Implementation and Evaluation of a Fixed Priority Scheduler based on Priority Promotions for the Linux Kernel

Master's thesis in Computer Systems and Networks

CHRISTOFFER ACKELMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Implementation and Evaluation of a Fixed Priority Scheduler based on Priority Promotions for the Linux Kernel

CHRISTOFFER ACKELMAN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Implementation and Evaluation of a Fixed Priority Scheduler
based on Priority Promotions for the Linux Kernel
CHRISTOFFER ACKELMAN

Supervisor: Risat Pathan, Department of Computer Science and Engineering
Advisor: Marcus Nordenberg, SSAB EMEA AB
Examiner: Jan Jonsson, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Implementation and Evaluation of a Fixed Priority Scheduler
based on Priority Promotions for the Linux Kernel
CHRISTOFFER ACKELMAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Priority promotions is a mechanism to make the priorities in a fixed priority schedulers more dynamic. The purpose of this thesis was to investigate whether the Fixed Priority with Priority Promotions (FPP) scheduler, which is uses a number of priority promotions to generate an Earliest Deadline First (EDF) schedule, is an effective strategy for reducing the overhead of real-time EDF schedulers when implemented on a real platform. To investigate, the FPP scheduler, a “standard” EDF scheduler based on a binary heap, and an EDF scheduler based on a Red-Black Tree were implemented. The FPP scheduler was then benchmarked against the standard EDF scheduler, but also against the rather mature and optimized EDF scheduler in the Linux kernel, SCHED_DEADLINE, which is based on a Red-Black Tree. The hypothesis was that the FPP scheduler would perform better than the binary-heap based EDF scheduler, but that it would be inferior to SCHED_DEADLINE since this scheduler has been developed and improved over several years by many very competent developers. The first results showed that the implemented FPP and EDF schedulers were largely incomparable to SCHED_DEADLINE with regard to the release and scheduling overheads. Thus, the second EDF scheduler using the Red-Black Tree from SCHED_DEADLINE was developed. The results showed that the FPP scheduler outperform both the heap based and Red-Black Tree based EDF schedulers implemented in this thesis. It was concluded that FPP is an effective strategy for reducing the overhead of EDF schedulers, even for very optimized implementations of EDF, when implemented on real platforms.

Keywords: Computer Science, Earliest-Deadline-First, Multiprocessor, Preemptive, Priority Promotion, Real-Time Scheduling.

Acknowledgements

First, I would like to thank my supervisor Risat Pathan for suggesting this thesis in the first place, and for his tremendous help with the report and presentation. I am also grateful to my examiner Jan Jonsson for his support and his interesting course in real-time systems. If it were not for the course by Jan, I would never have read the excellent follow-up course by Risat, which ultimately led to this thesis.

Second, I would like to thank the PSS team at SSAB, and my supervisor Marcus Nordenberg in particular, for their help during the thesis, for providing me with hardware, and for allowing me to do this thesis at their company in the first place.

Finally, I would like to thank my opponent Philip Ståhlhammar for reviewing the report, pointing out inconsistencies and asking valuable questions that helped raise the quality of the report.

Christoffer Ackelman, Gothenburg, June 2018

Contents

List of Acronyms

CBS Constant Bandwidth Server.

CFS Completely Fair Scheduler.

CPU Central Processing Unit.

DM Deadline Monotonic.

EDF Earliest Deadline First.

EDF-RBT EDF with Red-Black Tree.

EDF-BH EDF with Binary Heap.

FIFO First In, First Out.

FMLP Flexible Multiprocessor Locking Protocol.

FPP Fixed Priority with Priority Promotions.

GSN-EDF Global EDF with synchronization support.

HRT Hard Real-Time.

ISR Interrupt Service Routine.

L1 level-1.

L2 level-2.

L3 level-3.

liblitmus	The LITMUSˆRT User-Space Library.
LITMUSˆRT	Linux Testbed for Multiprocessor Scheduling in Real-Time Systems.
RM	Rate Monotonic.
RR	Round Robin.
SchedCAT	The Schedulability Test Collection And Toolkit.
SRT	Soft Real-Time.
SSAB	Svenskt Stål AB.
WCET	Worst-Case Execution Time.

List of Figures

1 Introduction

A real-time system is a system where the correctness of tasks depend not only on the result, but also on the time when the result is delivered. Usually, every task has a deadline and must deliver the result before that deadline. A common example of a real-time system is the braking system in a car, if you press the brake and the car brakes too late, after you have already crashed, then the result is useless. In real-time systems the tasks usually arrive (i.e. the task wants to execute) periodically with a fixed period between arrivals. Every new arrival of a task is called a *job*. A scheduling algorithm is responsible for deciding which job to execute next.

A simple scheduling algorithm is Rate-Monotonic (RM) which always schedules the task with shortest inter-arrival time. RM is a *fixed* priority scheduler since the priorities are assigned beforehand and never change during runtime. On uniprocessors, the slightly more complex Earliest Deadline First (EDF) is optimal. EDF works as follows: at every time instant, it schedules the job whose deadline is closest in time. EDF is a *dynamic* priority scheduler, since the priority of a task depend on the current time instant and thus change during runtime. Since version 3.14 the Linux kernel implements an EDF scheduler called SCHED_DEADLINE [1].

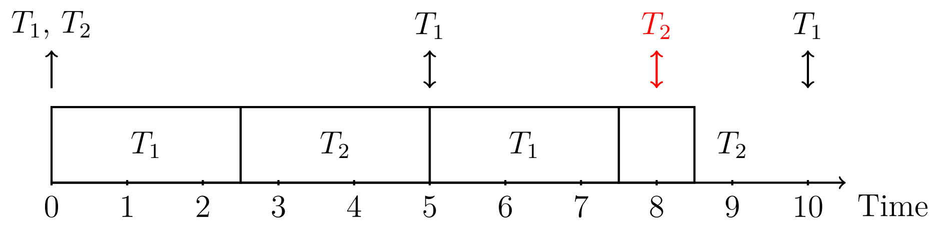
1.1 Problem Formulation

In real-time systems, schedulability analysis is used to determine if a task set is guaranteed to meet all its deadlines when scheduled by a particular scheduler. A problem with the research on real-time systems is that almost all such analysis are done in a platform-agnostic manner, where practical quirks, such as runtime overhead, are abstracted away and overlooked.

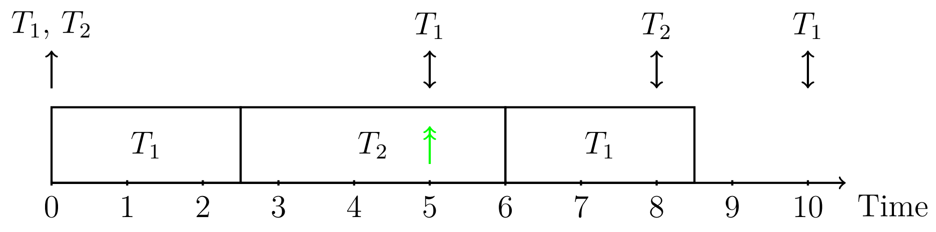
The problem with dynamic priority schedulers, such as EDF, is that they need to manage and sort the jobs in the ready queue, which causes considerable runtime overhead compared to fixed-priority schedulers, such as RM. This problem was solved in theory by Pathan in 2015 [15] by combining a fixed-priority scheduler with priority promotions to exhibit the behaviour of a dynamic priority scheduler.

A priority promotion point is a point in time where the priority of a task is promoted/increased, see figure 1.1. If assigned carefully, such a promotion point can promote the priority of a task when it is close to its deadline, such that it receives higher priority than one of the currently executing tasks and thus preempts that task. This master thesis aims at comparing the scheduling algorithm presented by Pathan [15] to the SCHED_DEADLINE scheduler, attempt to quantify the runtime overhead, and identify how it affects the schedulability of task sets when scheduled by these scheduling algorithms.

Task	Period	WCET
T_1	5	2.5
T_2	8	3.5



(a) Rate-monotonic scheduling



(b) Rate-monotonic with priority promotion

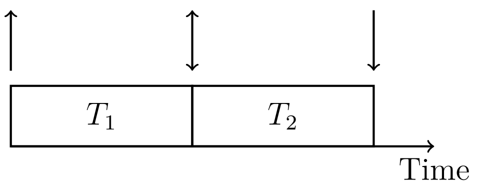
Figure 1.1: The execution of a task set with only two tasks, where task T_1 has a shorter period and thus higher priority under RM. Each rectangle represents the execution of a single job of the task and the down-arrows denote the deadline of a job. The green arrow denote a priority promotion of task T_2 such that it can no longer be preempted by T_1, and thus finishes before its deadline.

In [15, 16, 17] Pathan describe the details of the Fixed Priority with Priority Promotions (FPP) scheduler, and thus form the basis of the implementation that shall be evaluated in this master thesis. These works [15, 16, 17] also describe the benchmarks used by Pathan to empirically evaluate the scheduler on an emulated platform, similar benchmarks shall be used when evaluating the scheduler on the Linux kernel, see section 6.3. Pathan [15, 16, 17] indicates that FPP may cause fewer preemptions than EDF, but none of the papers [15, 16, 17] has investigated how a time penalty, such as the runtime overhead caused by these preemptions, would affect the performance of the two scheduling algorithms. This does however suggests a possible advantage that the FPP scheduler may have over EDF.

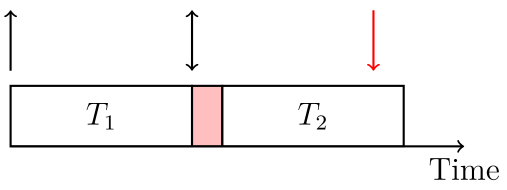
The implications of runtime overhead has been investigated to some extent by Brandenburg and Anderson [10] when comparing different optimization strategies for EDF schedulers, see section 4.2. Brandenburg et al. has also made a more detailed analysis of the cache-related overheads [5]. However, so far no such analysis has been done on the FPP scheduler presented by Pathan. As indicated by the paper “Design of an efficient ready queue for earliest-deadline-first (EDF) scheduler” [16], FPP can be considered as an optimization strategy for EDF, thus making it an ideal candidate for a comparison similar to the one done by Brandenburg and Anderson in [10]. The following example will show why it is important to consider the runtime overhead when analyzing a real-time system.

