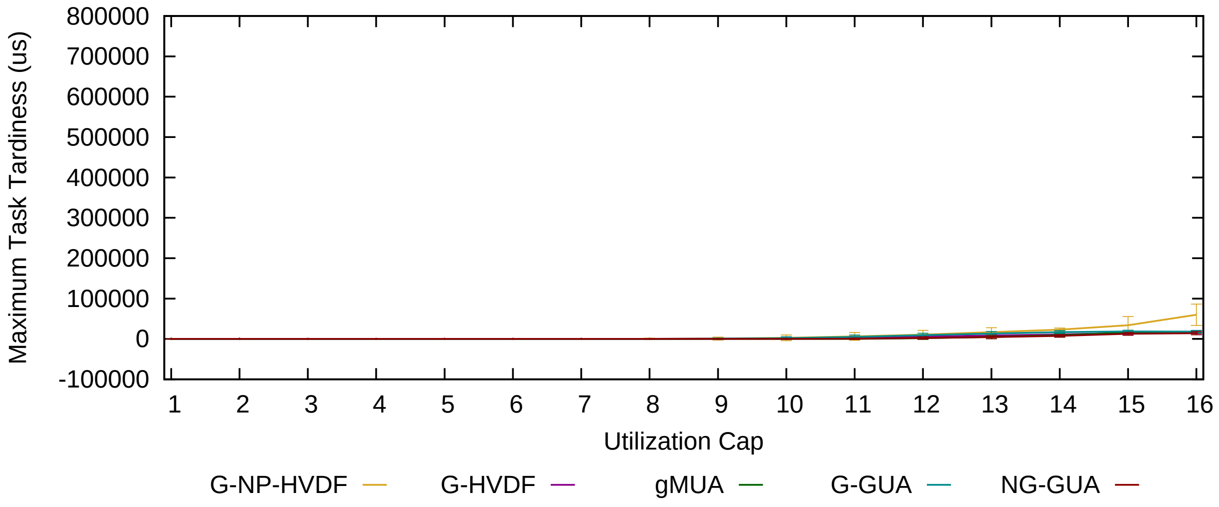


(c)

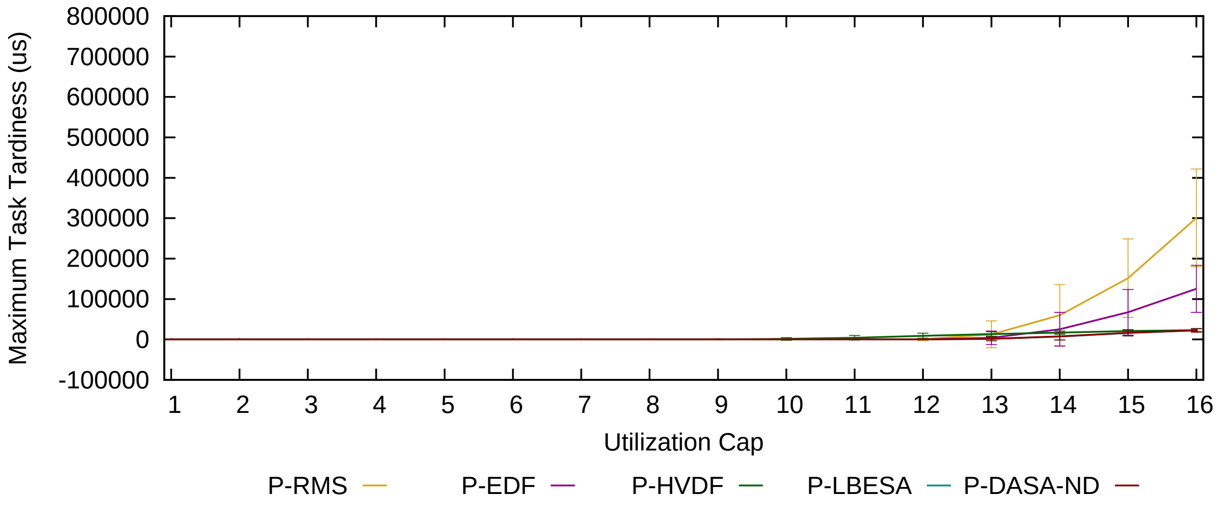
Figure D.6: 8-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

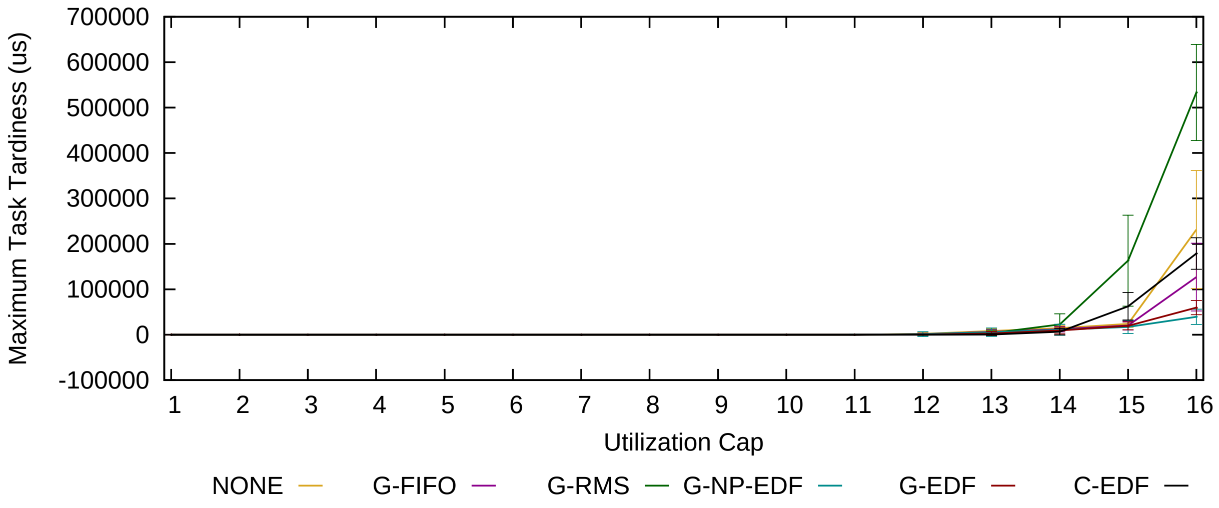


(b)

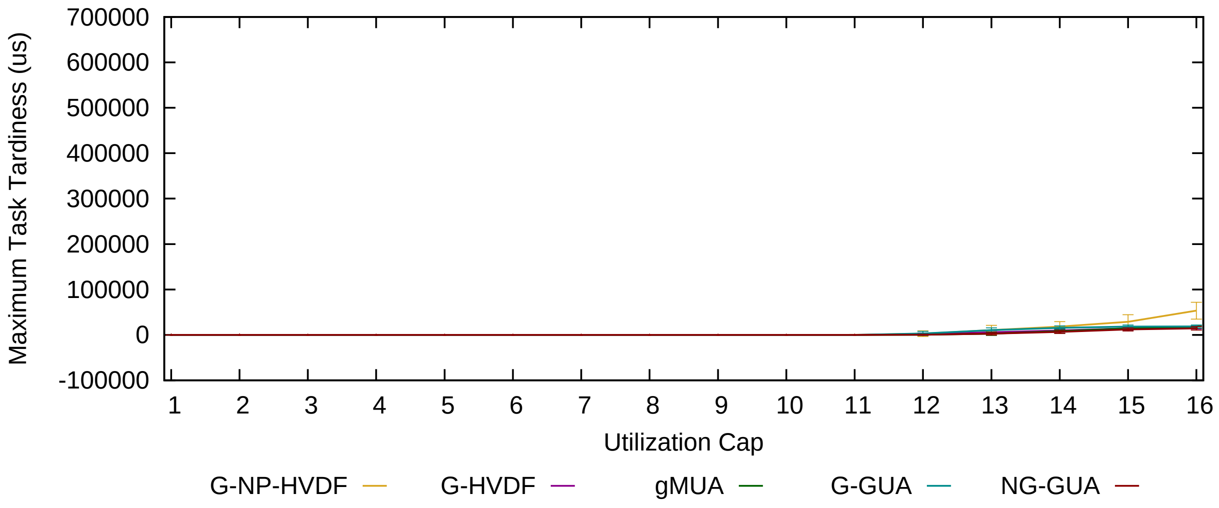


(c)

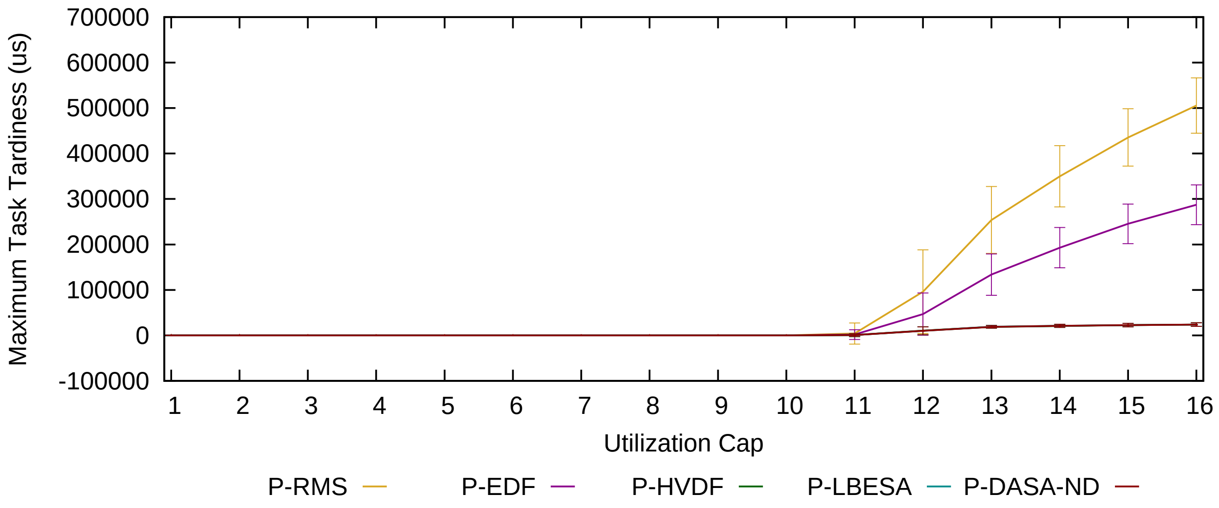
Figure D.7: 16-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

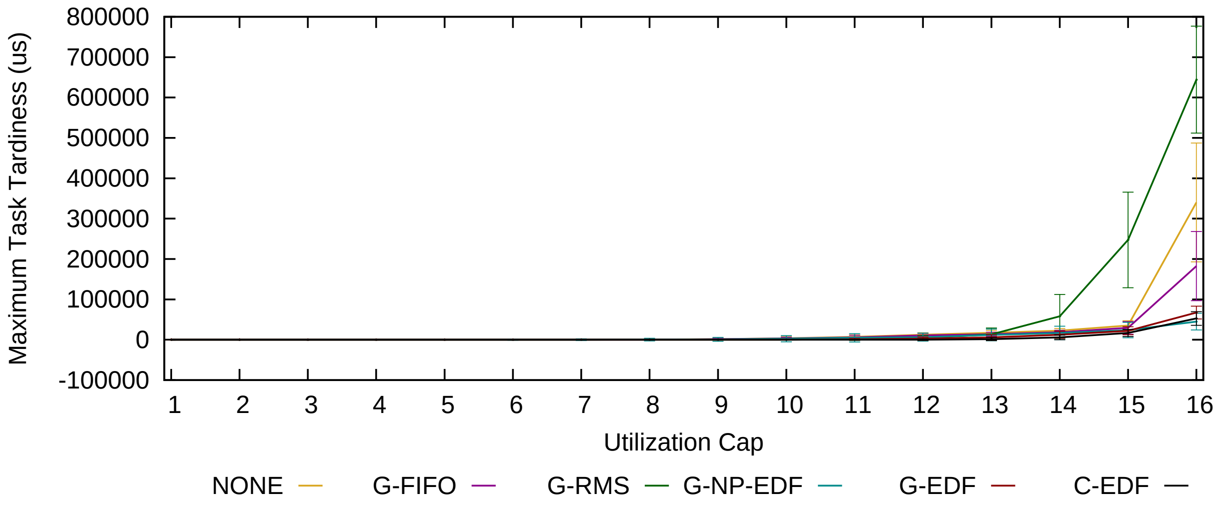


(b)

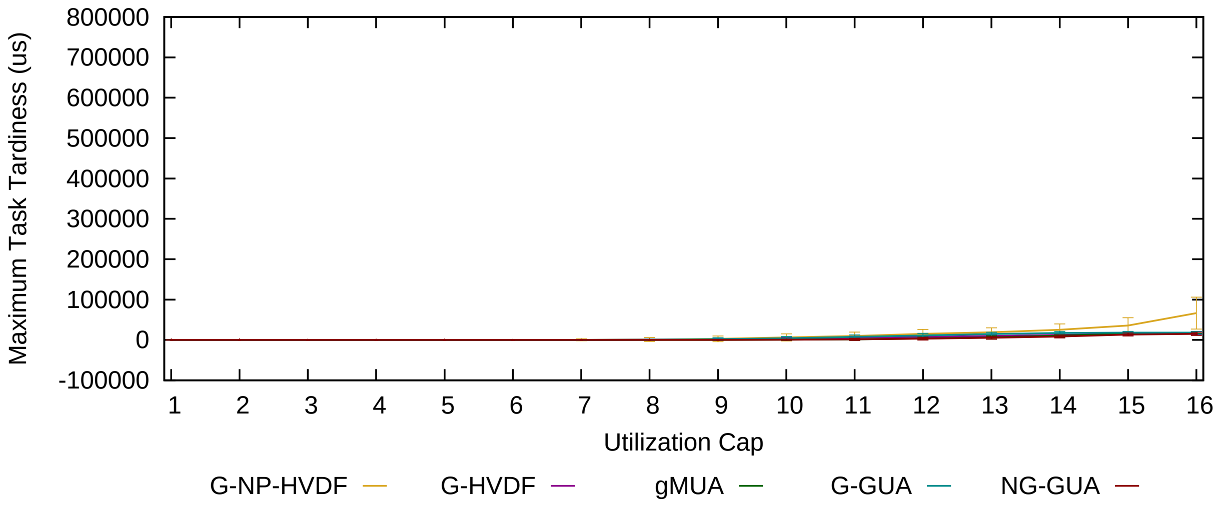


(c)

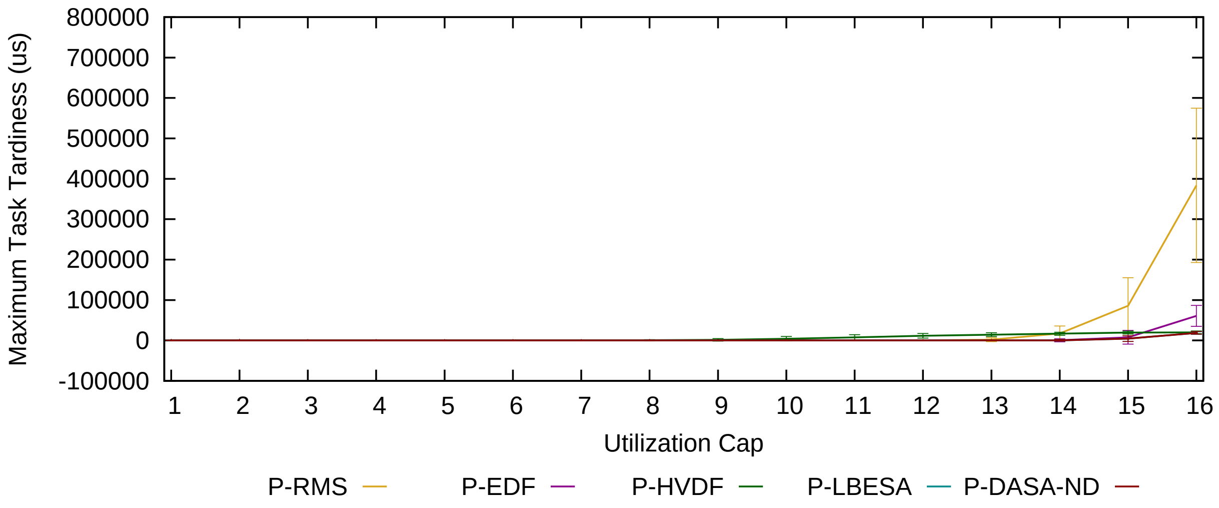
Figure D.8: 16-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

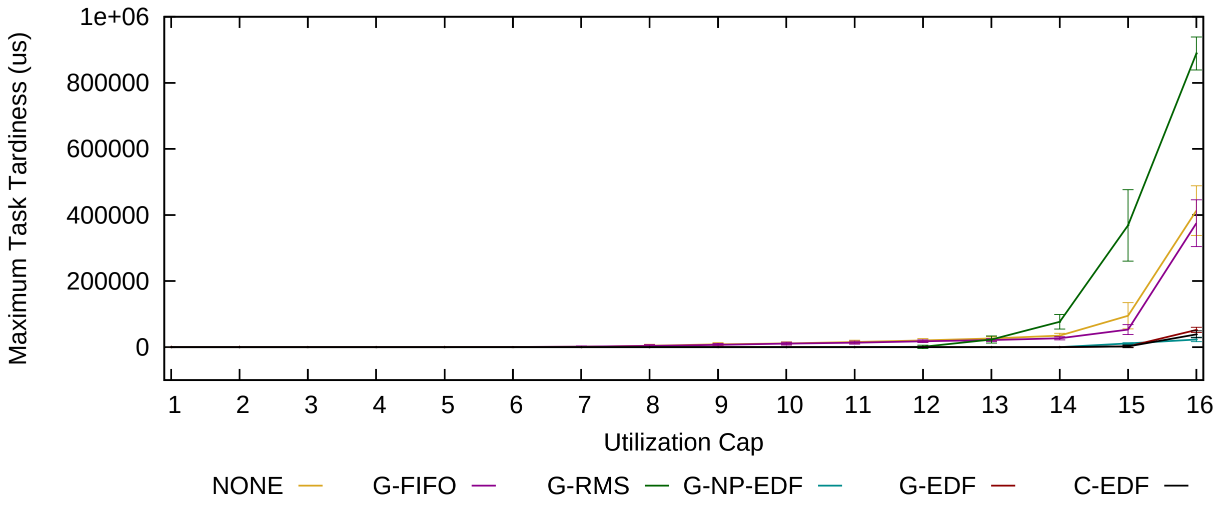


(b)

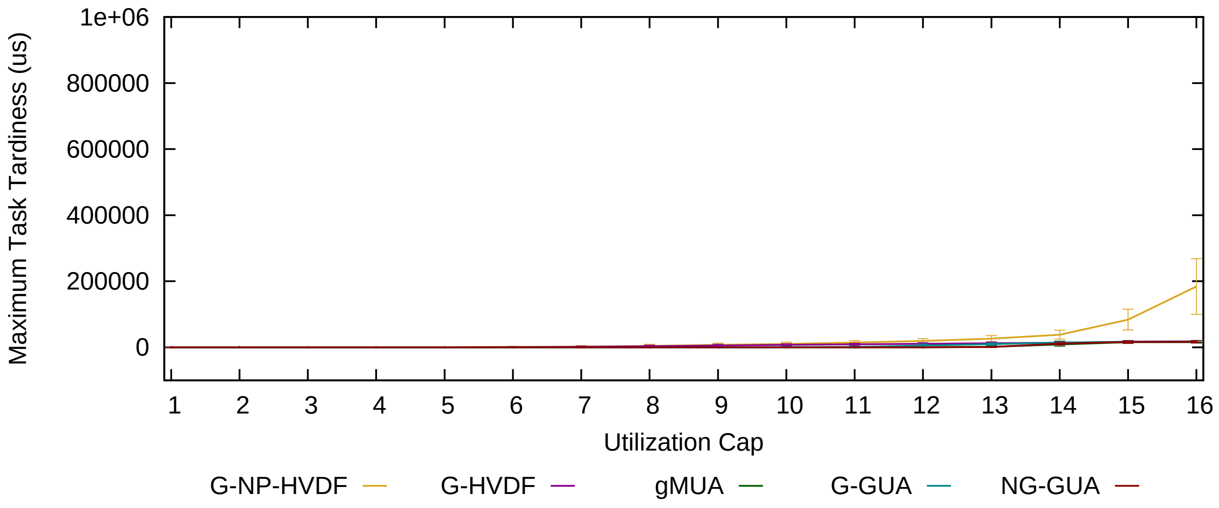


(c)

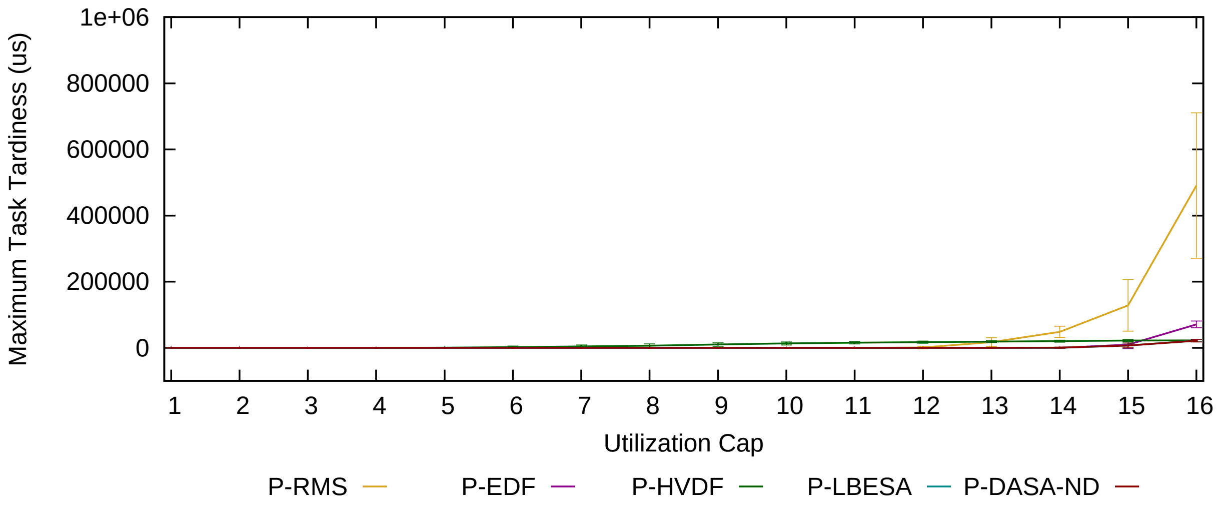
Figure D.9: 16-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

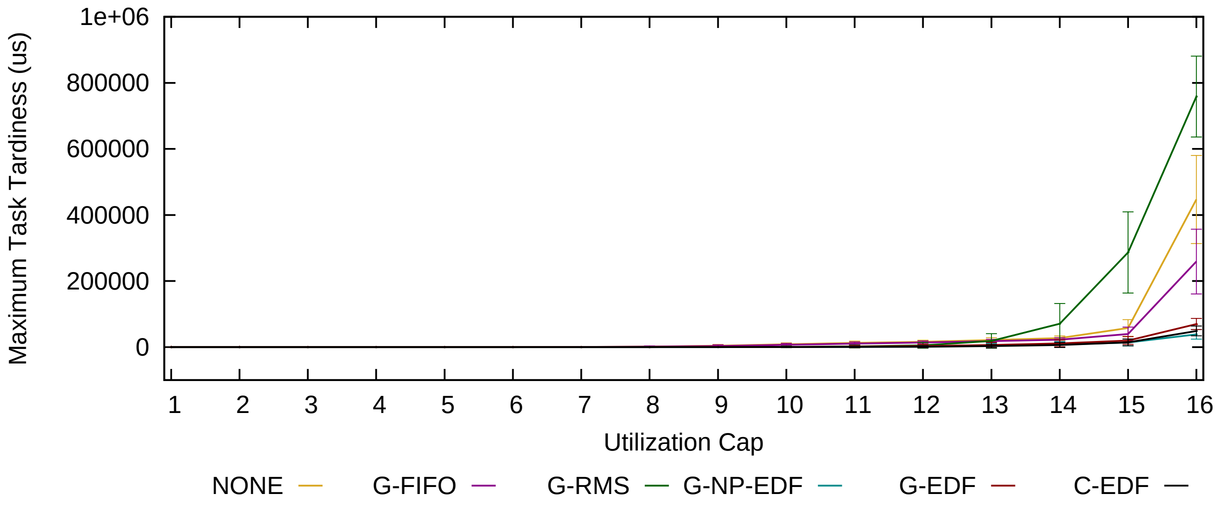


(b)

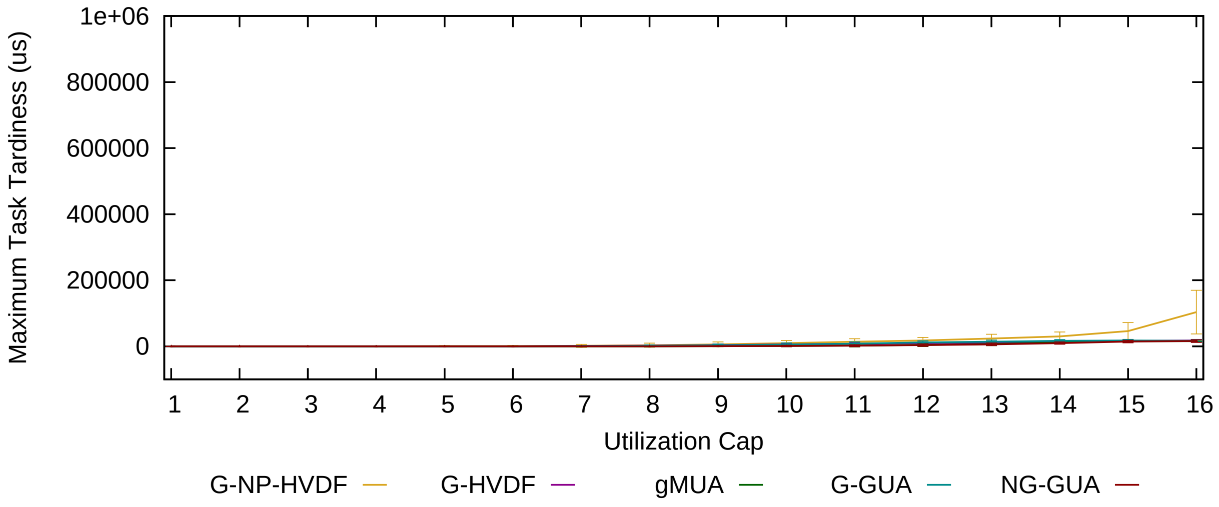


(c)

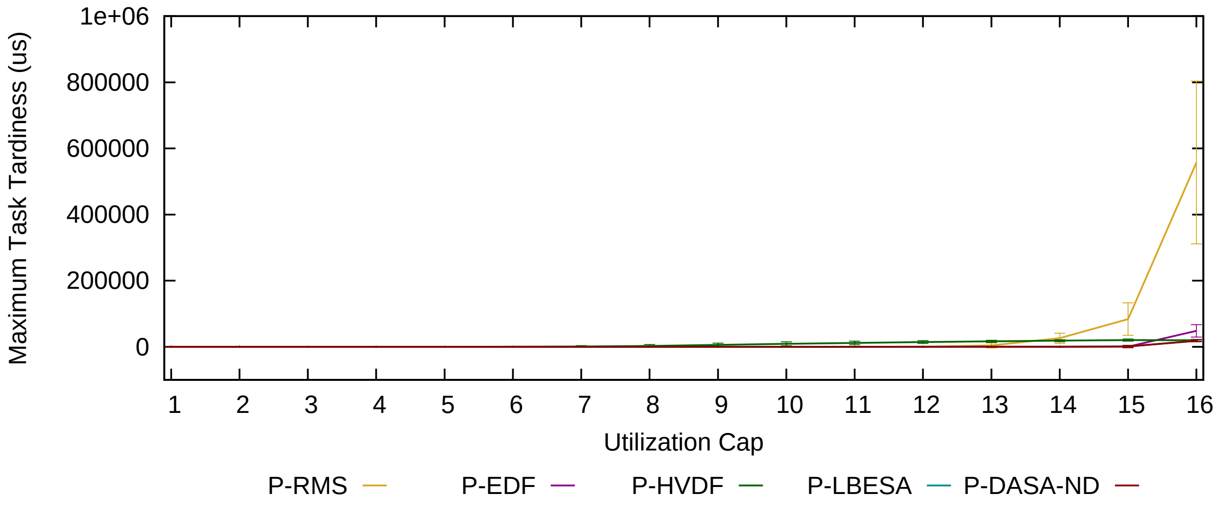
Figure D.10: 16-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