1.1.1 Problem Example

Consider a task set with two tasks, where the utilization of the task set is nearly 100%, for example 98%. In theory this task set should be schedulable on a uniprocessor, which is illustrated in figure 1.2 (a). However, on real systems the scheduler is invoked to make a scheduling decision before a new job can start executing. This scheduling decision causes some runtime overhead, which is illustrated by the red area in figure 1.2 (b). If the total runtime overhead is greater than 2% of the available processing time, the second task will miss its deadline even though it theoretically should be schedulable. To avoid deadline misses, the total runtime overhead that a task suffers from must be factored into the Worst-Case Execution Time (WCET) of the task before applying the schedulability analysis to determine if the task set is schedulable or not. Since the runtime overhead does not contribute anything to the functionality of the task, it is desirable to minimize it. As described in section 1, dynamic priority schedulers typically suffer from considerable runtime overhead, and thus many system designers choose to implement a fixed priority scheduler instead, even if the fixed priority scheduler may not be able to utilize the processor to the same extent as dynamic priority schedulers such as EDF can.



(a) The theoretical outcome without runtime overheads.



(b) The real world outcome with runtime overheads.

Figure 1.2: The execution outcome of a task set with only two tasks, where the tasks WCETs are equal to their periods. Up-arrows denote the arrival of a job, and down-arrows denote the deadline of a job. Each rectangle represents the execution of a single job of the task. The filled red rectangle denote the runtime overhead caused by the scheduler, and the red down-arrow denote the (missed) deadline of task T_2.

As discussed in section 1.1, Pathan [15, 16, 17] implies that the FPP scheduler is able to generate the same schedule as EDF, while causing fewer preemptions and migrations than a typical EDF scheduler. Fewer preemptions and migrations does imply the possibility of less overall runtime overhead, but this cannot be assumed without a proper implementation. If each preemption causes more runtime overhead, then it counteracts the benefit of fewer preemptions, and the result may actually be more overall runtime overhead. As was already stated, the resulting runtime overhead when running on a real system has not been analyzed previously and this is the aim of the master thesis.

1.2 Goals

The theoretical challenge of this master thesis is to quantify the runtime overhead and identify how it affects the performance of the two scheduling algorithms: FPP and standard EDF. Pathan [15, 16, 17] implies that FPP may reduce the number of preemptions and migrations, and therefore cause fewer context switches than EDF. Intuitively, FPP should then be able to perform even better on the Linux kernel, where context switches are expensive compared to the emulated platform used by Pathan. However, this has not been explored yet and remains to be seem.

The practical challenge of this master thesis is to implement the fixed-priority (FPP) scheduler presented by Pathan, and make the interface so similar to the dynamic priority SCHED_DEADLINE, that it can be considered as an updated version of SCHED_DEADLINE that does not break any backwards-compatibility. Switching between the schedulers should be as simple as using a single system call, such that system designers can switch from SCHED_DEADLINE to the FPP scheduler, and vice versa, with minimum effort. Preferably, every task set schedulable by SCHED_DEADLINE should also be schedulable by the FPP scheduler.

Besides a fully functional implementation of the FPP scheduler, the goal of the master thesis is to quantify the runtime overheads listed below and empirically acquire an upper bound for each source of runtime overhead. These overheads are further described in section 2.5.

- *Scheduling overhead* which is caused directly by the runtime scheduler when selecting the next job to run.
- *Release overhead* which occurs when a new job is released and the scheduler must either enqueue the job in the ready queue, or it may also decide to schedule the job. Many papers do not distinguish between release and scheduling overhead and simply consider the release overhead to be part of the scheduling overhead. For our purposes, it simplifies further discussions if we distinguish between them.

- *Context-switch overhead* which is caused by switching the stack and processor registers. The scheduler indirectly cause context-switch overhead when changing between tasks.

An additional goal of the thesis is that the implementation of the schedulers should be fairly straightforward and simple, such that other developers can quickly become acquainted with the source code and make use of it. The code should not contain any premature optimizations that make it difficult to understand.

1.3 Limitations and Scope

The master thesis should implement and analyze an FPP scheduler, and two EDF schedulers, one EDF scheduler with a binary heap and one with a red-black tree. The implemented schedulers and SCHED_DEADLINE shall be evaluated with respect to three different runtime overheads: release overhead, scheduling overhead and context-switch overhead. These runtime overheads shall be quantified for each of the schedulers by empirically obtaining upper bounds on two different Central Processing Unit (CPU)s, an Intel i3-4370 x86 processor and a quad-core ARM Cortex-A53. However, this master thesis will not provide an analytical model for these sources of runtime overhead, as this is a difficult research question on its own. The general performance of the schedulers shall also be evaluated by analyzing the number of randomly generated task sets that are guaranteed to successfully meet their deadlines when scheduled by each of these schedulers. Finally, there shall be a discussion about these runtime overheads, the general performance of the schedulers as well as how the implementation details of the schedulers affect their performance with respect to the runtime overheads.

Other tasks are out of scope for the master thesis and will not be carried out. In particular, SCHED_DEADLINE has been optimized for several years by many different people. This thesis does not have the time to optimize the implemented schedulers, when they are implemented and working as intended they must be considered “done”. Otherwise it is easy to get stuck optimizing and refining the code forever, a path that has led countless software projects into the grave.

1.4 Report Structure

The remainder of this report follows a rather traditional structure with Background, Method, Results and Conclusion. Chapter 2 provides a background on the field of real-time scheduling, presenting the theory and notation necessary to understand the contents of the report. Chapter 4 continues by presenting recent advances in the field, to get the reader up to date on the related works that this thesis builds upon. After that, chapter 5 describe the implementation of the scheduling algorithms evaluated in this thesis, and chapter 6 describe the tests and benchmarks used to evaluate the implemented schedulers. The results of these tests and benchmarks are presented in chapter 7. Finally, a discussion regarding these results are held in chapter 8 before the report ends with a few concluding remarks in chapter 9.

2

Theoretical Background

This chapter presents the background required to follow the rest of the thesis. It starts off with hardware fundamentals, then proceeds to explain the theoretical model used in real-time research, followed by a short background about common real-time schedulers, before concluding with schedulability analysis and scheduler overheads.

2.1 Hardware Fundamentals

We begin by giving a brief refresher on hardware fundamentals. Given the extent of the topic, a comprehensive background is beyond the scope of the thesis. Thus, we will present only the parts that have the biggest impact on real-time scheduling, namely multicore processors, caches and interrupts. If you as a reader want a more thorough background about computer hardware, there are free online courses in computer architecture offered by Harvard or MIT.

2.1.1 Multiprocessors

A *multiprocessor* is an umbrella term used to denote two or more processing units connected by a communication bus. The most common type of multiprocessor is a multicore processor where several processing cores are placed on a single CPU. From the perspective of real-time scheduling, there is no difference between a processor core and a single-core processor, and thus we make no difference between a four-core processor and four separate single-core processors. The term multiprocessor can also be used to denote several CPUs connected within a single system (for example a motherboard with several CPU sockets), or even separate systems connected by a communication bus (i.e. a distributed system). The remainder of this thesis will focus on homogeneous multiprocessors, more commonly called symmetric multiprocessors (SMP) because all processors are identical to each other. Systems where the different processors are not identical to each other, such as the ARM big.LITTLE processors where powerful and power-hungry (big) processor cores are coupled with slower and battery-saving (LITTLE) ones [4], are out of scope of this thesis and will not be analyzed.

2.1.2 Caches

To reduce the high latency of accessing the main memory, modern processors utilize a hierarchy of smaller and faster memories called *caches*. The caches store a copy of the most frequently used instructions and data fetched from the main memory such that this data can be read and manipulated quickly without accessing the main memory. Caches are organized in several layers where the smallest and fastest caches closest to the execution units are denoted level-1 (L1) caches, see figure 2.1. Most modern processors have two L1 caches per CPU core, one instruction cache (I-Cache) and one data cache (D-Cache). Some processors also have a larger and slightly slower level-2 (L2) cache per CPU core farther away from the execution units. The L2 caches store both instructions and data. Finally, most processor have a single level-3 (L3) cache shared by all cores. The L3 cache is larger and slower than the L2 caches, but it is still very small and fast compared to the main memory. To give you an idea of the relative speedup of utilizing caches, a L1 cache hit may take only around 1.5% of the time required to access the main memory. The cache latency values specific to the hardware used in this thesis is listed in section 2.1.2.1 and 2.1.2.2. For a more comprehensive summary about the memory hierarchies of modern processors, the paper “What Every Programmer Should Know About Memory” by Ulrich Drepper [12] is an excellent resource.

2.1.2.1 x86 processor

The x86 processor used in this thesis is a two core Intel Skylake i3-4370 with HyperThreading, which is an Intel-specific technique that allows one physical core to appear as two logical cores to the operating system, thus the i3-4370 has four logical cores. By duplicating the architectural state (i.e. the registers) of each core, one logical core can continue executing instructions while the other logical core has stalled (which is very common). The x86 processor has three levels of cache, see figure 2.1: L1 consists of a 32K instruction cache and a 32K data cache, L2 consists of a 256K shared cache and L3 consists of an 8MB cache shared across all cores. Given that the clock frequency is 3.4 Ghz, a L1 cache hit takes 1 ns, a L2 cache hit 3.5 ns, a L3 cache hit 13 ns, and main memory access 63 ns (excluding the time it takes to transfer the actual data, which is another 32 ns to transfer a 128 Byte block of data)[11].

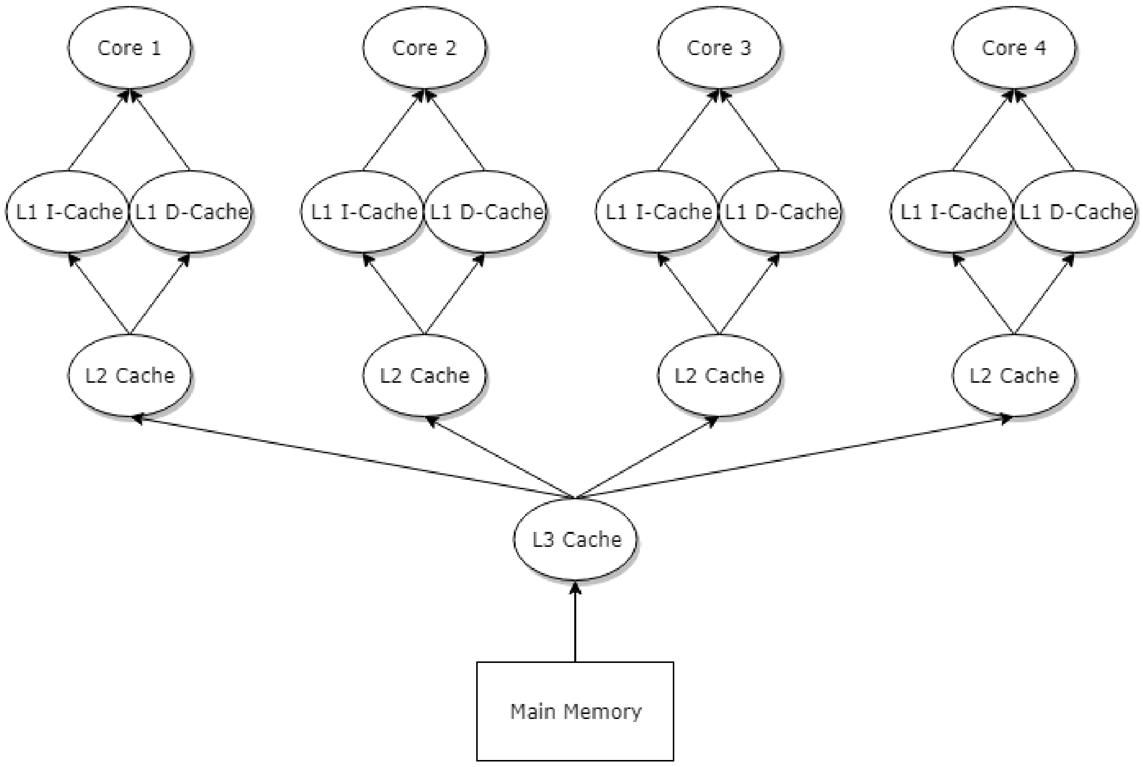


Figure 2.1: The 3-level cache hierarchy used in x86 processors with two level-1 (L1) caches per core, one instruction cache (I-Cache) and one data cache (D-Cache), one level-2 (L2) cache per core and a single level-3 (L3) cache shared by all cores.

2.1.2.2 ARM processor

The ARM board used in this thesis is a Raspberry Pi 3 model B, which has an ARM Cortex-A53 processor. It has two levels of cache, see figure 2.2: Level 1 consists of a 32K instruction cache and a 32K data cache, and Level 2 consists of a 512K cache shared across all cores. The “Level 2” cache of the Raspberry Pi corresponds to the L3 cache on x86 processors, thus the cache hierarchy of the Raspberry Pi is essentially the same as the x86 processor where the L2 caches of the x86 have been removed. Given that the clock frequency is 1.2 Ghz, a level 1 cache hit takes 2.5 ns, a level 2 cache hit 12.5 ns, and main memory access 140 ns [3].

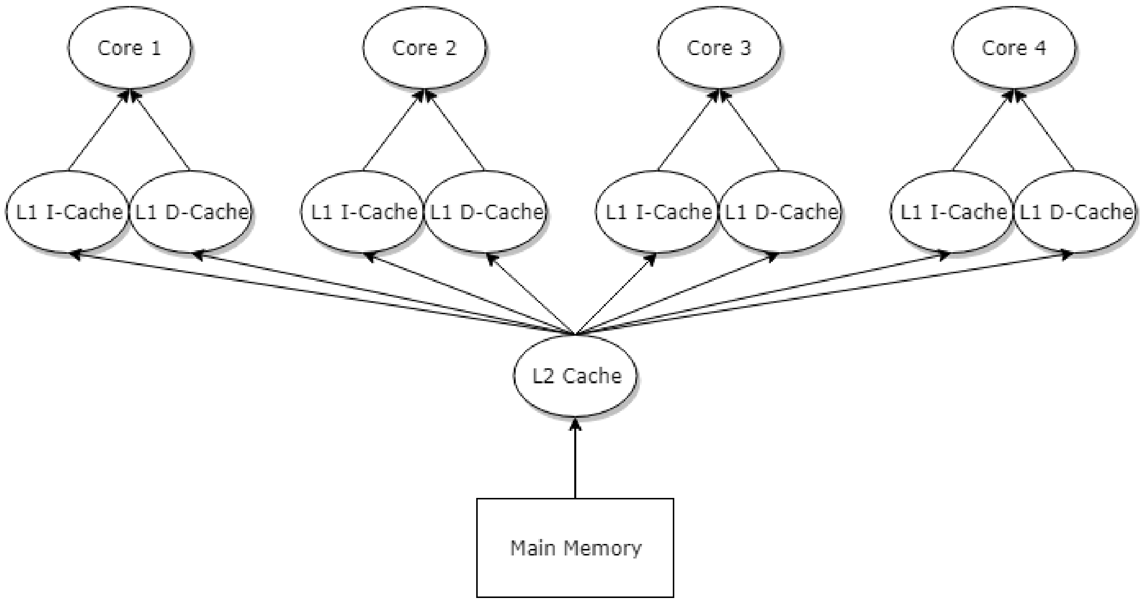


Figure 2.2: The 2-level cache hierarchy used in the Raspberry Pi with two level-1 (L1) caches per core, one instruction cache (I-Cache) and one data cache (D-Cache), and a single level-2 (L2) cache shared by all cores.

2.1.3 Interrupts

Instead of busy waiting for asynchronous events, such as accessing the main memory, modern processors use interrupts. With interrupts, the processor continues executing other instructions while waiting and when the asynchronous event has finished, an interrupt flag is set to notify the processor. When the processor detects an interrupt, it pauses the execution of the current process, saves a copy of all register values to the stack, and executes a special Interrupt Service Routine (ISR) instead. When the ISR has finished, the processor pops the registers from the stack and continues executing the previous task. Interrupts are important to us because the ISR causes delays that must be taken into account when determining if a task set is schedulable or not.

There are several types of interrupts, and the three main categories are Device interrupts, Timer interrupts and Inter-Processor interrupts. Device interrupts are generated by hardware devices, for example the main memory to indicate that data has been read and is now available to the processor. Timer interrupts are generated by the expiry of hardware timers that have been armed at an earlier point in time. These hardware timers can be used to trigger job releases or enforce execution-time budgets for example. Finally, Inter-Processor Interrupts are used in multiprocessor systems to synchronize caches or cause another processor to reschedule.

Most interrupts can be disabled, such that the processor delays the invocation of the ISR until interrupts are enabled again. There are however interrupts that cannot be disabled, such as the “watch dog” functionality that detects if the system has hanged. In the multicore processors used in this thesis, each processor core has its own set of interrupt flags, thus disabling interrupts on one processor core does not disable them on the other cores.

Some multicore processors support dedicated interrupt handling, where one of the cores receives and handles all device interrupts and most of the timer interrupts. Inter-Processor Interrupts can not be handled by a single core for obvious reasons. Recent versions of Linux does support dedicated interrupt handling, but the configuration is complicated and thus it was not used in this thesis.

2.2 Real-Time Task Model

In real-time systems, the tasks (also called processes outside of the real-time research area) are often recurrent and have a fixed period. An instance of a task – a job of that task – starts, executes for a short period, and then suspends itself. After a *period*, denoted p_i, of time, the task arrives again, meaning that the scheduler should invoke a new job of that task. The maximum time from the start of the task until it has finished executing and suspends itself is called the WCET, denoted c_i, of the task. This is an upper bound on the time it may take to execute the task. In real-time systems, the tasks may also have a *deadline*, denoted d_i, before which they must complete their execution. See figure 2.3 for a visual description of these terms. The *task set* τ is the set of all tasks, each denoted T_i, where i is the index of the task. This way of defining tasks by their worst-case execution time, period and deadline, $T_i = (c_i, p_i, d_i)$, is called the Liu and Layland task model or more simply the periodic task model.

If a task does not have a *constrained deadline* $d_i \leq p_i$, the task is given an *implicit deadline* equal to the period of the task, $d_i = p_i$. This is done to prevent previous jobs of a task from interfering with new jobs of the same task. The jobs of a *periodic* task always have a fixed time interval p_i between job arrivals. If the jobs of a task arrive with a minimum time between interval p_i, but may arrive less frequently, then the task is *sporadic*. Sporadic tasks may be used to model external events, for example incoming network packets. It may be the case that the system should receive sensor data over the network every 50 ms, but if there is contention over the network, the data may be delayed or even dropped.

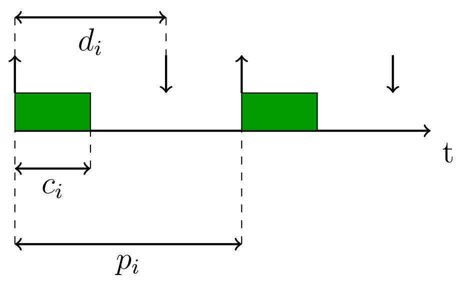


Figure 2.3: A visual representation of the Liu and Leyland task model. Up-arrow denotes arrival of the task, and down-arrow denotes the deadline of the task. Each filled rectangle represents the execution of a single job of the task. c_i denotes the worst-case execution time, p_i denotes the period, and d_i denotes the deadline of the task.

Based on the Liu and Layland task model, it is common to derive parameters that are specific to each job of the task. The *arrival time* (denoted $a_{i,j}$) of a job is the time when the job arrives to the scheduler and can start executing. For periodic tasks, the arrival time can easily be calculated as $a_{i,j} = (j-1)p_i$. The *start time* (denoted $s_{i,j}$) is the time when the job is assigned to a processor by the scheduler. The difference between the arrival time and the start time of a job is that the arrival time is the time when the job is ready to start executing, while the start time is the time when the job is scheduled on a processor and actually starts executing. When a job arrives to the scheduler, it may be scheduled immediately, and thus the start time will simply be the arrival time plus the runtime overhead of the scheduler. However, the job may also be stored in the ready queue of the scheduler to be scheduled at a later point in time. The *finishing time* (denoted $f_{i,j}$) of a job is the time when it has finished execution and suspends itself. Finally, the *response time* (denoted $r_{i,j}$) is the difference between the completion time and the arrival time of the job. See figure 2.4 for a visual description of these terms.

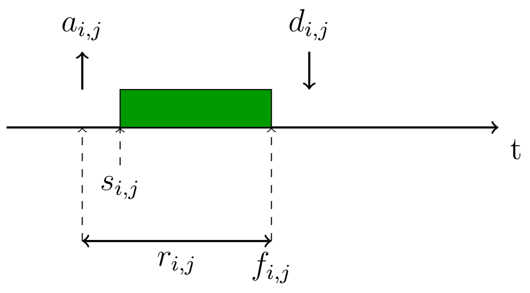


Figure 2.4: A visual representation of the job parameters derived from the Liu and Leyland task model. Up-arrow denotes arrival of the task, and down-arrow denotes the deadline of the task. The filled rectangle represents the execution of a single job of the task. $a_{i,j}$ denotes the arrival time, $s_{i,j}$ denotes the start time, $d_{i,j}$ denotes the absolute deadline, $f_{i,j}$ denotes the completion time and $r_{i,j}$ denotes the response time of the job.