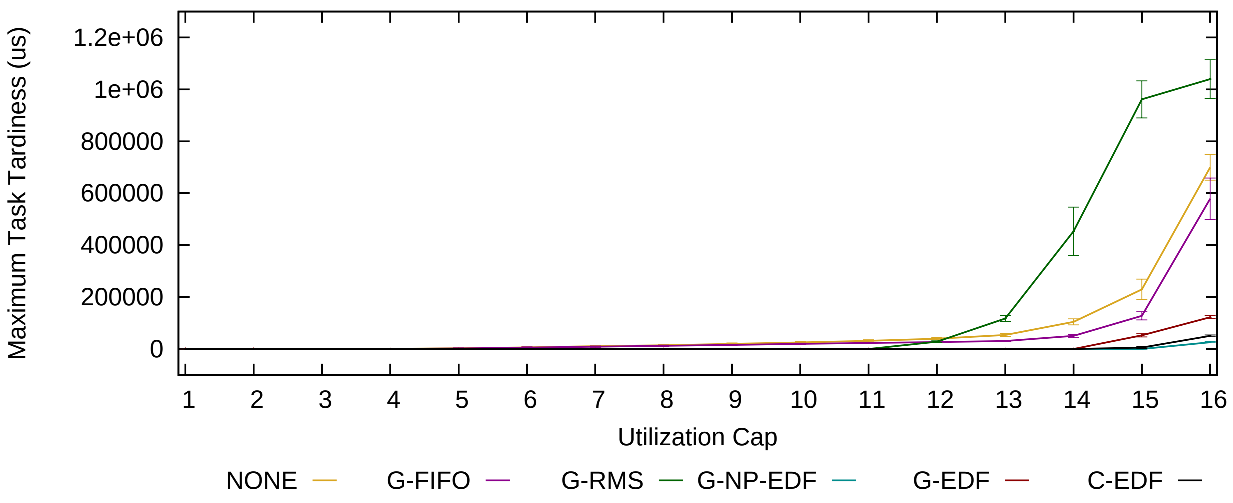


(b)

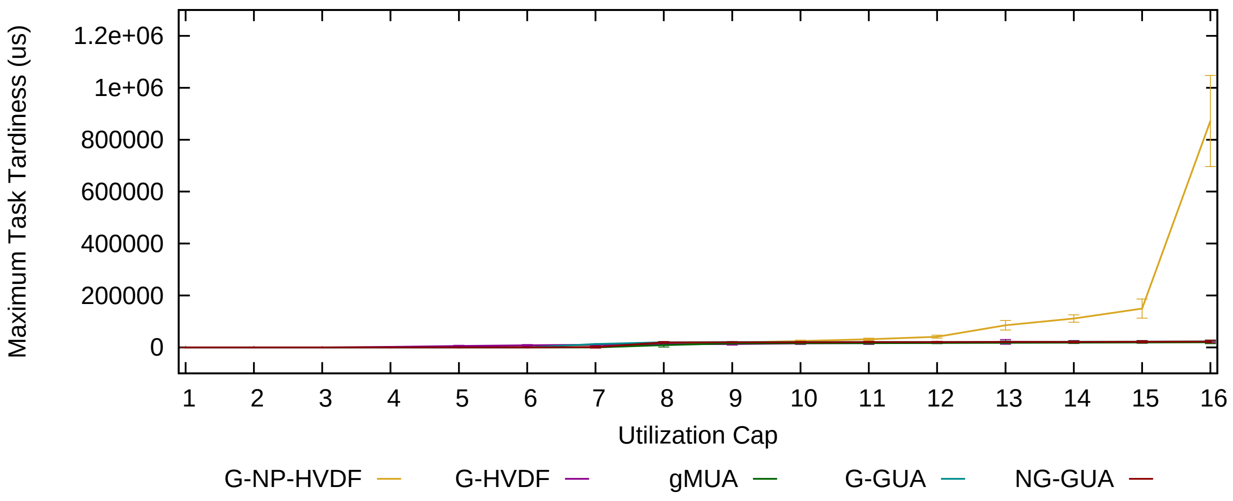


(c)

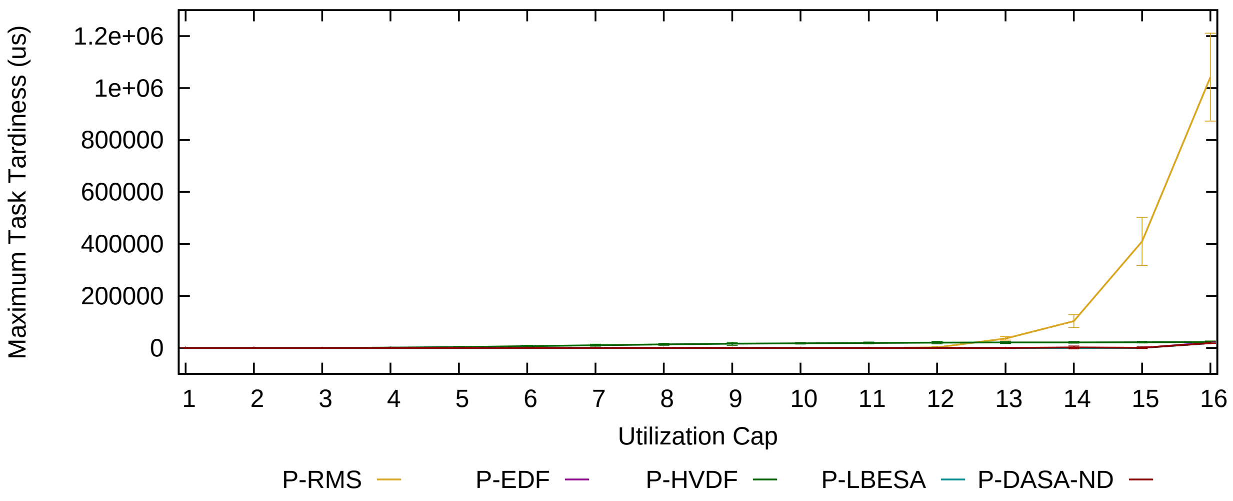
Figure D.11: 16-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)

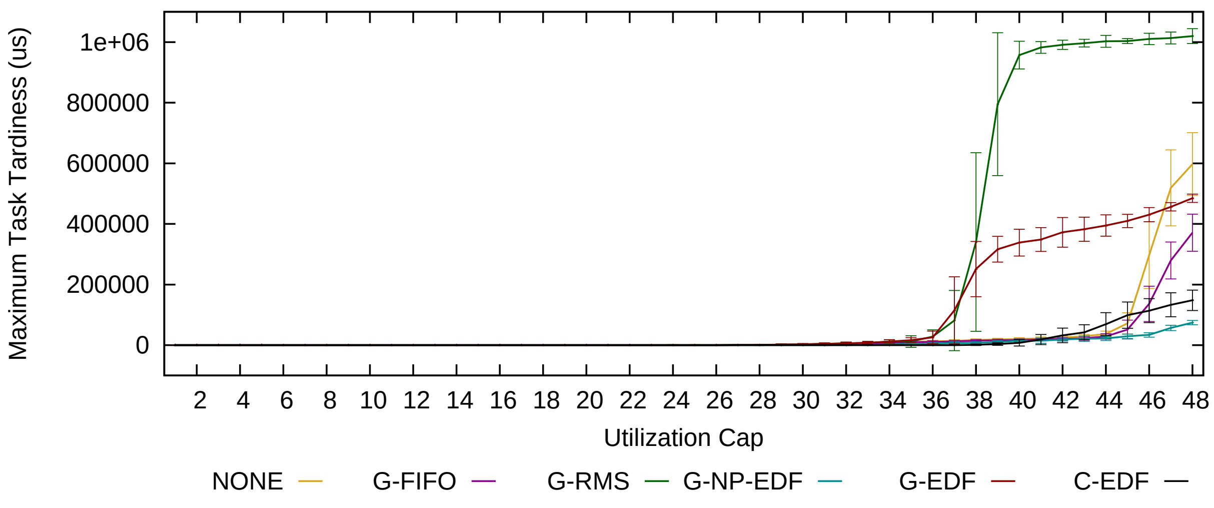


(b)

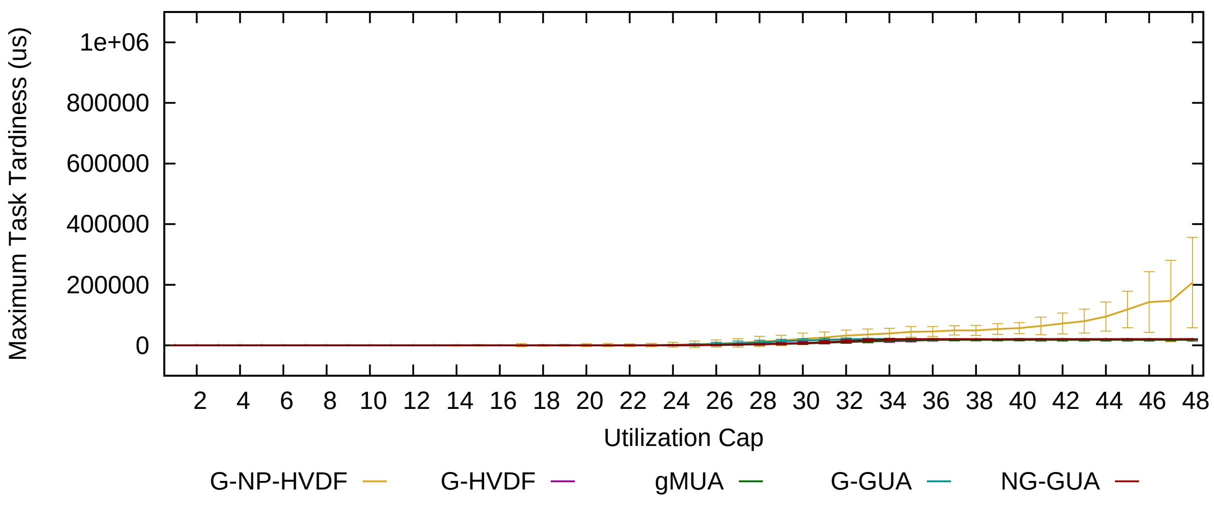


(c)

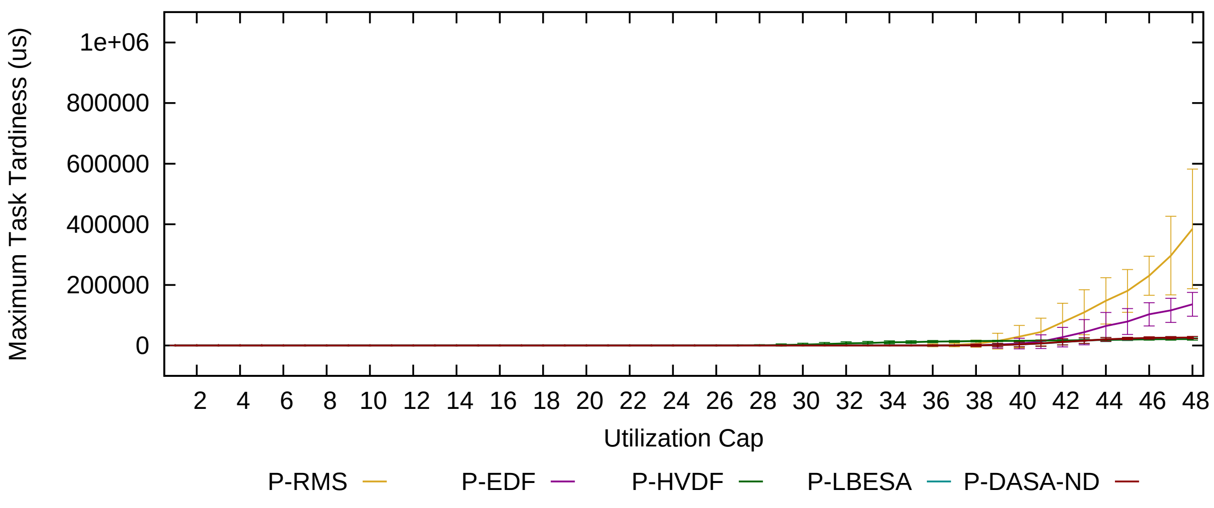
Figure D.12: 16-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions



(a)

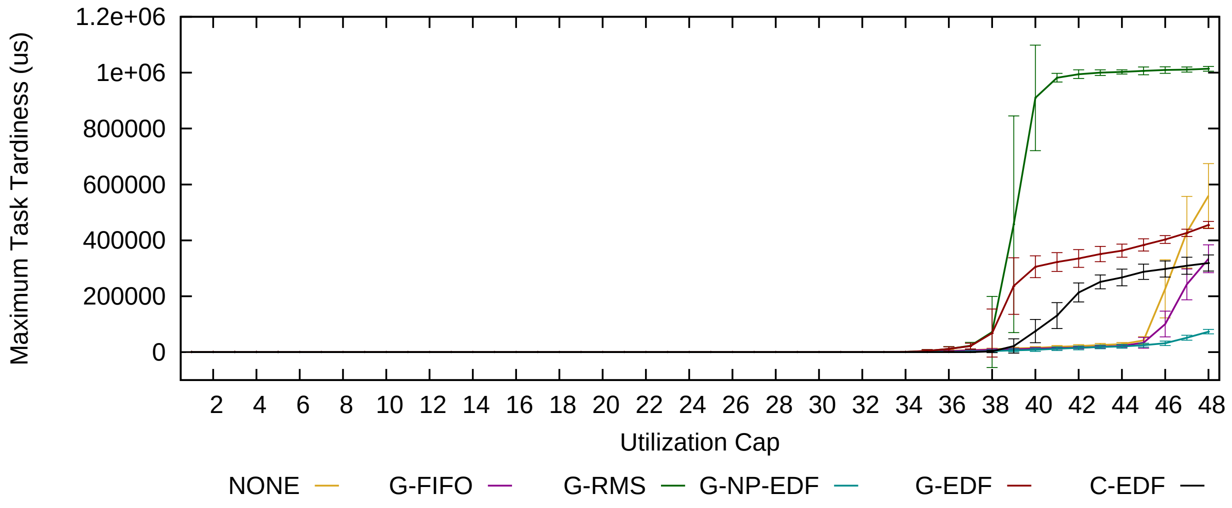


(b)

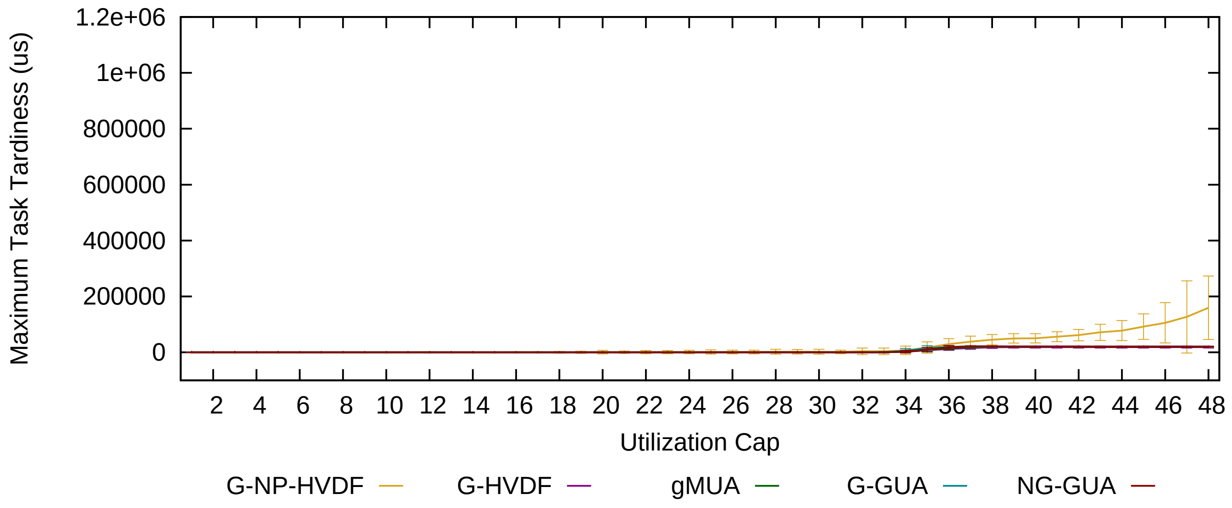


(c)