2.3 Real-Time Scheduling

A *scheduler* is an implementation of a *scheduling algorithm* and is the program responsible for selecting which job to execute at any given time instant.

2.3.1 Uniprocessor Real-Time Scheduling

In *fixed* priority schedulers, the task priorities are assigned beforehand and never change during runtime. An example of a fixed priority scheduling algorithm is Rate Monotonic (RM) which always schedule the task with the shortest period. The tasks are sorted by their period, and priorities are assigned depending on the resulting order beforehand. Another fixed priority scheduling algorithm is Deadline Monotonic (DM) which is very similar to RM, but instead of scheduling the task with the shortest period, it always schedule the task with the shortest relative deadline. As with RM, the tasks are sorted and priorities are assigned beforehand, but under DM the sorting is done based on the relative deadline of each task.

In *dynamic* priority schedulers, the priority of a task change during runtime. An example of a dynamic priority scheduling algorithm is EDF, which is an optimal scheduling algorithm on uniprocessors. EDF works as follows: at every time instant, it schedules the job whose absolute deadline is closest in time. Thus, the priority of a task depend on the current time instant.

2.3.2 Multiprocessor Real-Time Scheduling

Multiprocessor scheduling algorithms can be implemented in two primary ways. In *partitioned* scheduling each processor is considered as a separate uniprocessor machine, where each processor has its own ready queue and the tasks are divided among the processors beforehand. Schedulability analysis of partitioned scheduling algorithms is rather straightforward, except for the problem of dividing the task set among the processors, the analysis is equivalent to the uniprocessor case.

The other way of implementing multiprocessor scheduling is *global* scheduling, where the processors are considered as a common pool of resources that every task may use and where all processors share a single ready queue. Global scheduling has the potential to better utilize the processors. In contrast to partitioned scheduling, where the schedulability analysis is rather straightforward, global scheduling requires new techniques for schedulability analysis. The rest of this thesis will focus on global scheduling.

2.4 Schedulability Analysis

As real-time systems are often safety-critical, it is important that their correctness can be verified, not only in their logic, but also in their timing characteristics. Schedulability analysis is a way of formally proving that a certain task set will meet all deadline even in the worst-case scenario. Typically, schedulability tests use the parameters of the tasks combined with the knowledge of the scheduler used to answer the question of whether all jobs can be guaranteed to finish before their deadline.

In Hard Real-Time (HRT) systems, deadlines are firm and every job must finish execution before its (absolute) deadline. From this follows that the maximum response time for every task $T_i \in \tau$ must be less than or equal to its relative deadline d_i. Thus, we can define schedulability for hard real-time systems as follows:
Definition 2.4.1. A task set τ is HRT *schedulable* under a scheduling algorithm A if and only if $r_i \leq d_i$ for all tasks T_i when scheduled by A.

In some cases, deadline misses are acceptable if the extend of the deadline misses are bounded by a constant. Such systems are called Soft Real-Time (SRT), and we can define schedulability for soft real-time systems as:
Definition 2.4.2. A task set τ is SRT *schedulable* under a scheduling algorithm A if and only if there exists a constant C such that $r_i \leq d_i + C$ for all tasks T_i when scheduled by A.

A real-time system may also have a mix of both hard real-time tasks and soft real-time tasks.

A schedulability test is *sufficient* if a positive answers from the schedulability test guarantees that the task set is schedulable. A negative answer from a sufficient test does not say anything about the task set, and it may be schedulable anyway.

2.5 Overheads

As said in section 1.1, a problem with most research on real-time systems is that runtime overheads are abstracted away. On real platforms, there are several sources of runtime overhead [10] and those most relevant to this master thesis are listed below:

- *Scheduling overhead* which is caused directly by the runtime scheduler when selecting the next job to run.
- *Release overhead* which occurs when a new job is released and the scheduler must either enqueue the job in the ready queue, or it may also decide to schedule the job. Many papers do not distinguish between release and scheduling

overhead and simply consider the release overhead to be part of the scheduling overhead. For our purposes, it simplifies further discussions if we distinguish between them.

- *Context-switch overhead* which is caused by switching the stack and processor registers. The scheduler indirectly cause context-switch overhead when changing between tasks.

An example showing these sources of runtime overhead can be seen in figure 2.5

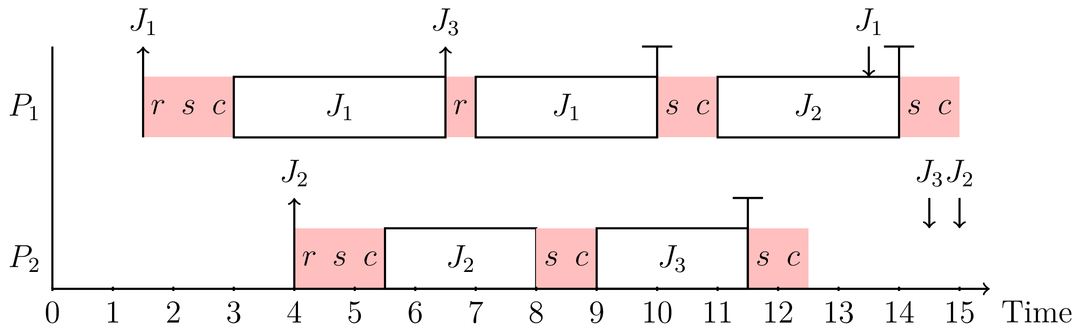


Figure 2.5: A more complicated execution example taken from [10] with three jobs (J_1, J_2 and J_3) on two processors (P_1 and P_2), showing the different overheads caused by the scheduler. Each rectangle represents the execution of a single job of the task, up-arrows denote the arrival of a job, down-arrows denote the deadlines and small "T" represent job completions. The filled pink rectangles denote the runtime overheads caused by the scheduler, where "r" is the release overhead, "s" is the scheduling overhead and "c" is the context switch overhead. Note that J_3 is released on processor P_1, which asks processor P_2 to reschedule since J_3 has higher EDF priority than J_2, but lower than J_1.

The impact of runtime overheads must be considered when doing the schedulability analysis, otherwise the task set cannot be guaranteed to meet all its deadlines. The typical way of accounting for the runtime overheads is to inflate the worst-case execution time of each task by the maximum amount of overhead it may experience from the scheduler. Thus, task set designers must know the worst-case overhead of the scheduler used, which this thesis aims to provide by quantifying the overheads of the implemented schedulers and SCHED_DEADLINE.

3

Practical Background

This chapter presents the practical work that this thesis builds upon.

3.1 Real-Time Scheduling in Linux

Linux uses a hierarchy of schedulers, see table 3.1. Each task has a parameter “policy” that indicates which scheduler it wants to run under. Linux iterates over all available scheduling classes, and lets each scheduling class decide if there are any jobs that it wants to schedule. A lower-priority scheduling class is only invoked if the higher-priority scheduling classes do not have any jobs to schedule. After a scheduling class has picked a job to schedule, the iteration stops, see listing 1. For example, SCHED_DEADLINE only schedule jobs of tasks with policy “SCHED_ DEADLINE”. If at least one job from a task with policy SCHED_DEADLINE wants to execute, SCHED_DEADLINE will return that job and the loop will exit. If no job with policy SCHED_DEADLINE is ready for execution, the SCHED_DEADLINE scheduler returns NULL (i.e. false) and on the next iteration of the loop, the next scheduling class, SCHED_RT, will decide if it has any jobs to schedule. Thus, if no jobs from real-time tasks want to execute at the moment, the Completely Fair Scheduler (CFS) will be invoked to schedule non real-time tasks.

1	STOP
2	DEADLINE
3	RT
4	CFS
5	IDLE

Table 3.1: The hierarchical scheduling classes in Linux.

```
for_each_class(class) {
  p = class->pick_next_task(rq, prev);
  if (p) {
    return p;
  }
}
```

Listing 1: Source code from the Linux kernel that iterates over the available scheduling classes and lets each scheduling class decide if it has any tasks to schedule.

The STOP scheduling class is a special scheduler that schedules a per-CPU stop task that preempts everything and can not be preempted by anything. This scheduling class is used to migrate a task between processors. The IDLE scheduling class is another special scheduler that schedules a per-CPU idle task, which can be preempted by everything and is used to signal that the CPU is idle.

```
struct sched_class {
  const struct sched_class *next;

  // new task arrived -> set up first job release
  void (*task_woken) (struct rq *rq, struct task_struct *p);

  // task woke up after sleeping or blocking for IO
  // may release a new job if the task was sleeping
  // while waiting for the next job release
  void (*enqueue_task) (struct rq *rq, struct task_struct *p,
                        int flags);

  // currently executing task is going to sleep or block
  // -> ask the cpu to reschedule
  void (*yield_task) (struct rq *rq);

  // the main ``schedule'' function
  struct task_struct * (*pick_next_task) (struct rq *rq,
                                          struct task_struct *prev,
                                          struct pin_cookie cookie);

  // calculate actual execution time
  // if the task exhausted its budget -> ask the cpu to reschedule
  void (*task_tick) (struct rq *rq, struct task_struct *p,
                     int queued);
};
```

Listing 2: Abbreviated interface of a scheduling class in the Linux kernel.

The abbreviated interface of a scheduling class in Linux can be seen in listing 2.

There is no need to fully understand the scheduling class interface, the only thing of importance is the responsibility of each function. The following description may be slightly confusing since the Linux kernel does not make a distinction between a *task* and a *job*, and thus use the term *task* even when it actually refer to a *job*. Therefore most function names are confusing and unless we write otherwise, all functions work on a single *job* despite their name.

The first field in the scheduling class, the `next` field, is a pointer to the next scheduling class in the linked list traversed in listing 1. The `task_woken` function is invoked when a new *task* has arrived, and it is responsible for setting up the first job release of the task. `yield_task` is invoked when the currently executing job is going to sleep or block, in which case it should ask the CPU to reschedule. `enqueue_task` is invoked when a previously sleeping/blocked job has woken up, and it is responsible for adding the job to the ready queue or asking the CPU to preempt the currently executing job. If the *task* associated with the job was sleeping while waiting for the next job release, the `enqueue_task` function should release the next job of the task. The `task_tick` function is invoked periodically by a timer and is used to calculate the actual execution time of the currently executing job, it can therefore be used to enforce execution time budget. Finally, the `pick_next_task` function is the main scheduling function responsible for making all scheduling decisions, it is invoked when the CPU detects the “reschedule” interrupt flag and it should return the next job to schedule.

3.1.1 SCHED_RT Scheduler

The SCHED_RT scheduler in Linux uses an array of linked lists and a bitmap as the ready queue. Every entry (linked list) in the array is associated with a fixed task priority, i.e. the k-th linked list contains tasks with priority k. To efficiently find the highest priority ready task, the ready queue use a bitmap with one bit per priority level. A bit in the bitmap is cleared if the corresponding linked list is empty, and set if the linked list is non-empty. This ready queue is illustrated in figure 3.1. Modern processors have an instruction called `ffs` which finds the first bit set in a 64 bit bitmap (on 64-bit processors). The SCHED_RT scheduler allow for 100 different priority levels.

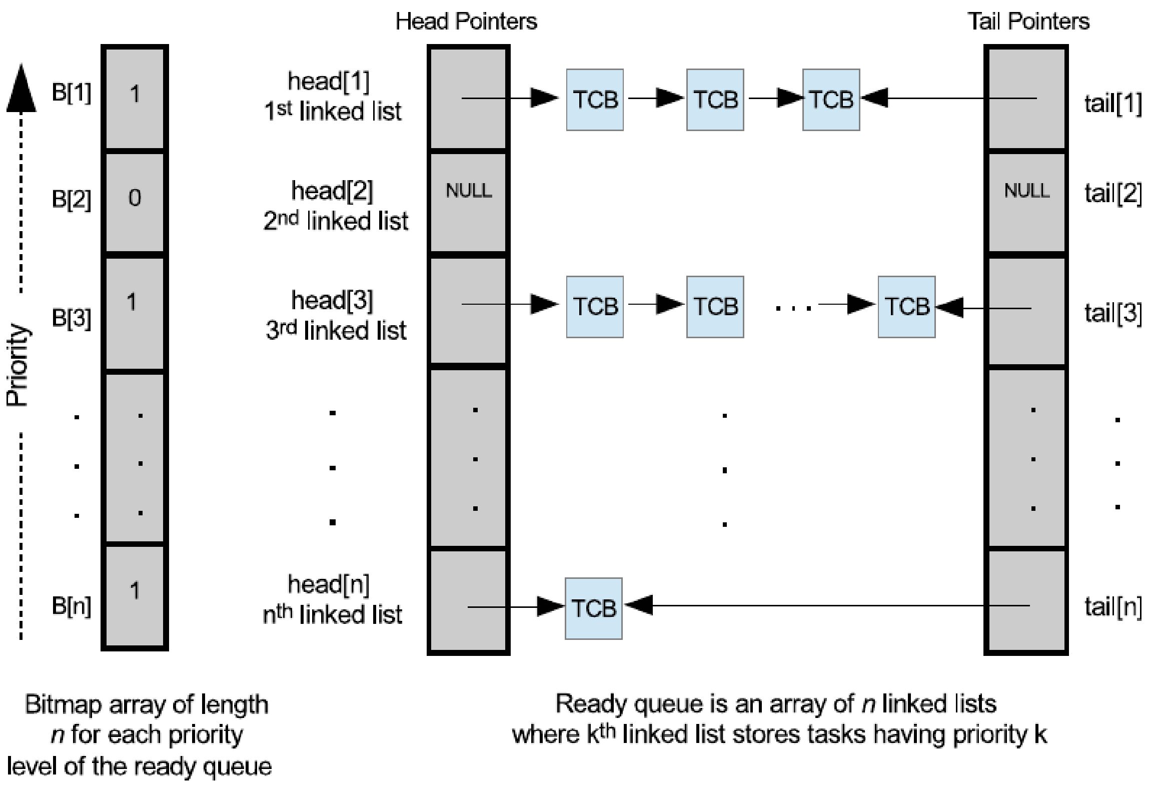


Figure 3.1: The ready queue used by FPP and the SCHED_RT scheduler in Linux. Figure taken from [15].

The SCHED_RT scheduler can schedule tasks with two different scheduling policies: First In, First Out (FIFO) or Round Robin (RR). Regardless of which, the scheduler first finds the highest priority non-empty linked list. For tasks with the FIFO policy, the scheduler then executes the first task in the linked list until completion and removes it from the list, while tasks with the RR policy are only allowed to execute for a fixed interval of time, after which they are appended to the end of their linked list again (unless the task finished executing during the time interval). When one linked list becomes empty, the SCHED_RT scheduler finds the next (lower priority) non-empty linked list and repeats the procedure. The functionality of the SCHED_RT scheduler can be summarized by the pseudocode in listing 3.

1. Find the highest priority non-empty linked list.
2. Fetch and remove the first task in the list.
3. If the task has policy FIFO: execute the task until completion.
4. If the task has policy RR: execute the task for a short interval of time, then append it to the end of the list again.
5. Goto step 1.

Listing 3: Pseudocode for the SCHED_RT scheduler in the Linux kernel.

3.1.2 SCHED_DEADLINE Scheduler

The Linux kernel includes a global EDF scheduler called SCHED_DEADLINE. This scheduler uses a red-black tree as the ready queue. The decision to use a red-black tree instead of a heap seems to be out of convenience rather than optimization [13]. The heap data structure in the Linux kernel is implemented on top of a fixed size array. Since the scheduler does not know how many jobs that may arrive, it needs a data structure with dynamic size. Thus, the kernel developers chose to use the red-black tree, which was also used by the CFS at the time, instead of adding another heap implementation based on pointers to the Linux kernel [13].

In addition to being a global EDF scheduler, SCHED_DEADLINE also implements a Constant Bandwidth Server (CBS) to provide temporal protection and real-time guarantees for sporadic tasks. Every period, the task is reserved and guaranteed a specified amount of execution time. Tasks may execute earlier and consume of their future reserved time if the system is idle. Tasks are also assigned a server period. By assigning an execution time larger than or equal to the WCET of the task, and a server period smaller than or equal to the task inter-arrival time, the task is guaranteed by the CBS to complete before its deadline without interference from other tasks [2].

3.2 Linux Testbed for Multiprocessor Scheduling in Real-Time System & Related Work

In Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (LITMUS^RT), a custom LITMUS scheduling class is added at the top of Linux hierarchical scheduling classes, see table 3.2. Thus, the LITMUS scheduling class will be invoked first when picking the next job to schedule. Instead of letting each task decide which LITMUS^RT scheduler it wants to use, LITMUS^RT allows only one scheduler plugin to be active at a time. This simplifies the implementation of new schedulers slightly, but it also removes the flexibility that different tasks can be scheduled simultaneously by different schedulers. The LITMUS scheduling class invokes the currently active LITMUS^RT scheduler plugin, which will schedule all tasks with the LITMUS policy.

1	LITMUS
2	STOP
3	DL
4	RT
5	CFS
6	IDLE

Table 3.2: The hierarchical scheduling classes in LITMUS^RT.

LITMUSˆRT also defines a new set of task and job parameters that are not present in Linux. Tasks with this extra set of parameters are hereby called LITMUSˆRT tasks. This is done to make the outwards interface more similar to the Liu and Leyland task model. However, from this follows that the Linux schedulers are not compatible with LITMUSˆRT tasks, but also that LITMUSˆRT scheduling plugins are not compatible with vanilla Linux task sets. In contrast to Linux tasks that have a rather sparse set of parameters, a WCET, period, deadline and fixed priority, LITMUSˆRT tasks also have an offset, a CPU (in the case of partitioned scheduling), and flags that indicate if they are periodic or sporadic, soft or hard real-time. LITMUSˆRT also add the derived job parameters (see section 2.2) which are not present at all in Linux: job number, release time, actual execution time, absolute deadline, and the extent of an eventual deadline miss.

3.2.1 Release Queue

To set up and queue future time-based releases, LITMUSˆRT has a release queue. When a job of a periodic task has finished executing, it is passed to the currently active LITMUSˆRT scheduler plugin, which is responsible for triggering the next job release. To minimize code duplication, this is handled by a release queue shared by all LITMUSˆRT scheduler plugins. The release queue internally use high-resolution hardware timers to trigger the job releases. However, instead of using one timer per job, LITMUSˆRT use one timer per (absolute) release time together with a release heap to store the jobs if several jobs are to be released at the same time instant. When a new job with release at time t is passed to the release queue, it checks if a release heap and timer for time t exists, in which case the new job is added to the existing release heap. Otherwise, a new release heap is created and a new timer is armed for time t. When a timer expires, the corresponding release heap is passed to the scheduler plugin via the `release_jobs()` function.

3.2.2 Global EDF with Synchronization Support

LITMUSˆRT includes another global EDF scheduler called Global EDF with synchronization support (GSN-EDF). This scheduler use binomial heaps both as the ready queue and as release heaps to minimize the release overhead. When new jobs are released, the (newly) released jobs in the release heap must be merged with the jobs already in the ready queue. Binomial heaps can be merged in $O(\log n)$ time, while binary heaps require $O(n)$ time. The ready queue is protected by a single course-grained lock. The scheduler is event driven and is thus invoked when a job is released, preempted, finishes execution, sleeps, or blocks. GSN-EDF also implements the Flexible Multiprocessor Locking Protocol (FMLP) to synchronize access to shared resources, a detailed description of FMLP can be found in [6].

3.2.3 Feather Trace

Feather Trace is a logging framework, that can be used to record overhead data. It contains functions to log the start and end times of a release, scheduling decision, and context switch. Every function writes the type of event, e.g. “SCHED_START”, together with the current timestamp to a log file. To avoid logging data for non real-time processes, the functions in Feather Trace only write to the log files if the current task has policy LITMUS. Feather Trace is designed to generate less overhead than traditional logging, which is accomplished by using one log file per CPU to avoid the contention over a common log file that otherwise occur. It also writes the data in binary instead text format, and writes the log file entirely into memory to avoid accessing secondary storage [9]. In addition to the logging functions, Feather Trace also includes a few scripts that can be used to post-process the generated log files and extract statistics.

3.2.4 The Schedulability Test Collection And Toolkit

The Schedulability Test Collection And Toolkit (SchedCAT) is a collection of schedulability tests, see section 2.4. Since one schedulability test does not necessarily dominate another, SchedCAT includes several schedulability tests for each type of scheduler. For EDF it includes six different schedulability tests for hard real-time task sets, and an additional test for soft real-time task sets. SchedCAT can also generate task parameters at random according to different probability distributions, see section 4.2.

4 Prior work

This chapter presents the theoretical work that this thesis builds upon.