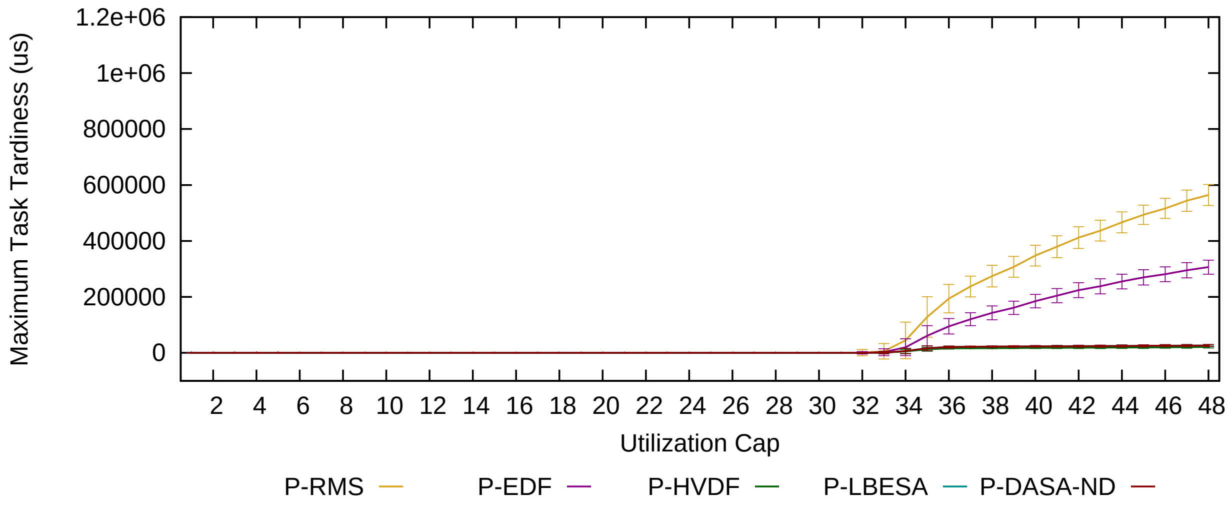
Figure D.13: 48-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy bimodal per-task weight distributions



(a)

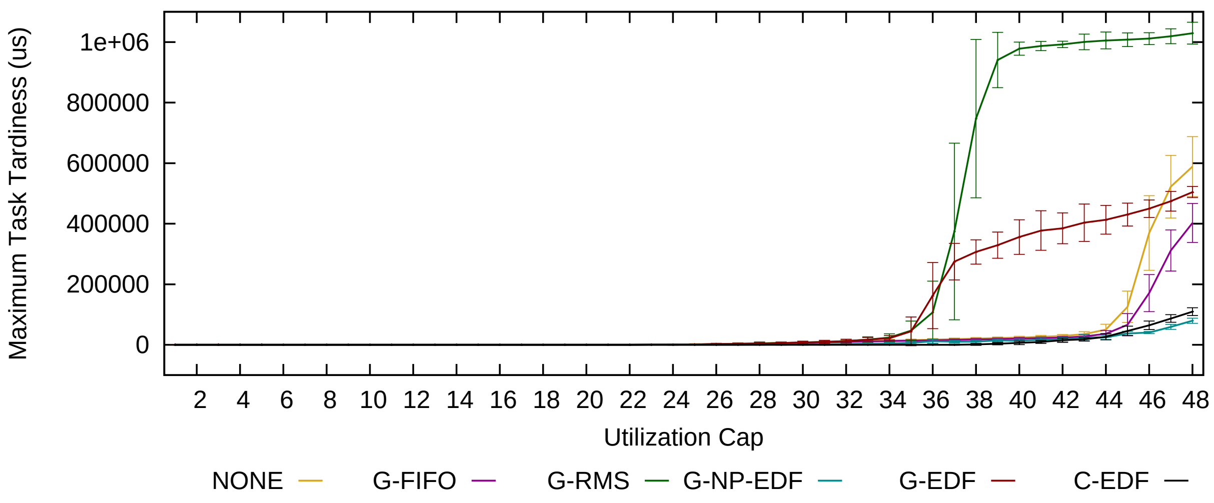


(b)

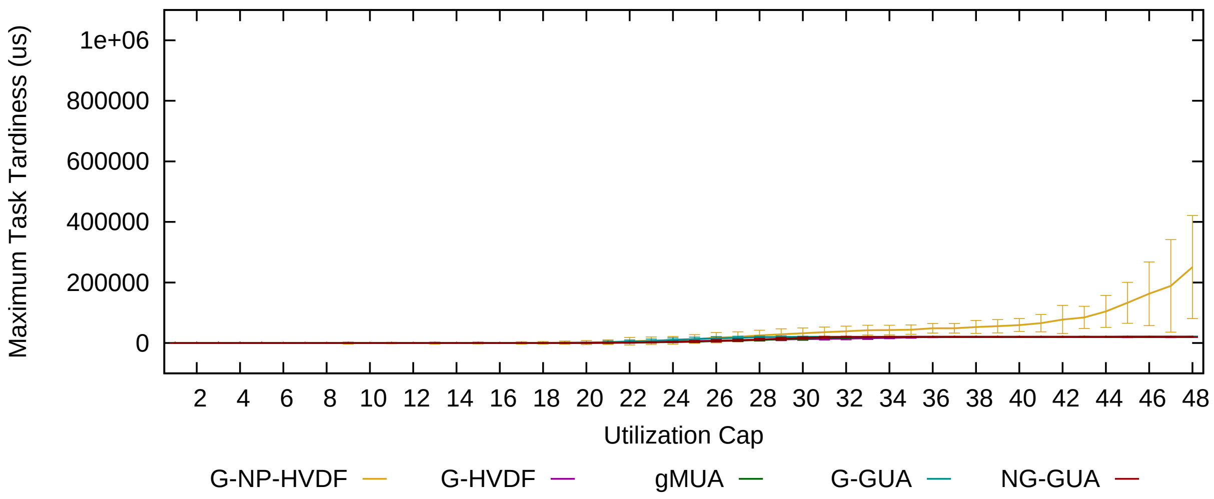


(c)

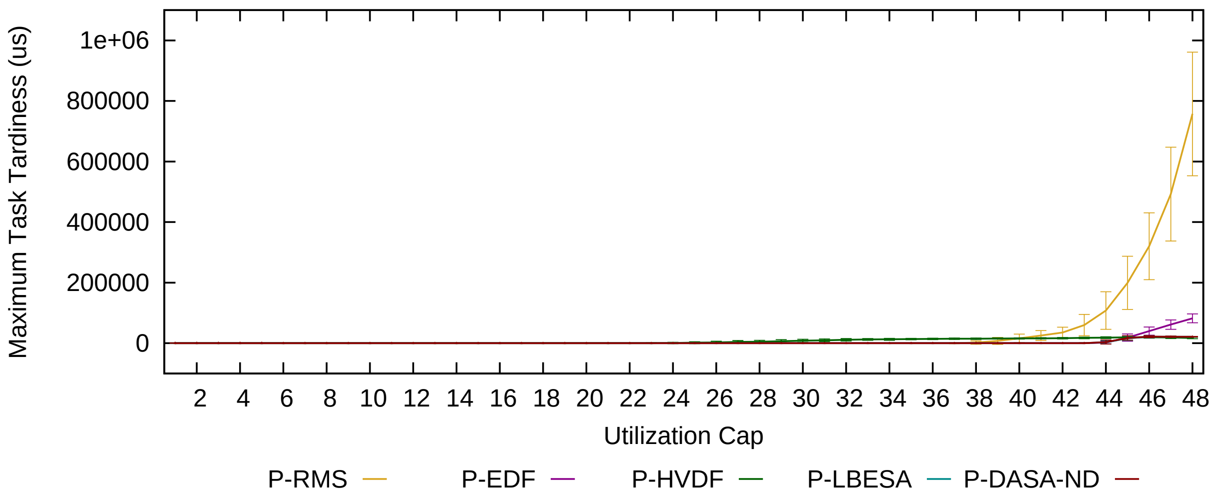
Figure D.14: 48-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under heavy uniform per-task weight distributions



(a)

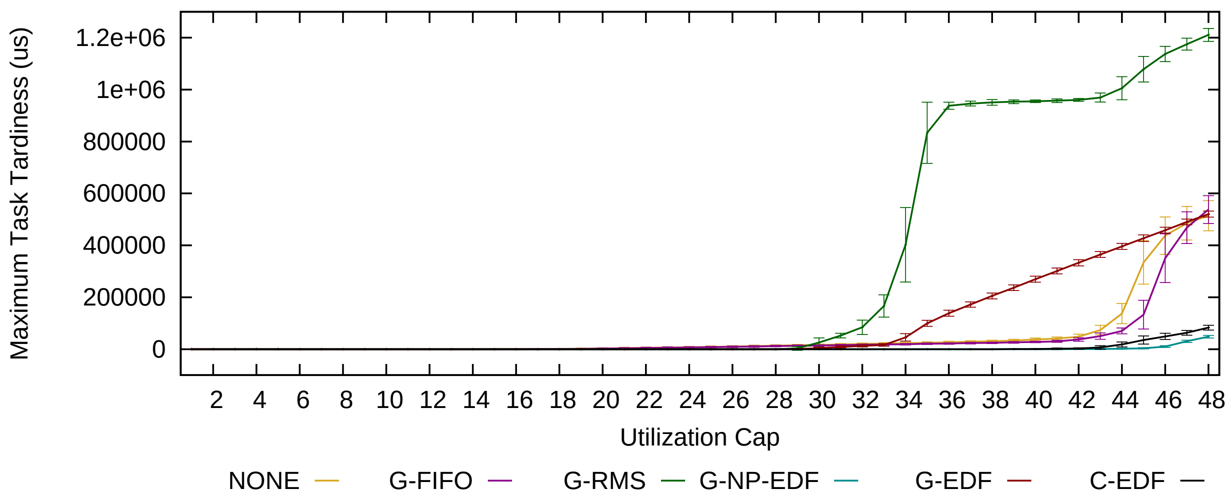


(b)

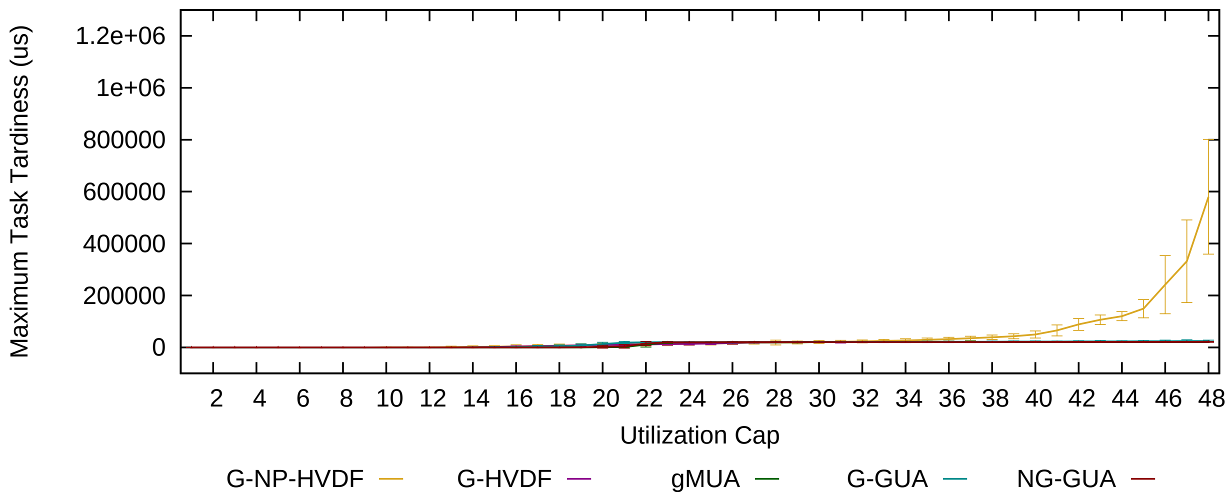


(c)

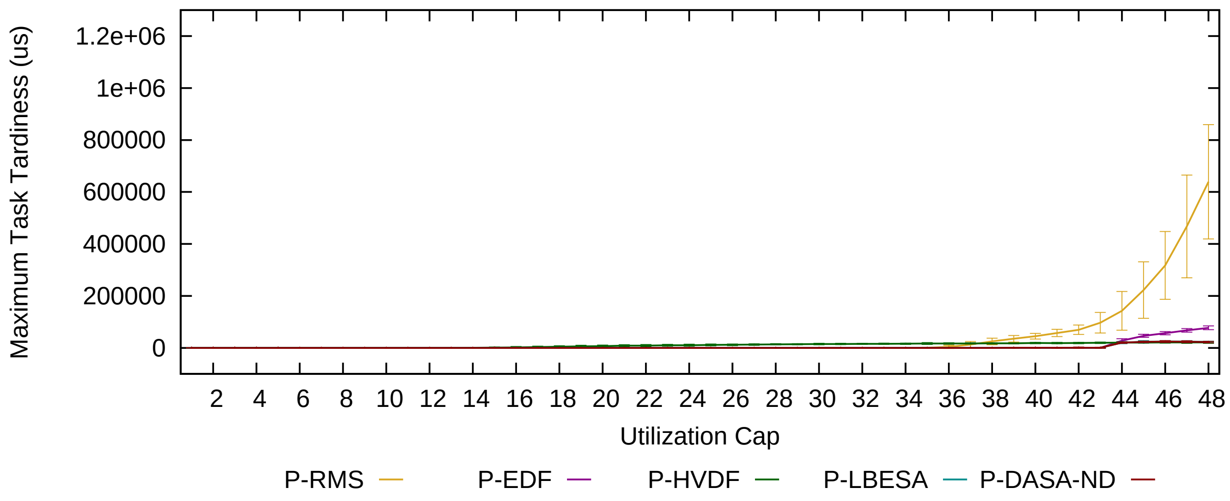
Figure D.15: 48-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium bimodal per-task weight distributions