4.1 Fixed Priority Scheduling with Priority Promotions

A problem with dynamic priority schedulers, such as EDF, is that they cause significant runtime overhead compared to fixed-priority schedulers, like RM. This problem was solved in theory by Pathan in 2015 [15] by combining a fixed-priority scheduler with priority promotions in order to exhibit the behaviour of a dynamic priority scheduler. A priority promotion point is a point in time where the priority of a job is promoted/increased. If assigned carefully, such a promotion point can promote the priority of a job when it is close to its deadline, such that it receives higher priority than one of the currently executing tasks and thus preempts that task.

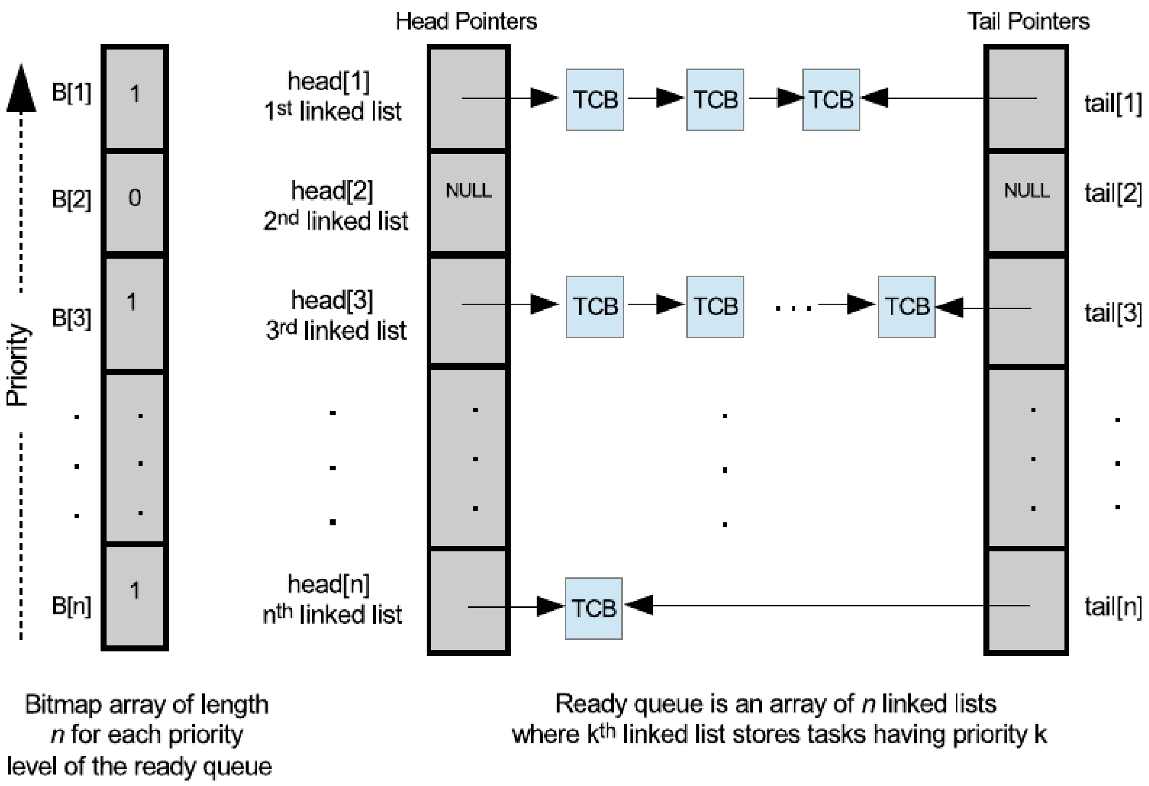
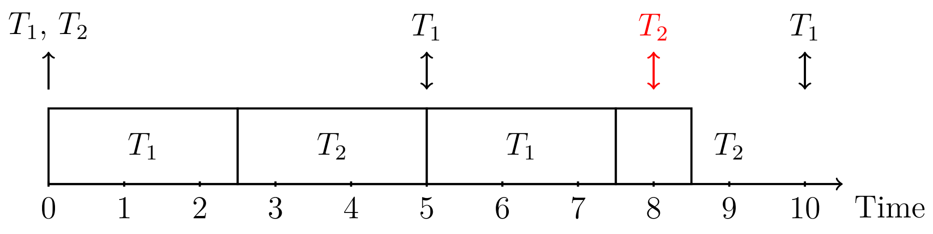


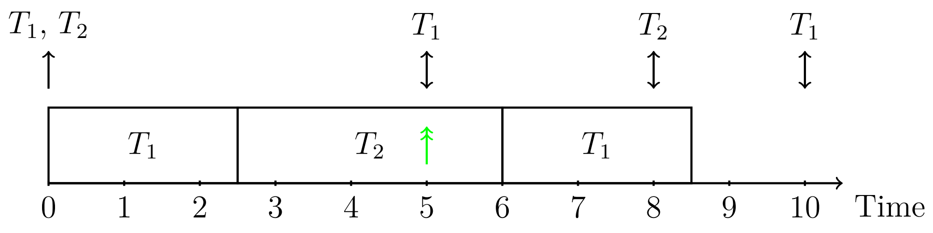
Figure 3.1: The ready queue used by FPP and the SCHED_RT scheduler in Linux. Figure taken from [15]. (Figure repeated from page 20)

The resulting algorithm, FPP uses the same ready queue as the SCHED_RT scheduler in Linux: an array of linked lists together with a bitmap to quickly search the array and find the non-empty linked lists, see section 3.1.1 and figure 3.1. What makes FPP unique is the three ready queue operations that enables it to generate EDF compatible schedules: `release_preempt()`, `release_no_preempt()` and `idle_remove()`.

Task	Period	WCET
T_1	5	2.5
T_2	8	3.5



(a) Rate-monotonic scheduling



(b) Rate-monotonic with priority promotion

Figure 1.1: The execution of a task set with only two tasks, where task T_1 has a shorter period and thus higher priority under RM. Each rectangle represents the execution of a single job of the task and the down-arrows denote the deadline of a job. The green arrow denote a priority promotion of task T_2 such that it can no longer be preempted by T_1, and thus finishes before its deadline. (Figure repeated from page 2)

In his papers [15, 16, 17], Pathan presents three slightly different implementations of FPP. The first two versions use timers to trigger a separate `priority_promotion()` operation [15, 17]. The third and final version drops the timers and instead handles the priority promotions in the `release_no_preempt` operation [16]. See figure 1.1 for an illustration, the priority promotion of task T_2 is triggered by the release of the second job of T_1 at time $t = 5$. In this thesis work two implementations of FPP were evaluated, one with timers and one without. The implementation without timers quickly proved to perform better, thus the implementation with timers was discarded and not considered further. Therefore only the ready queue operations of the third and final version of the FPP scheduler is described below, since this is the version of FPP that were evaluated later in the thesis.

The first ready queue operation, `release_preempt()` is invoked when a new job should preempt one of the currently executing jobs. This is done by first finding the DM priority of the currently executing job, k, and the least set bit in the bitmap, l. If l is an invalid index, i.e. $l < 0$ or $l \geq n$, then set $l := n + 1$. It then calculates the minimum of these, $p = min(k, l)$, and inserts the currently executing job into the ready queue by prepending it to the p^{th} linked list, and setting the corresponding bit in the bitmap, $B[p] := 1$, to specify that the linked list is non empty.

The second ready queue operation, `release_no_preempt()` is invoked when a new job, with DM priority k, should not preempt the currently executing job. This operation is done in four steps:

Step 1: Find the index h of the next higher-indexed non-empty linked list after the k^{th} linked list, i.e. find the minimum h such that $B[h] = 1$ and $h > k$. This is done by calling the `find_next_bit()` function. If h is an invalid index, i.e. $h < 0$ or $h \geq n$, then jump to step 4.

Step 2: If the last element in the h^{th} linked list has a smaller absolute deadline than the new job, then all jobs in the h^{th} linked list must have smaller deadlines and thus higher EDF priority then the new job. The entire h^{th} linked list is moved to the end of the k^{th} linked list. This process is repeated by jumping to step 1.

Step 3: If the last element in the h^{th} linked list has a larger absolute deadline than the new job, then some jobs in the h^{th} linked list may still have smaller deadline than the new job. Thus, we must iterate over the h^{th} linked list and move only the elements that have smaller deadlines to the end of the k^{th} linked list. As soon as an element with larger deadline is found, this process stops since the elements are sorted according to their absolute deadlines.

Step 4: The new job is inserted at the end of the k^{th} linked list, and the corresponding bit in the bitmap is set, $B[k] := 1$, to specify that the k^{th} linked list is not empty.

The third ready queue operation, `idle_remove()` is invoked when the currently executing job has finished and the CPU becomes idle. `idle_remove` finds the lowest-indexed non-empty linked list, then removes and dispatches the first element in that linked list. If the linked list is now empty, the corresponding bit in the bitmap is cleared, $B[k] := 0$.

4.2 On the Implementation of Global Real-Time Schedulers

In [10], Brandenburg and Anderson evaluated 7 different implementations of EDF and this evaluation forms the basis of the method used to evaluate the schedulers in this thesis, see section 6. In [10], the runtime overhead of each of the 7 schedulers were obtained by tracing periodic task sets with 50 to 450 tasks in steps of 50, using Feather Trace. For each such task set size, 10 task sets were generated and traced for 30 seconds each. The periods of the generated tasks follow a “Uniform moderate” distribution and the utilizations follow a “Uniform light” distribution, see below.

After acquiring the runtime overheads of each scheduler, these were used to assess how the overheads of each scheduler affects the schedulability of randomly generated task sets. Task sets were generated using three period distributions and six utilization distributions. For each combination of period and utilization distribution, task sets were generated by generating tasks until a specified utilization cap (in the range $[0, m]$) was reached, and for each such utilization level, 1000 task sets were generated. Before testing the schedulability of a task set, the task parameters were adjusted to account for the overhead of the currently analyzed scheduler. This work later resulted in SchedCAT, see section 3.2.4.

The three period distributions were:

- Uniform short $[3ms, 33ms]$
- Uniform moderate $[10ms, 100ms]$
- Uniform long $[50ms, 250ms]$

and the six utilization distributions were:

- Uniform light $[0.001, 0.1]$
- Uniform medium $[0.1, 0.4]$
- Uniform heavy $[0.5, 0.9]$
- Bimodal light $[0.001, 0.5]$ with prob. $^8/_9$ or $[0.5, 0.9]$ with prob. $^1/_9$
- Bimodal medium $[0.001, 0.5]$ with prob. $^6/_9$ or $[0.5, 0.9]$ with prob. $^3/_9$
- Bimodal heavy $[0.001, 0.5]$ with prob. $^4/_9$ or $[0.5, 0.9]$ with prob. $^5/_9$

A *uniform* distribution means that the numbers generated by the random function are uniformly distributed over the interval, i.e. all numbers in the interval are equally likely to be generated by the random function. A *bimodal* distribution means that the random function is first used to pick one of the two intervals, and then used to generate a uniformly distributed number within that interval.