(a)

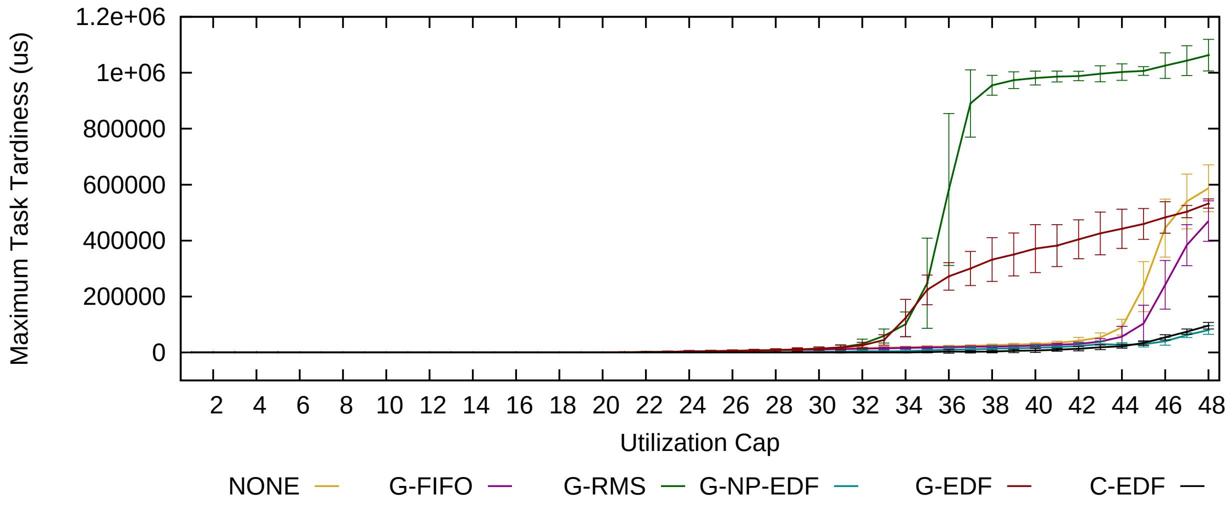


(b)

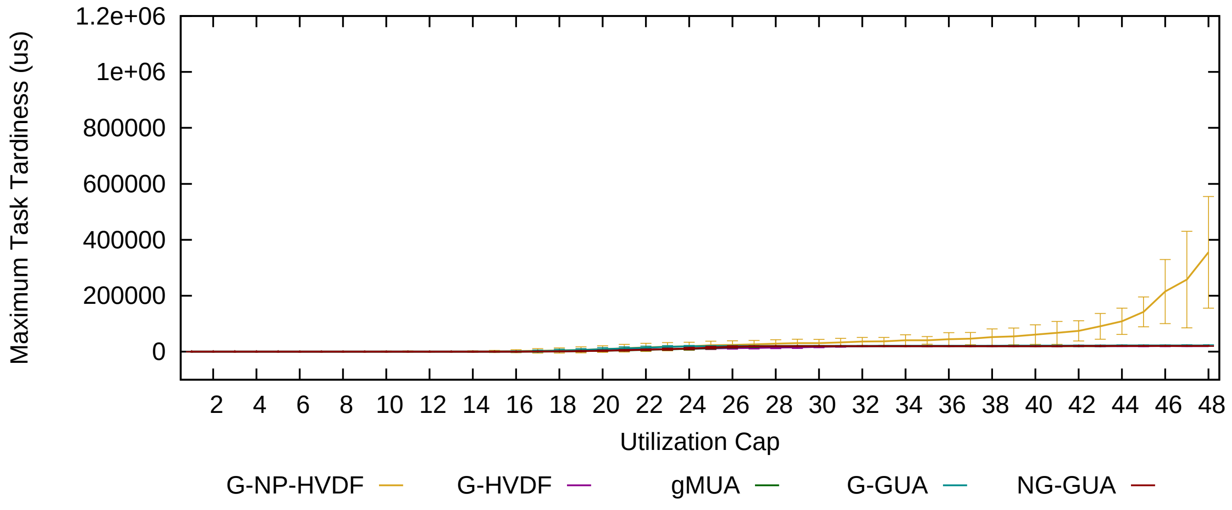


(c)

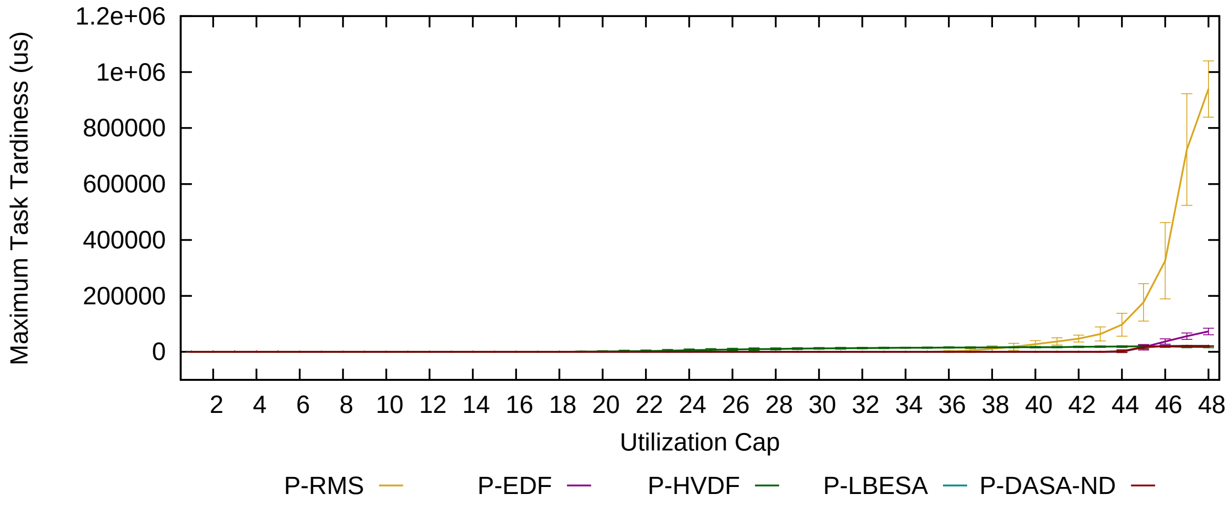
Figure D.16: 48-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under medium uniform per-task weight distributions



(a)

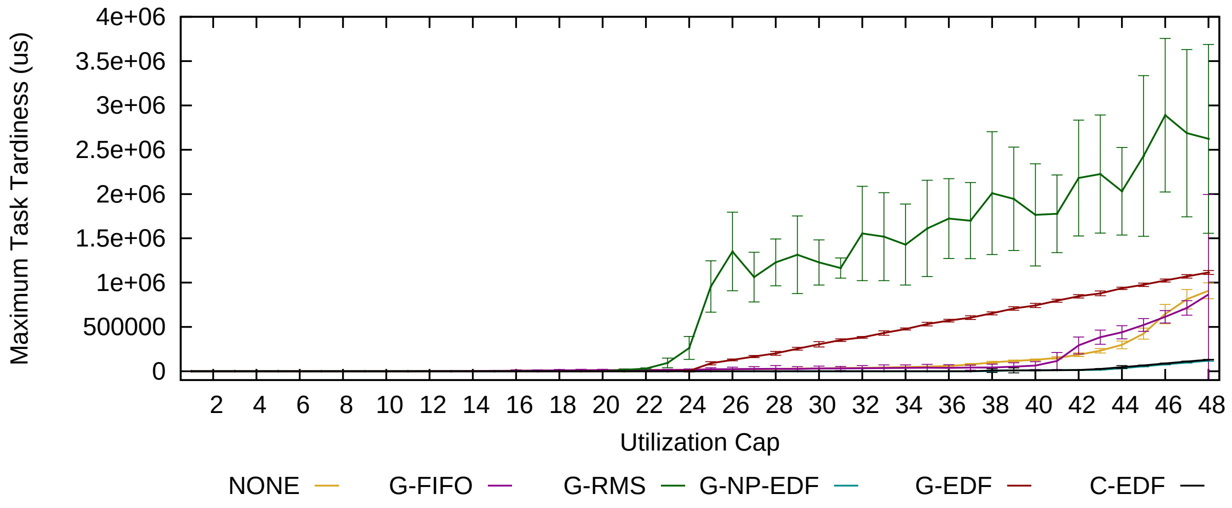


(b)

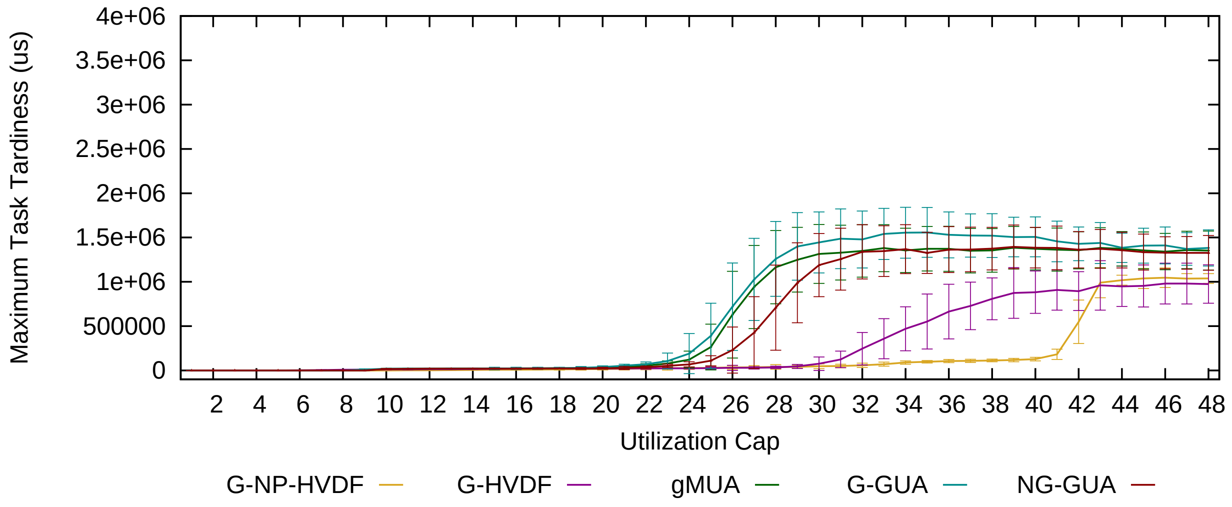


(c)

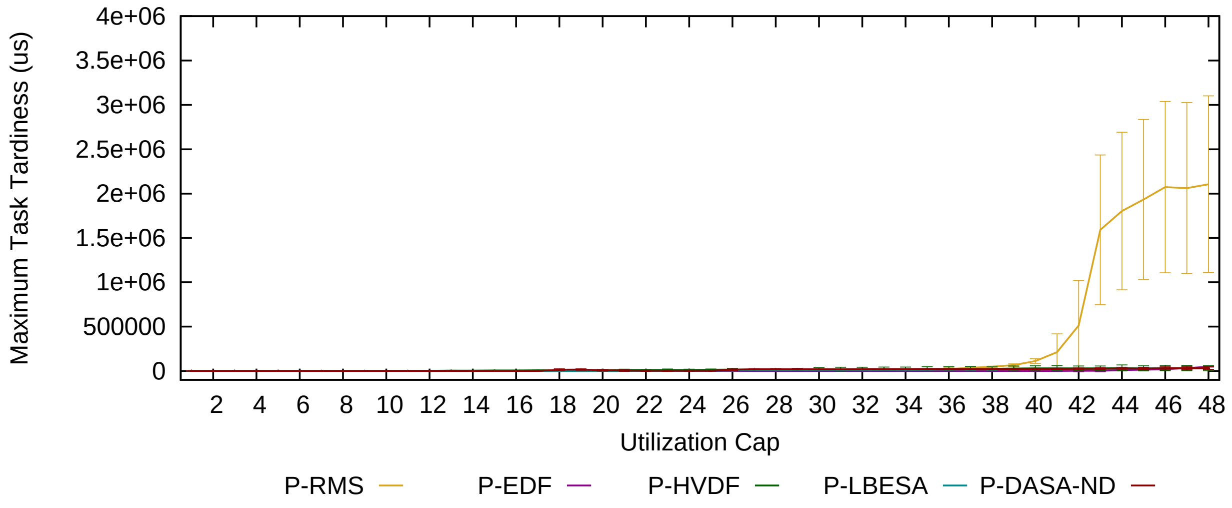
Figure D.17: 48-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light bimodal per-task weight distributions



(a)



(b)



(c)

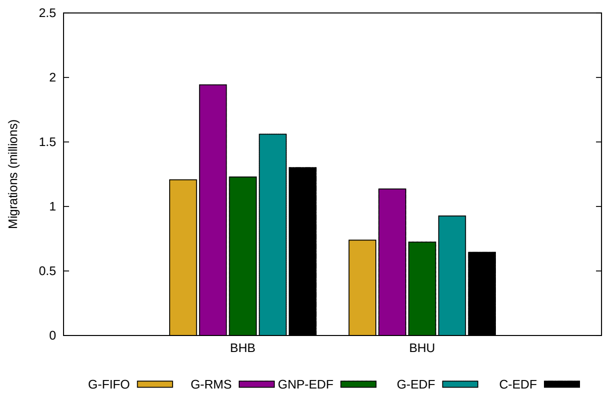
Figure D.18: 48-Core tardiness results for (a) traditional global, (b) global utility accrual, and (c) partitioned algorithms under light uniform per-task weight distributions

Appendix E

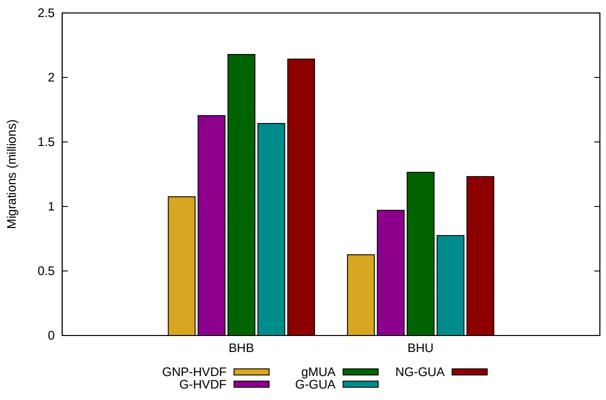
Complete Migration and Abortion Results

This appendix provides our complete migration and abortion results. It contains 24 plots grouped in 4 figures by machine and metric. It is organized as follows:

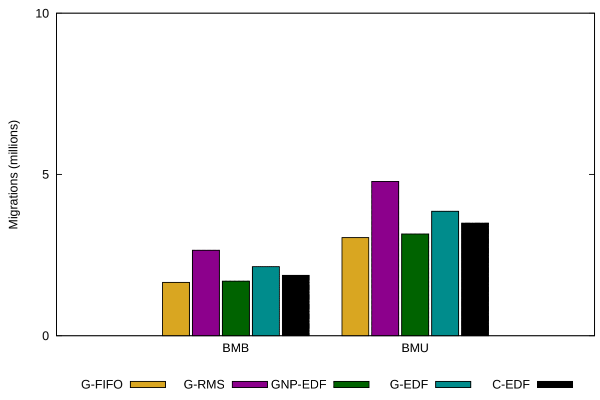
- Figure E.1 shows migration results for our 8-core platform
- Figure E.2 shows migration results for our 16-core platform
- Figure E.3 shows migration results for our 48-core platform
- Figure E.4 shows abort counts for all of our platforms



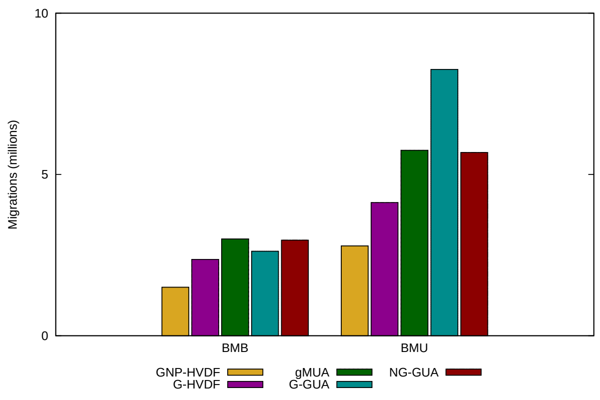
(a)



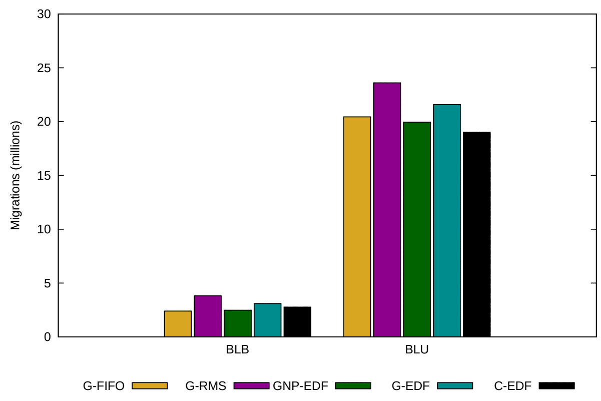
(b)



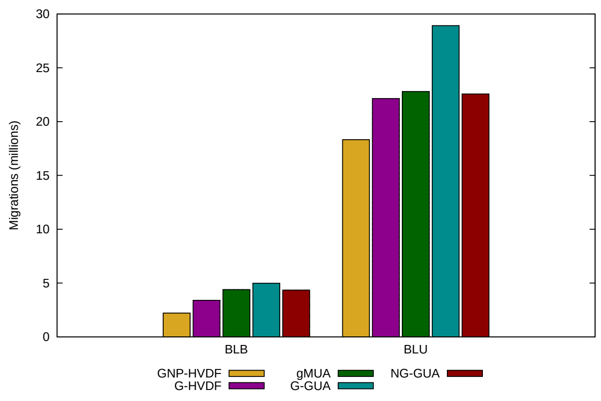
(c)



(d)

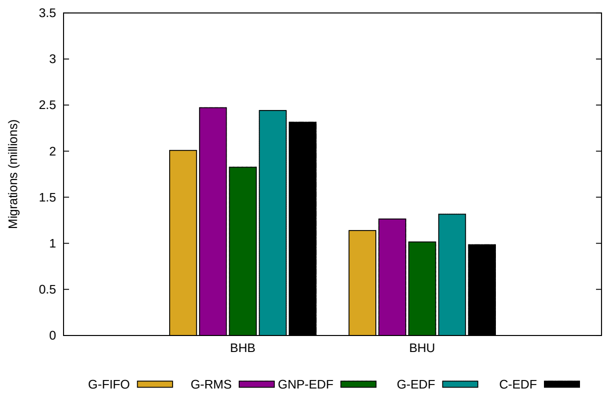


(e)

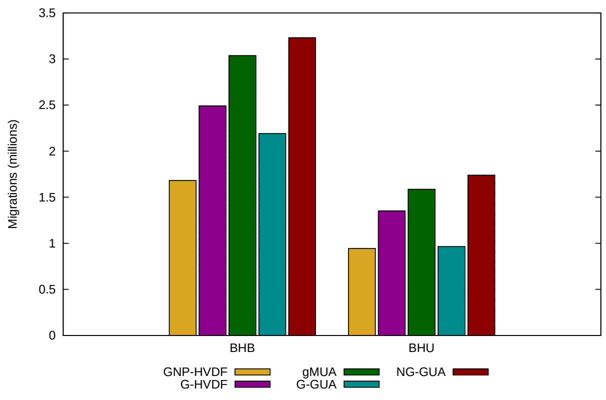


(f)

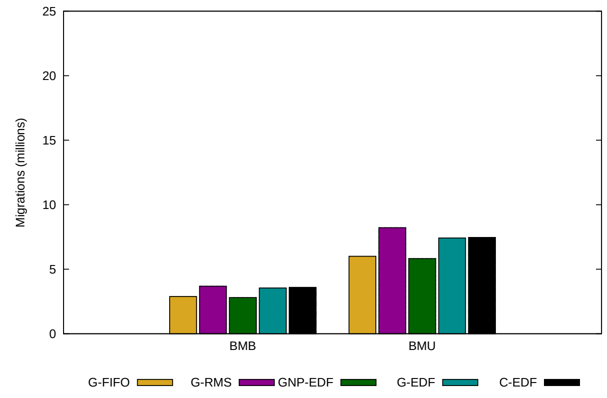
Figure E.1: 8-Core migration results for (a) heavy (c) medium and (e) light distributions under traditional algorithms and (b) heavy (d) medium and (f) light distributions under utility accrual algorithms



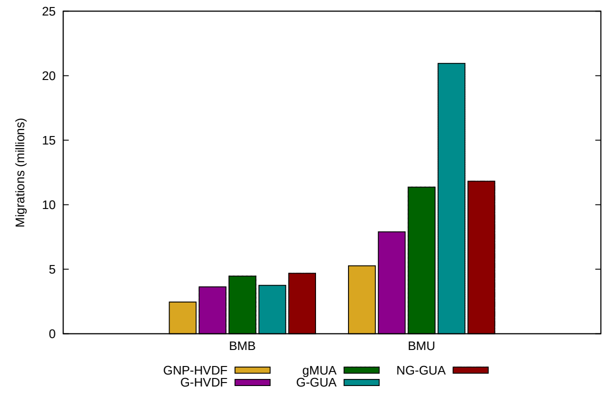
(a)



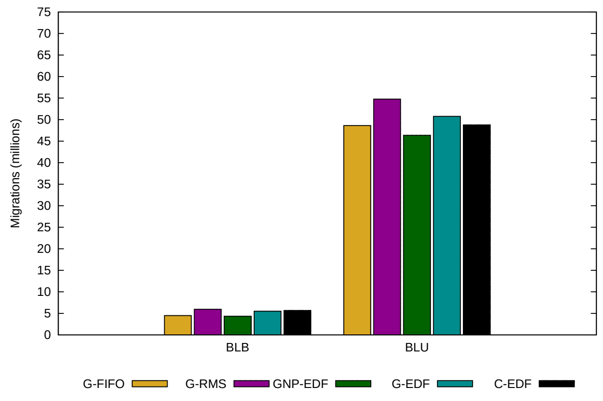
(b)



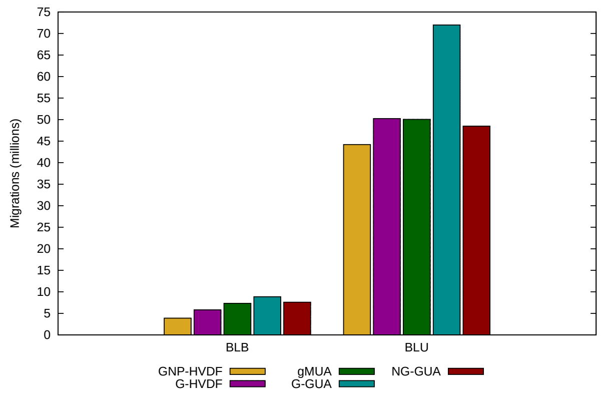
(c)



(d)

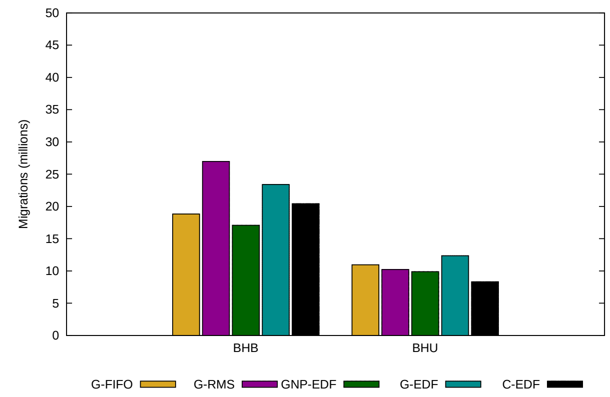


(e)

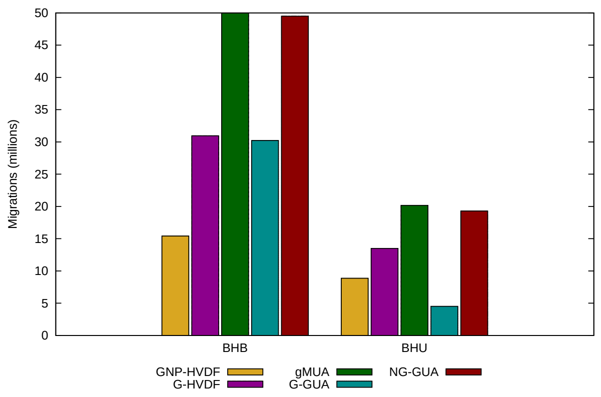


(f)

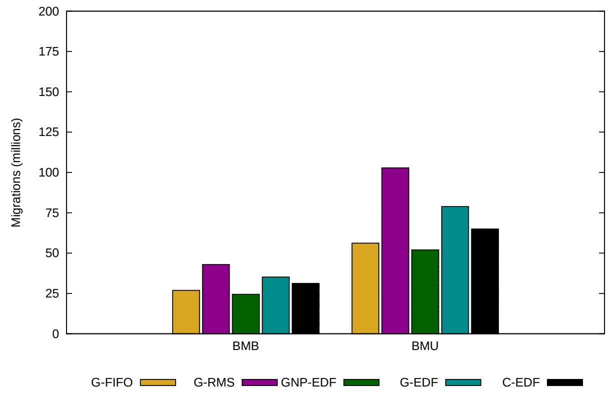
Figure E.2: 16-Core migration results for (a) heavy (c) medium and (e) light distributions under traditional algorithms and (b) heavy (d) medium and (f) light distributions under utility accrual algorithms



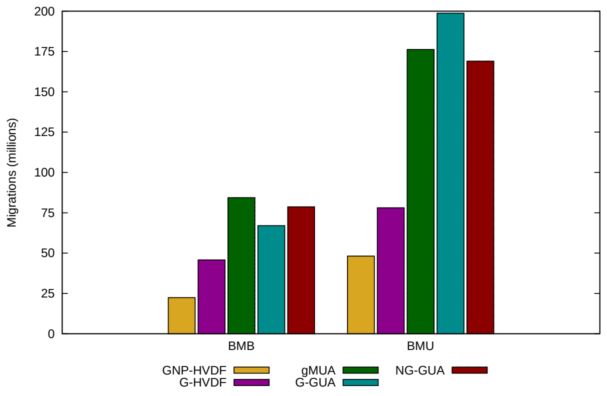
(a)



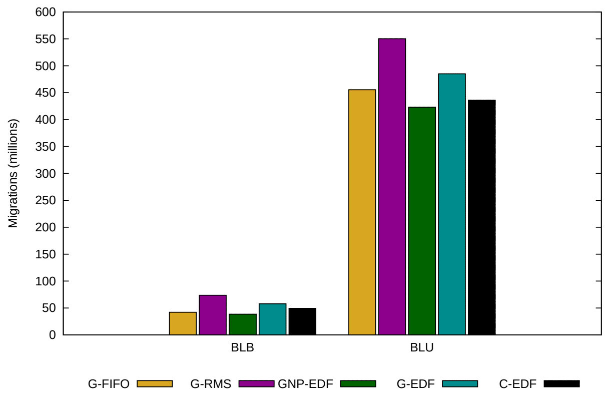
(b)



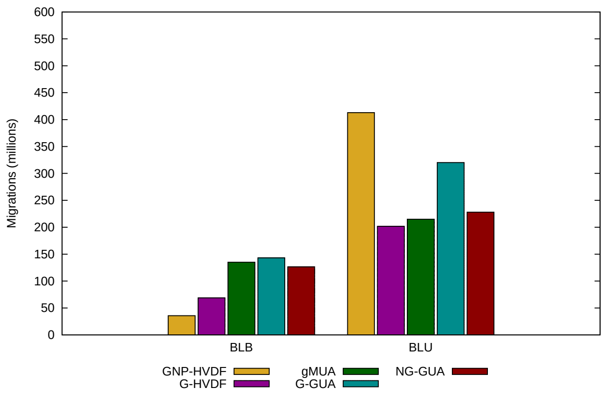
(c)



(d)

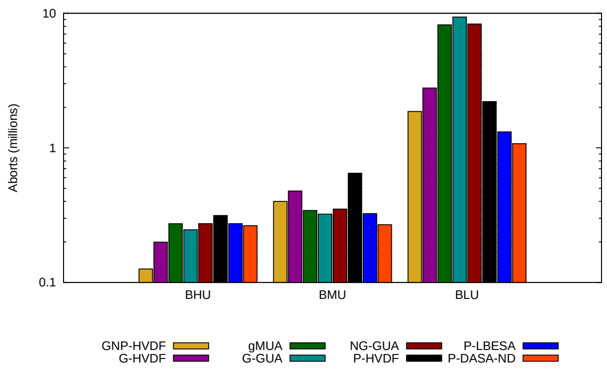


(e)

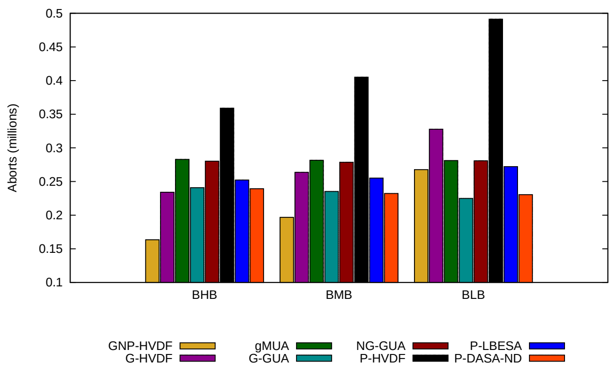


(f)