5

Implementation

This chapter presents the design and implementation details of the implemented schedulers, starting with the common functionality shared by all the implemented schedulers. Then the implementation of the “standard” EDF scheduler is described, followed by the second EDF scheduler implemented. The chapters end with the description of the implemented FPP scheduler.

5.1 Common Functionality

The code to assign jobs to processors at runtime is shared by all three implemented schedulers. They all store the CPUs in a heap sorted by the deadline of the currently executing job on each CPU. At the top of the heap is the CPU currently executing the lowest EDF priority job. This enables us to efficiently check if a new job should preempt any of the currently executing jobs. If the new job has lower priority than the current lowest-priority job, then it should not preempt any of the currently executing jobs. If it has higher priority than the current lowest-priority job, then that job is preempted, the deadline associated with the CPU is updated and the heap is rebalanced. Instead of $O(m)$ for the preemption check, where m is the number of processors, we now get $O(1)$ with the tradeoff that we must rebalance the heap, $O(\log m)$, whenever a new job preempts.

All three implemented schedulers support batching of simultaneous job releases via a release queue. The implementation of this release queue roughly follow the description by Brandenburg in [7], and is thus similar to the release queue in LITMUS^RT, see section 3.2.1. The main difference is that our release queue use unordered linked lists instead of the binomial heaps used in LITMUS^RT. This decision to use unordered linked lists was done to simplified the implementation of the schedulers. Instead of implementing an efficient but complicated merge operation, we simply iterate over the linked list, decide if each job should preempt or not, and call the corresponding release function. This design decision does however increase the release overhead of our schedulers since the release queue is not efficiently merged with the ready queue in the same way that binomial heaps can be merged.

Since the batching done by the release queue was added as a small optimization and is not part of the core functionality of the schedulers, it is turned off by default and each task must signal to the scheduler if it wants to use the release queue. With batching turned off, the schedulers default back to the core functionality provided by Linux, which does not distinguish between a task sleeping for a short amount of time, or waiting for the next job arrival. The task must arm a high-resolution timer, setting up the next job release and suspending itself by yielding the CPU. When a job yields, the task is gone from the perspective of the schedulers. Later when the timer expires and the task wakes up again, it is therefore treated as a completely new task by the scheduler.

All three implemented schedulers also support budget reinforcement, which is a subset of the functionality provided by a full CBS, and can be used to provide temporal isolation between tasks. Similarly to a CBS, every job is allocated a budget, i.e. reserved a specified amount of execution time. When a job has executed for its budget, it is preempted and discarded to avoid it from interfering with the execution of other jobs. This is done by arming a high-resolution timer when the job starts executing, which triggers the `pick_next_task()` function of the schedulers upon expiry. The `pick_next_task()` function determines that the job has executed for its budget and then acts the same way as if the job finished execution. The job parameters are replaced by those of the next job, thus reusing the allocated memory, and then the next job is passed to the `release` function of the scheduler.

5.2 EDF with Binary Heap Scheduler

A “standard” EDF scheduler was implemented in order to see how the FPP scheduler would perform compared to an EDF scheduler implemented by the same developer. Some developers are more skilled and write better code than others, thus the performance of a scheduler is highly influenced by the proficiency of the developer implementing it. The purpose of the EDF with Binary Heap (EDF-BH) scheduler is to compare the EDF algorithm to the FPP algorithm, and to avoid comparing an implementation of EDF by a very senior developer with an implementation of FPP by a junior developer. This is the reason for implementing the EDF-BH scheduler instead of using one of the global EDF schedulers already available, such as the GSN-EDF scheduler in LITMUSˆRT. To avoid influence from [10] in particular, the EDF-BH scheduler was developed before reading the paper.

In addition, the implemented EDF scheduler will be used to assess how optimized SCHED_DEADLINE is. Thus, the EDF-BH scheduler was developed to be as simple and straightforward as possible. A binary heap was chosen as the ready queue of the EDF scheduler since it seems to be the most common data structure used by EDF schedulers in literature. The global ready queue is protected by a single lock to avoid issues stemming from concurrent access.

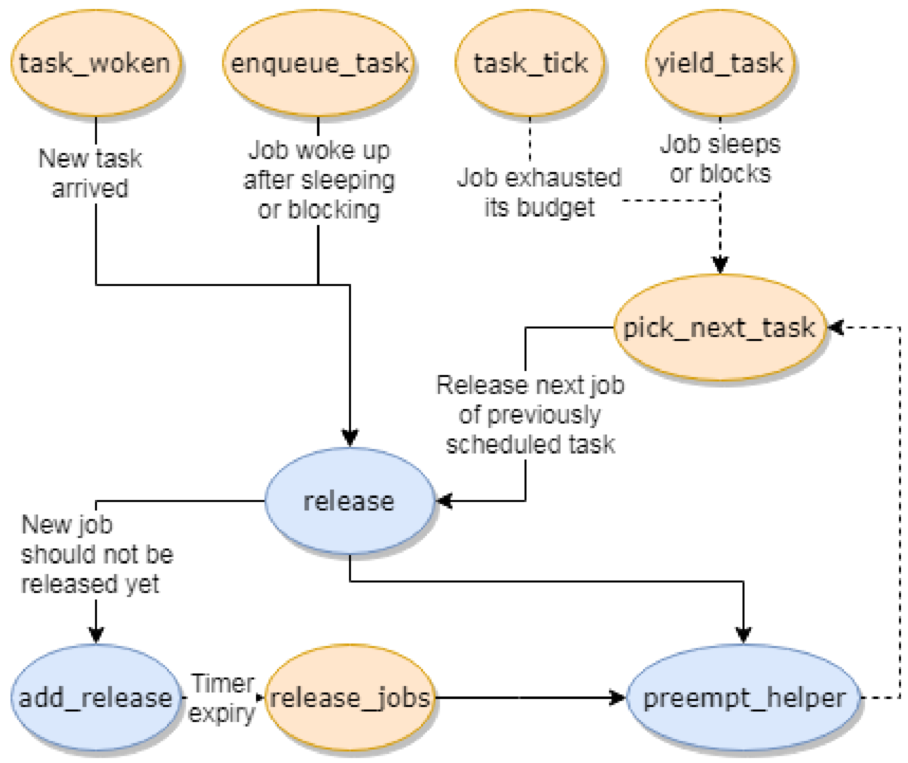


Figure 5.1: Simplified flowchart of the EDF-BH scheduler. The orange squares denote the event driven scheduler entry points, and the blue squares denote internal helper functions.

The EDF-BH scheduler is an event-driven scheduler. In contrast to the FPP scheduler by Pathan [15, 16, 17], the EDF-BH scheduler does not have a specialized `release_no_preempt()` function. When a new job is released, the common `release()` function check if the job should preempt any of the currently executing jobs, otherwise it is added to the ready queue. If it should preempt, a helper function similar to `release_preempt()` is invoked that does the following:

- First, the currently executing job is added to the ready queue.
- Second, the new job is "linked" to the CPU (LITMUS^RT terminology: In Linux only the `pick_next_task()` function may make scheduling decisions. To circumvent this, other functions temporarily store the result of their most recent scheduling decision as a "linked" job).
- Third, the scheduler asks the CPU to preempt the currently executing job by setting the "reschedule" interrupt flag.

If the currently executing job is successfully preempted, it invokes the `pick_next_task()` function. The currently executing task may be executing a critical section with preemptions disabled on the local CPU, thus we cannot preempt it immediately and instead ask the CPU to preempt by setting the "reschedule" interrupt flag. The

CPU later preempts the currently executing job when it exits its critical section and enables preemptions again.

Note that the name of the `release()` function is a bit misleading since it may be called for jobs that have a release time in the future and should not be released yet. If this is the case, the `release()` function passes the job to the release queue instead. When the release timer expires, the release queue pass the job back to the scheduler via `release_jobs()`. The difference between `release_jobs()` and `release()` is that `release_jobs()` receive a release heap that may contain several jobs, while `release()` only receive a single job at a time.

When the currently executing job is preempted, finished execution, exhausts its budget, sleeps, or blocks, the job invokes the `pick_next_task()` function. The `pick_next_task()` function first check if the currently executing job, or the next job of the task, should be added to the ready queue. Then, if a job is linked to the CPU, it schedules the linked job. If no job is linked to the CPU, i.e. the processor becomes idle, then the scheduler schedules the job at the top of the heap and removes it from the heap. This is similar to the `idle_remove()` operation of the FPP scheduler, see section 4.1, except with a different data structure.

After implementing the EDF-BH scheduler, it can be concluded that it turned out to be conceptually similar to the GSN-EDF scheduler in LITMUS^RT. Both are event driven, use a heap as the ready queue and protect the ready queue with a single lock. The main difference is that GSN-EDF use a binomial heap, while the implemented EDF scheduler use a binary heap, and that the GSN-EDF scheduler can only schedule LITMUS^RT tasks. The GSN-EDF scheduler also provide support for synchronizing access to shared resources by implementing the FMLP [6], see section 3.2.2. Any such synchronization support is not provided by EDF-BH or any of the other implemented schedulers. Task set designers are recommended to use an external library if synchronization support is needed.

5.3 EDF with Red-Black Tree Scheduler

The first results from the benchmarks showed that the runtime overheads differed so much between the implemented schedulers and SCHED_DEADLINE that they were incomparable. To identify if the performance of the SCHED_DEADLINE scheduler is due to an efficient ready queue (a red-black tree), or if it is unrelated to the data structure used as ready queue, a third scheduler was implemented. The EDF with Red-Black Tree (EDF-RBT) scheduler is identical to the EDF-BH scheduler, with the only difference that the binary heap has been replaced by the red-black tree used by the SCHED_DEADLINE scheduler. As the SCHED_DEADLINE scheduler, EDF-RBT cache the leftmost node in the red-black tree so that the highest EDF priority job can be accessed in $O(1)$ time.

5.4 FPP Scheduler

The FPP scheduler use the same array of linked lists and bitmap data structure as the Linux SCHED_RT scheduler. As the EDF-BH scheduler, the ready queue is protected by a single lock. A problem with this data structure is that we can not increase the size of the data structure at runtime, since allocating memory at runtime is very expensive. The FIFO/RR schedulers statically allocates 100 linked lists at compile time, but we chose to use 512 linked lists instead because each linked list in the FPP scheduler is initially associated with a unique (relative) deadline. Before any priority promotions have occured, the initial priorities of the tasks are assigned in DM order, and thus each linked list is associated with a unique relative deadline. If only 100 linked lists are used, this would allow for up to 100 tasks with unique relative deadlines. To support larger task sets, 512, which is the number of priority levels in LITMUS^RT, was chosen instead. If a task set designer has an even larger task set, the task set designer can easily change the single line in the source code that defines the size of the FPP ready queue and recompile the kernel.