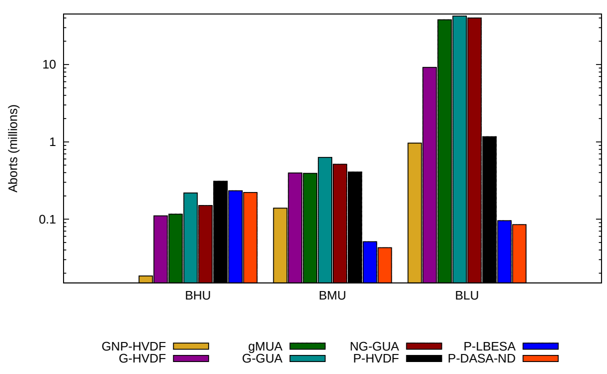
Figure E.3: 48-Core migration results for (a) heavy (c) medium and (e) light distributions under traditional algorithms and (b) heavy (d) medium and (f) light distributions under utility accrual algorithms



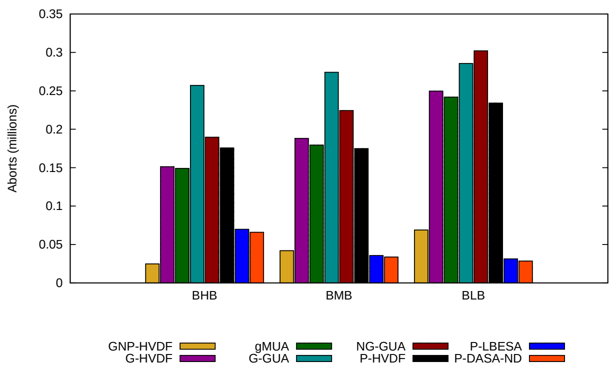
(a)



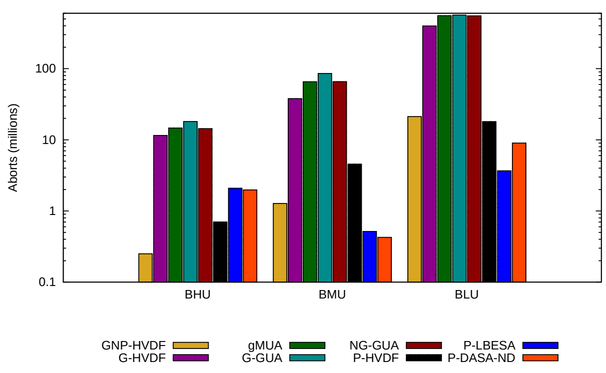
(b)



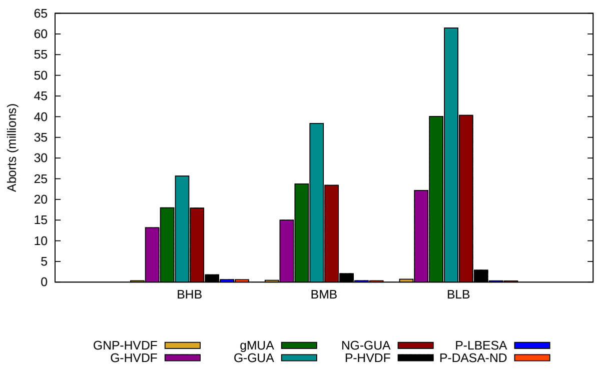
(c)



(d)



(e)



(f)

Figure E.4: Abortion results for various uniform per-task weight distributions on (a) 8 cores (c) 16 cores and (e) 48 cores and results for various bimodal per-task weight distributions on (b) 8 cores (d) 16 cores and (f) 48 cores

Appendix F

Complete Scheduling Statistics

This appendix provides the complete scheduling statistics for each scheduling algorithm, distribution, and platform in tabular form. It is organized as follows:

- Table F.1 shows scheduling statistics for G-FIFO on our 8-core platform under various load distributions
- Table F.2 shows scheduling statistics for G-NP-EDF on our 8-core platform under various load distributions
- Table F.3 shows scheduling statistics for G-RMS on our 8-core platform under various load distributions
- Table F.4 shows scheduling statistics for G-EDF on our 8-core platform under various load distributions
- Table F.5 shows scheduling statistics for G-NP-HVDF on our 8-core platform under various load distributions
- Table F.6 shows scheduling statistics for G-HVDF on our 8-core platform under various load distributions
- Table F.7 shows scheduling statistics for gMUA on our 8-core platform under various load distributions
- Table F.8 shows scheduling statistics for NG-GUA on our 8-core platform under various load distributions
- Table F.9 shows scheduling statistics for G-GUA on our 8-core platform under various load distributions

- Table F.10 shows scheduling statistics for P-RMS on our 8-core platform under various load distributions
- Table F.11 shows scheduling statistics for P-EDF on our 8-core platform under various load distributions
- Table F.12 shows scheduling statistics for P-HVDF on our 8-core platform under various load distributions
- Table F.13 shows scheduling statistics for P-LBESA on our 8-core platform under various load distributions
- Table F.14 shows scheduling statistics for P-DASA-ND on our 8-core platform under various load distributions
- Table F.15 shows scheduling statistics for C-EDF on our 8-core platform under various load distributions
- Table F.16 shows scheduling statistics for G-FIFO on our 16-core platform under various load distributions
- Table F.17 shows scheduling statistics for G-NP-EDF on our 16-core platform under various load distributions
- Table F.18 shows scheduling statistics for G-RMS on our 16-core platform under various load distributions
- Table F.19 shows scheduling statistics for G-EDF on our 16-core platform under various load distributions
- Table F.20 shows scheduling statistics for G-NP-HVDF on our 16-core platform under various load distributions
- Table F.21 shows scheduling statistics for G-HVDF on our 16-core platform under various load distributions
- Table F.22 shows scheduling statistics for gMUA on our 16-core platform under various load distributions
- Table F.23 shows scheduling statistics for NG-GUA on our 16-core platform under various load distributions
- Table F.24 shows scheduling statistics for G-GUA on our 16-core platform under various load distributions
- Table F.25 shows scheduling statistics for P-RMS on our 16-core platform under various load distributions

- Table F.26 shows scheduling statistics for P-EDF on our 16-core platform under various load distributions
- Table F.27 shows scheduling statistics for P-HVDF on our 16-core platform under various load distributions
- Table F.28 shows scheduling statistics for P-LBESA on our 16-core platform under various load distributions
- Table F.29 shows scheduling statistics for P-DASA-ND on our 16-core platform under various load distributions
- Table F.30 shows scheduling statistics for C-EDF on our 16-core platform under various load distributions
- Table F.31 shows scheduling statistics for G-FIFO on our 48-core platform under various load distributions
- Table F.32 shows scheduling statistics for G-NP-EDF on our 48-core platform under various load distributions
- Table F.33 shows scheduling statistics for G-RMS on our 48-core platform under various load distributions
- Table F.34 shows scheduling statistics for G-EDF on our 48-core platform under various load distributions
- Table F.35 shows scheduling statistics for G-NP-HVDF on our 48-core platform under various load distributions
- Table F.36 shows scheduling statistics for G-HVDF on our 48-core platform under various load distributions
- Table F.37 shows scheduling statistics for gMUA on our 48-core platform under various load distributions
- Table F.38 shows scheduling statistics for NG-GUA on our 48-core platform under various load distributions
- Table F.39 shows scheduling statistics for G-GUA on our 48-core platform under various load distributions
- Table F.40 shows scheduling statistics for P-RMS on our 48-core platform under various load distributions
- Table F.41 shows scheduling statistics for P-EDF on our 48-core platform under various load distributions

- Table F.42 shows scheduling statistics for P-HVDF on our 48-core platform under various load distributions
- Table F.43 shows scheduling statistics for P-LBESA on our 48-core platform under various load distributions
- Table F.44 shows scheduling statistics for P-DASA-ND on our 48-core platform under various load distributions
- Table F.45 shows scheduling statistics for C-EDF on our 48-core platform under various load distributions

Table F.1: G-FIFO scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	2819086	1958379	3578307	5744045	4809142	28865632
Block	0	0	0	0	0	0
Preschedule	6192271	5745601	6589119	7598483	7184551	18962606
Local	8686948	7497328	9738243	12731792	11435095	45032189
IPI Sent	585	618	696	807	754	921530
IPI Received	583	616	690	801	742	770632
IPI Missed	2	2	6	6	12	150898
Migrations	1206798	738521	1652724	3042886	2400078	20438504
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Aborts	0	0	0	0	0	0

Table F.2: G-NP-EDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	2818110	1958391	3576855	5741727	4807028	28860934
Block	0	0	0	0	0	0
Preschedule	6156781	5745335	6529951	7587191	7106629	18738446
Local	8667626	7518347	9700877	12708750	11368295	44472360
IPI Sent	602	591	630	478	518	288232
IPI Received	588	587	622	464	510	238334
IPI Missed	14	4	8	14	8	49898
Migrations	1228472	723748	1692460	3155239	2479302	19947961
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Aborts	0	0	0	0	0	0

Table F.3: G-RMS scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	11879755	10387500	13108191	16330400	15016091	50492656
Block	38760566	30258636	45328275	65123917	55828129	242086054
Preschedule	0	0	0	0	0	0
Local	38115094	29612445	45248211	67657275	56868994	266843572
IPI Sent	29133683	21893767	35133469	54046698	44847828	215751483
IPI Received	29131403	21893085	35129719	54035814	44840606	215576495
IPI Missed	2282	682	3750	10884	7222	174988
Migrations	1943027	1135483	2650260	4781877	3810060	23598759
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Aborts	0	0	0	0	0	0

Table F.4: G-EDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	11117837	9712829	12296928	15415267	14119566	46839128
Block	39922467	32270304	46110746	63203285	55409579	220408026
Preschedule	0	0	0	0	0	0
Local	40288154	32696652	47029585	66309868	57307061	242152664
IPI Sent	31680705	25198809	37405663	53707455	46029086	199720397
IPI Received	31679217	25198407	37403181	53700473	46024344	199635305
IPI Missed	1488	402	2482	6982	4742	85092
Migrations	1559957	926322	2142289	3859036	3090628	21586361
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Aborts	0	0	0	0	0	0

Table F.5: G-NP-HVDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	2817954	1957477	3576771	5740923	4807124	28858152
Block	0	0	0	0	0	0
Preschedule	6211352	5674614	6706191	8254920	7547877	24614876
Local	8604712	7369155	9700845	13010167	11527223	49877005
IPI Sent	704	706	707	908	2732	187154
IPI Received	700	700	697	886	2670	147454
IPI Missed	4	6	10	22	62	39700
Migrations	1075128	625606	1502871	2786301	2211382	18315194
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Aborts	163532	126018	196734	200106	267742	1864600

Table F.6: G-HVDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	11054823	9591509	12276676	15332030	14158194	46638114
Block	35394892	27898060	41552922	60738473	51585248	219382512
Preschedule	196202	160854	229424	491776	308792	3100144
Local	35396034	28041832	42023644	63694434	53048526	244282778
IPI Sent	26905096	20707354	32495676	50811311	41738582	195189868
IPI Received	26902714	20706700	32491362	50800411	41730818	195022216
IPI Missed	2382	654	4314	10900	7764	167652
Migrations	1703974	969847	2364878	4129364	3400232	22144814
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	234048	199449	263668	478449	327710	2786704

Table F.7: gMUA scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	10938324	9425646	12162098	15468404	14124926	37770085
Block	38119758	29570032	44963721	63730007	55468470	166430256
Preschedule	271700	256142	274084	346866	278710	8828680
Local	38431088	29908829	45936590	67270044	57551042	191183605
IPI Sent	29796728	22548050	36128412	53701978	45709373	147687834
IPI Received	29793934	22547350	36123726	53687306	45699711	147563212
IPI Missed	2794	700	4686	14672	9662	124622
Migrations	2177296	1264342	3002687	5750154	4399200	22793313
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	282850	273465	281650	342722	281188	8213574

Table F.8: NG-GUA scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	10918422	9412204	12170434	15456446	14117187	37510055
Block	38044511	29480585	45025888	63722679	55476060	165828165
Preschedule	269274	256232	271024	355471	277935	8950353
Local	38358098	29789671	45970222	67272123	57580728	190676316
IPI Sent	29722322	22432533	36153163	53702360	45730500	147279440
IPI Received	29719558	22431720	36147987	53686926	45720142	147157796
IPI Missed	2764	813	5176	15434	10358	121644
Migrations	2142001	1231025	2961955	5681157	4345871	22563665
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	280306	273598	278538	351282	280850	4162931

Table F.9: G-GUA scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	10980197	9446434	12348857	15471074	14197589	35217904
Block	36373687	28149835	43131693	64500137	54238600	153369213
Preschedule	210432	211137	209699	324384	213429	10073230
Local	35409028	27336863	42658800	68019288	55464302	177778915
IPI Sent	27021012	20123473	33188147	54442828	43898594	136651692
IPI Received	27017812	20122597	33182128	54421278	43886234	136501336
IPI Missed	3200	876	6019	21550	12360	150356
Migrations	1642280	774164	2618399	8257639	4983188	28913025
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	240908	246242	235301	321929	224964	9368928

Table F.10: P-RMS scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	9505767	7933408	10895253	15118681	13206440	57608774
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	0	0	0	0	0	0

Table F.11: P-EDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	9257512	7457912	10695502	14465679	12916969	53442235
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	0	0	0	0	0	0

Table F.12: P-HVDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	8952807	7335877	10298794	14136778	12492978	55717061
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	358972	314599	405081	648778	245609	2206516

Table F.13: P-LBESA scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	9487171	7365454	11146298	15576333	13689131	59379494
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	252326	273897	255162	325008	272118	1313907

Table F.14: P-DASA scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	9522587	7396541	11189364	15664132	13761103	59421468
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	239318	264483	232281	268989	230513	1074510

Table F.15: C-EDF scheduling statistics on the 8-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	10838592	9471092	12142152	15531177	14098600	50997780
Block	16900948	12336388	20273296	28918404	25039625	106320128
Preschedule	0	0	0	0	0	0
Local	22031424	16575876	26171980	37321281	32208692	142040696
IPI Sent	13387373	9088840	16449244	24556648	20814468	96874396
IPI Received	13384788	9088220	16444600	24545988	20806360	96661864
IPI Missed	2585	620	4644	10660	8108	212532
Migrations	1300857	644292	1870704	3489400	2765529	19003729
Segments	2822145	1953732	3579648	5744266	4809356	28872082
Abortions	0	0	0	0	0	0

Table F.16: G-FIFO scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	7087041	4953292	8937944	14279177	11918988	71493952
Block	0	0	0	0	0	0
Preschedule	15827479	14984491	16647614	19298707	18042268	51533013
Local	21869469	19327546	24152218	31147827	27952199	112058947
IPI Sent	9898902	5479405	12292075	16361166	15132757	33062685
IPI Received	9661687	5333792	11979498	15832575	14697407	30174082
IPI Missed	237215	145613	312577	528591	435350	2888603
Migrations	2008008	1138257	2887307	6004151	4489777	48612725
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	0	0	0	0	0	0

Table F.17: G-NP-EDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	7087041	4953292	8937944	14279177	11918988	71493952
Block	0	0	0	0	0	0
Preschedule	15727848	14889113	16864248	19636935	18314327	53084630
Local	21716614	19176388	24291186	31297034	28103695	111865229
IPI Sent	9671565	5307305	11707599	15034622	14253919	32870896
IPI Received	9466592	5191291	11455256	14577879	13881876	30083522
IPI Missed	204973	116014	252343	456743	372043	2787374
Migrations	1826362	1015336	2800171	5822733	4341038	46352093
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	0	0	0	0	0	0

Table F.18: G-RMS scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29968267	26527321	32895697	40465519	37342369	114504896
Block	101540063	79683992	116001084	157483860	137165641	457641021
Preschedule	0	0	0	0	0	0
Local	74562600	58092227	89272785	137254760	112917433	481738800
IPI Sent	98895031	78092399	112405399	150627785	131857027	410232387
IPI Received	98891701	78091834	112399990	150612115	131847153	409649286
IPI Missed	3330	565	5409	15670	9874	583101
Migrations	2471498	1263325	3686670	8219425	5949801	54758570
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Abortions	0	0	0	0	0	0

Table F.19: G-EDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29450420	25965539	32324553	39798169	36698456	110492896
Block	111391854	88937874	125641793	167493690	146918704	535109234
Preschedule	0	0	0	0	0	0
Local	86036170	68657156	100584849	149124190	124475080	558135490
IPI Sent	108810290	87390813	122121406	161015225	141797749	493135417
IPI Received	108807374	87390248	122116304	161002019	141788695	492793893
IPI Missed	2916	565	5102	13206	9054	341524
Migrations	2440688	1316370	3544625	7420337	5509613	50758760
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Abortions	0	0	0	0	0	0

Table F.20: G-NP-HVDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	7087003	4953282	8937827	14278259	11918678	71411589
Block	0	0	0	0	0	0
Preschedule	15533337	14760685	16355788	19249389	17807031	55460846
Local	21485452	19045722	23666130	30330470	27270009	111582591
IPI Sent	13700921	6581483	15045988	16565403	16471240	32743377
IPI Received	13385470	6423187	14718529	16056197	16020034	30043248
IPI Missed	315451	158296	327459	509206	451206	2700129
Migrations	1681456	944909	2456789	5262714	3889921	44203793
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	24706	18457	41889	138918	68857	961839

Table F.21: G-HVDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29424022	25995336	32297333	39507364	36623557	98478062
Block	132691023	90273730	159005737	248069044	201768291	897005744
Preschedule	96591	65357	131032	418650	212985	10337029
Local	90926685	67022011	114048366	207182658	157178022	874184030
IPI Sent	129450998	88423253	154701536	240377565	195676667	851486331
IPI Received	129442631	88415820	154692043	240363911	195664988	851028620
IPI Missed	8367	7433	9493	13654	11679	457711
Migrations	2491222	1352545	3634935	7901293	5804628	50240334
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	151245	110517	188144	395686	249605	9184092

Table F.22: gMUA scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29125872	25773720	31857542	38868234	36055360	56328364
Block	152417126	111379788	181866067	271116009	226723014	484394113
Preschedule	139843	104025	171440	391345	236924	41908082
Local	113052075	81770337	140259159	232978548	184933562	504727059
IPI Sent	149210456	109433329	177507065	263374155	220457317	467502055
IPI Received	149207768	109432623	177502426	263361025	220448523	467297742
IPI Missed	2688	706	4639	13130	8794	204313
Migrations	3036783	1586202	4464985	11366834	7304138	50055470
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	149018	116187	179367	391979	241866	37915711

Table F.23: NG-GUA scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29187664	25783202	31994650	38865354	36142675	53987354
Block	147440603	102103629	178906444	271938464	226619661	467704679
Preschedule	174913	129629	213330	519199	295867	46657566
Local	112003420	80448002	139760724	235481357	186459399	497110884
IPI Sent	143935258	100103330	174158397	263547577	219793039	453829792
IPI Received	143930980	100100366	174152665	263536395	219784590	453636472
IPI Missed	4278	2964	5732	11182	8449	193320
Migrations	3231142	1739139	4685759	11820785	7596390	48477868
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	189553	150331	224367	513090	302135	40035853

Table F.24: G-GUA scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29069972	25597648	31881166	38649494	36128176	50044906
Block	134996872	93785693	164577377	271787296	217222388	426237684
Preschedule	202597	171469	221088	638397	258623	49720127
Local	95999298	68626358	121922730	235686107	174240726	458776220
IPI Sent	131348793	91624300	159680299	263338995	210244228	414791160
IPI Received	131341863	91620730	159669469	263308239	210224562	414501505
IPI Missed	6930	3570	10830	30756	19666	289655
Migrations	2190749	964053	3751696	20958047	8859383	71970570
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	257102	218446	274169	630325	285657	42370901

Table F.25: P-RMS scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	22986579	19468267	26206034	36059943	31525376	136739580
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	7087041	4953292	8937944	14279177	11918988	71493629
Aborts	0	0	0	0	0	0

Table F.26: P-EDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	24438289	18987701	28748093	39798626	35062966	151258531
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	7087041	4953292	8937944	14279177	11918988	71493629
Aborts	0	0	0	0	0	0

Table F.27: P-HVDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	22542683	18906093	25665314	34851026	30850590	134206875
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	7087041	4953292	8937944	14279177	11918988	71493629
Aborts	175727	309836	174851	407601	234100	1165563

Table F.28: P-LBESA scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	24457970	18866324	28760661	40028220	35122568	151661968
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	7087041	4953292	8937944	14279177	11918988	71493629
Aborts	69814	233231	35686	51113	31280	95565

Table F.29: P-DASA-ND scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	24463214	18897268	28774725	40065782	35153114	151473053
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	7087041	4953292	8937944	14279177	11918988	71493629
Aborts	65896	221350	33649	42741	28468	84988

Table F.30: C-EDF scheduling statistics on the 16-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	29543923	25606257	32911955	42114144	38234412	128044452
Block	64631094	46705784	80578093	125360781	105221755	524866713
Preschedule	0	0	0	0	0	0
Local	58795816	41798741	74836010	121860937	99733257	557761647
IPI Sent	63398463	46014412	78815566	122019537	102607347	499133185
IPI Received	63385169	46012953	78790956	121958891	102568234	497852982
IPI Missed	13294	1459	24610	60646	39113	1280203
Migrations	2312438	984683	3592692	7462860	5673582	48775525
Segments	7087041	4953292	8937944	14279177	11918988	71493952
Aborts	0	0	0	0	0	0

Table F.31: G-FIFO scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	62159119	44061636	77816203	124587697	103829753	619252663
Block	0	0	0	0	0	0
Preschedule	140529744	134121010	146769210	167119870	157880202	437992393
Local	195324083	173496504	215472402	276208970	248688064	963439248
IPI Sent	161317480	88874849	194385344	243620728	232026634	2767101199
IPI Received	152394895	84283073	182289576	223367358	215292454	2692824631
IPI Missed	8922585	4591776	12095768	20253370	16734180	74276568
Migrations	18829900	10951142	26929554	56137054	42140806	455410662
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	0	0	0	0	0	0

Table F.32: G-NP-EDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	62159119	44061636	77816203	124587697	103829753	619252663
Block	0	0	0	0	0	0
Preschedule	138999558	132597124	145248854	168049626	156911650	366915196
Local	192724364	171381432	212926178	275516358	246813370	902383800
IPI Sent	153910936	81069106	187024132	224730218	220651846	6541575696
IPI Received	145324360	76965952	175251380	204965066	204188850	6429381908
IPI Missed	8586576	4103154	11772752	19765152	16462996	112193788
Migrations	17087044	9894048	24453802	52028732	38502620	423068148
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	0	0	0	0	0	0

Table F.33: G-RMS scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	226443024	206592868	243864780	287186816	269269132	586457212
Block	3970394577	3329642057	4125195513	5230927692	4623853604	13055763357
Preschedule	0	0	0	0	0	0
Local	2858578532	2669398932	3289217228	4578503497	3849725796	12909426213
IPI Sent	3917196353	3298049564	4054921804	5102385261	4521598225	12311048060
IPI Received	3917176613	3298046820	4054847332	5101744205	4521322272	12278748157
IPI Missed	19740	2744	74472	641056	275953	32299903
Migrations	26974792	10219364	42901772	102842836	73623041	550188205
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	0	0	0	0	0	0

Table F.34: G-EDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	225859225	205862932	243008944	280851064	266879876	525283612
Block	5044150700	3993803644	5387342669	6982770757	6252014304	15129602187
Preschedule	0	0	0	0	0	0
Local	4097611009	3474995109	4732952677	6524485376	5682367832	15033543943
IPI Sent	5000218708	3966265965	5332487328	6888157259	6174991160	14573821820
IPI Received	5000180388	3966265012	5332441084	6887990133	6174912740	14572213324
IPI Missed	38320	953	46245	167124	78420	1608496
Migrations	23395768	12358940	35122996	78850705	57800716	485028021
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	0	0	0	0	0	0

Table F.35: G-NP-HVDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	62159818	44099232	77869058	124603618	103726294	618379084
Block	0	0	0	0	0	0
Preschedule	137198378	131476578	143187744	165933508	154695540	496475188
Local	190736192	170052502	209268478	268504840	241166666	996905268
IPI Sent	249061332	118140686	256114616	252896138	260699628	3585724460
IPI Received	236710988	112055690	241481346	230907520	242040868	3509445536
IPI Missed	12350344	6084996	14633270	21988618	18658760	76278924
Migrations	15421638	8865140	22370152	48130304	35614780	412955856
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	344830	250038	454026	1276832	701832	21187812

Table F.36: G-HVDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	187577742	172454490	198232226	203798100	208348794	202211052
Block	3994103584	2952611764	4561455286	5868197364	5410385584	5561582160
Preschedule	12023092	9522232	14481612	39159090	22440960	423559452
Local	2973395564	2350372854	3520281606	5262037446	4566793224	5329382592
IPI Sent	3921855860	2911108806	4467303376	5728917898	5287610960	5154857912
IPI Received	3921814882	2911095688	4467220958	5728670654	5287426830	5151777956
IPI Missed	40978	13118	82418	247244	184130	3079956
Migrations	30955356	13490688	45806130	78088904	68748490	201626940
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	13161248	5756807	14991234	18941113	11083435	397652240

Table F.37: gMUA scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	175691328	166506392	178910704	156555204	173688340	99820904
Block	3761689556	2902599552	4300430552	4509462384	4699319252	2143833960
Preschedule	16976928	11121792	23332028	66153736	40022468	607101648
Local	2981210424	2362530628	3553059896	4133734696	4124585188	2416130472
IPI Sent	3687465432	2855357960	4201875800	4375853440	4572174028	1969042052
IPI Received	3687404368	2855334828	4201744260	4375180552	4571855096	1965707088
IPI Missed	61064	23132	131540	672888	318932	3334964
Migrations	52287120	20170144	84368280	176276780	135061176	214923600
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	17979552	14640644	23737756	65511432	40064992	556212512

Table F.38: NG-GUA scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	168150112	159717844	170282036	148495784	163646920	100294108
Block	3585219510	2728475930	4060119294	4270631820	4364024678	2261183200
Preschedule	16744224	10922894	22982456	66130044	40289856	601994948
Local	2831970608	2198077706	3339757450	3910393674	3807759440	2550835240
IPI Sent	3506667314	2677084284	3959839836	4133515102	4233868630	2081200736
IPI Received	3506606628	2677061954	3959728738	4133080654	4233631820	2077976124
IPI Missed	60686	22330	111098	434448	236810	3224612
Migrations	49489522	19293068	78671956	169017754	126349436	228016100
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	17917780	14366476	23425924	65620424	40350502	551889400

Table F.39: G-GUA scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	130789710	131371918	122119910	104428640	113011918	89322404
Block	1981689624	1668928240	2120139240	2494621244	2377515542	1983903664
Preschedule	17926864	7464612	34279068	87116302	61498648	616148432
Local	1366002164	1164066786	1561758864	2178146546	1941392194	2221953568
IPI Sent	1873612972	1587060370	1995201320	2347036068	2233646896	1813285348
IPI Received	1873565472	1587050540	1995098166	2346537010	2233380810	1809434532
IPI Missed	47500	9830	103154	499058	266086	3850816
Migrations	30224630	4519802	67033620	198755388	143162250	320279872
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	25655088	18096910	38368766	85596330	61443918	564657144

Table F.40: P-RMS scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	203727013	174308697	230773668	315535033	276411617	1196452737
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	0	0	0	0	0	0

Table F.41: P-EDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	216363100	169420137	252954168	347188360	306674793	1299400592
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	0	0	0	0	0	0

Table F.42: P-HVDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	199856857	169325896	225812384	304331257	269302165	1166730976
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	1787012	701439	2056473	4558221	2920377	18040856

Table F.43: P-LBESA scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	217106529	168542701	253792888	350335200	308103660	1321765373
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Aborts	613924	2086804	360077	515013	326576	3671847

Table F.44: P-DASA-ND scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	0	0	0	0	0	0
Block	0	0	0	0	0	0
Preschedule	0	0	0	0	0	0
Local	217354588	168888581	253953852	350858864	308456403	1289616160
IPI Sent	0	0	0	0	0	0
IPI Received	0	0	0	0	0	0
IPI Missed	0	0	0	0	0	0
Migrations	0	0	0	0	0	0
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Abortions	588649	1976765	349905	425804	293553	2245727

Table F.45: C-EDF scheduling statistics on the 48-core platform

	BHB	BHU	BMB	BMU	BLB	BLU
Global	245562084	214034101	271681967	346486004	314285381	1036775180
Block	672611621	519918049	795029949	1145255301	986394988	4450421024
Preschedule	0	0	0	0	0	0
Local	615001761	454965525	743018084	1125732321	943823244	4906453469
IPI Sent	659394325	512904161	776448327	1110815361	958874125	4299407082
IPI Received	659037683	512870241	775610786	1108058140	956960924	4229956842
IPI Missed	356642	33920	837541	2757221	1913201	69450240
Migrations	20423060	8321503	31220461	64928044	49321269	436000343
Segments	62159119	44061636	77816203	124587697	103829753	619252663
Abortions	0	0	0	0	0	0