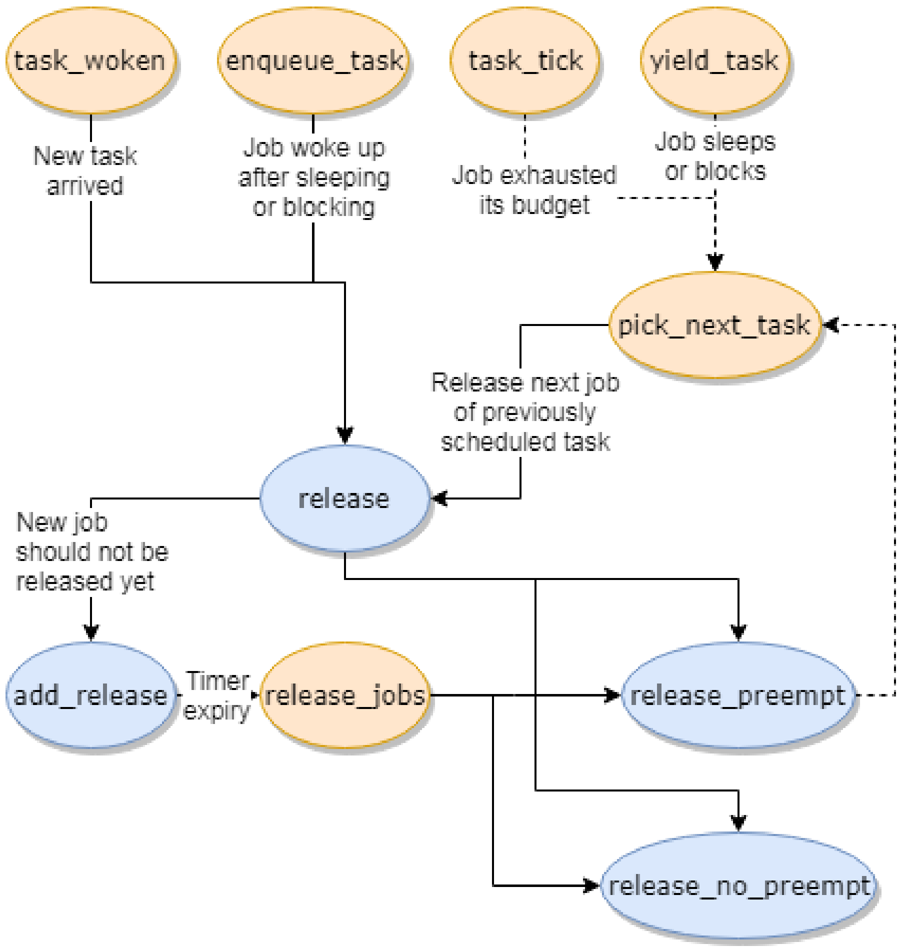


Figure 5.2: Simplified flowchart of the basic FPP scheduler. The orange squares denote the event driven scheduler entry points, and the blue squares denote internal helper functions.

The FPP scheduler has two different functions that are executed when a new job arrives depending on if the job should preempt or not. `release_preempt()` is called when the new job should preempt one of the currently executing jobs, and `release_no_preempt()` is called when the new job should not preempt any of the currently executing jobs. When a new job arrives, the common `release()` function determines if the job should preempt and then calls the correct function. The `release_preempt()` and `release_no_preempt()` functions closely follow the description by Pathan, see section 4.1 for a summary or [17] for complete details. The `release_preempt()` function only differ from the description by Pathan in that it does not directly preempt the currently executing job, but instead links the newly arrived job to the CPU and asks the CPU to preempt by setting the "reschedule" interrupt flag, just as the EDF-BH scheduler does.

The `pick_next_task()` function is nearly identical to that of the EDF-BH and EDF-RBT schedulers, the only difference is that we call `idle_remove()` if no job is linked to the CPU and the CPU becomes idle.

6

Evaluation Methods

This chapter presents the methods used to evaluate and compare the implemented schedulers. The evaluation was done in three steps. First the schedule generated by the schedulers was verified, second the runtime overheads caused by the schedulers were quantified and analyzed, third and finally the acquired runtime overheads were used to assess the schedulability of randomly generated task sets when scheduled by the schedulers.

6.1 Schedule Verification

To verify that the FPP scheduler generates a schedule identical to SCHED_DEADLINE, a small verification script was written. The script reads a schedule generated by each of the schedulers when running the overhead benchmarks, see section 6.2, and check the schedule against a simple EDF decision test. Every time a scheduling decision was made, the verification script check that one of the current m earliest-deadline jobs was scheduled.

6.2 Runtime Overheads

The runtime overheads caused by the schedulers were evaluated using the same uniform moderate periods and uniform light utilizations as Brandenburg and Anderson [10], see section 4.2 for more information.

A dummy periodic task was written that takes a pair of task parameters, waits for a start signal, and then alternates between spinning for its WCET and sleeping until the next release of the next job. The LITMUS^RT User-Space Library (liblitmus) includes a similar dummy task called “rtspin” which only works under LITMUS^RT plugins. However, it was deemed as more time-consuming to rewrite rtspin, such that it works under the standard Linux schedulers than to simply write a new dummy task from scratch.

For each task set size between $[m, 20m]$ in steps of m, the benchmark script calls SchedCAT to generate the task parameters for n tasks. The periods of the tasks are “Uniform moderate”, i.e. uniformly distributed between $[10ms, 100ms]$, and the utilization of the tasks are “Uniform light”, i.e. uniformly distributed between $[0.001, 0.1]$, see section 6.3. For each scheduler, the script initializes the n dummy tasks with the given parameters, starts Feather Trace to collect overhead samples, and then synchronously releases the dummy tasks. It then waits for 60 seconds before stopping Feather Trace and killing the tasks. The procedure of calling SchedCAT to generate a task set and tracing the task set under all four schedulers is repeated 10 times before proceeding to the next task set size. The reason for tracing 10 task sets for each size is to decrease the influence that the randomization of task parameters may have and therefore increase the level of confidence in the obtained overheads. For example, since the parameters of the tasks are random, it may happen that every task in a generated task set get the maximum period of $100ms$ which likely will not cause the maximum amount of overhead since the scheduler is invoked infrequently.

Before computing the worst- and average-case overheads, so called “outliers” were removed. When repeatedly measuring the overheads of a scheduler, some of the obtained samples may be “outliers”, i.e. samples appear that significantly differ from the general trend. For example, there may be a single sample showing an overhead ten times as large as the other samples. This is likely caused by the scheduler being interrupted by an external event during overhead measurement. Brandenburg found four causes of outliers [8], of which two have been solved and are no longer a problem when recording and analyzing overheads with Feather Trace. The remaining two causes are:

- Interrupts in between recorded timestamps.
- Preemptions in user space before reporting a timestamp to the kernel.

While these have not been solved, Feather Trace does export an interrupt count together with each recorded timestamp such that these outliers can be detected and manually removed.

6.3 Schedulability

The schedulability of the schedulers were evaluated using randomly generated task sets, with the same three period distributions and six utilization distributions as Brandenburg and Anderson [10], see section 4.2. SchedCAT includes functions for generating task parameters according to these distributions. For example, if called with “uni-moderate” periods and “uni-short” utilizations, the resulting task set will follow these distributions and the parameters of one task in the task set may be a period of $3ms$, a utilization of 0.001, and thus a WCET of $3ms * 0.001 = 0.003ms$. For each combination of period and utilization distribution, task sets were generated by generating tasks until a specified utilization cap (in the range $[0, m]$) was reached, and for each such utilization level, 1000 task sets were generated.

For each of the 18 different combinations of period and utilization distribution, task sets were generated by creating tasks until a specified utilization cap (in the range $[0, m]$) was reached. The initial sampling point range from 0 to m in steps of 1, and then additional sampling points are chosen adaptively using the algorithm from "The Yacas Book of Algorithms" [18] with a maximum depth of 10, such that the density of sampling points is high where the curve changes rapidly. For each sampling point, 1000 task sets were generated.

As in [10], we are unlikely to capture the true worst-case overhead in a Linux-based system. Thus, we have to interpret "HRT" as deadline are *almost never* missed, and "SRT" as deadline tardiness is bounded *on average*. With those slightly softer definitions, our acquired worst-case overheads can be used when testing for hard real-time schedulability, and our acquired average-case overheads when testing for soft real-time schedulability.

Before testing the schedulability of a task set, the parameters of each task were adjusted to account for the overhead of the currently analyzed scheduler using SchedCAT, which implements the rather complicated equations at the bottom of page 262 in Brandenburgs Dissertation [7]. After inflating the parameters of each task, the task set was tested to see if it passed the schedulability tests. Since we cannot know which schedulability test the task set designers may use, a task set was deemed as "schedulable" if it passed at least one of the schedulability tests available in SchedCAT, and "non-schedulable" otherwise. This was repeated for each of the schedulers before a new task set was generated.

Note that SchedCAT post-process the runtime overheads in order to acquire a monotonically increasing continuous function $f(n)$ for each overhead. The first step of this post-processing is to make the overheads monotonically increasing with respect to task set size. A monotonically increasing function g is a function where $g(m) \leq g(n)$ for all n and m where $m \leq n$. In order to make a function f monotonically increasing we can simply define g as $g(x) = max\{f(x), f(x-1)\}$ with $g(0) = f(0)$. Note that this is a recursive definition, if f has a maximum at $f(0)$ then g will be a constant, $g(x) = f(0)$. This step is done because a large task set may invoke the scheduler more frequently and thus the data structures of the scheduler are more likely to exploit temporal locality and remain cached, which in turn may reduce the observed overheads compared to a smaller task set. However, since caches are largely unpredictable, we cannot depend on them to always decrease the runtime overheads when scheduling large task sets. Therefore, we assume that when analyzing the schedulability of a task set, it may suffer from the same runtime overheads as smaller task sets.

The second step of the post-processing is to create a continuous function from the discrete points. SchedCAT does this in the simplest possible way, by piecewise linear interpolation between the discrete points. Piecewise linear interpolation works like this: to get the overhead at a point x that is between two discrete points n and m, you first calculate the slope k of the curve between n and m, and the point m where this linear curve intercepts the y-axis. Then plug in x into the equation for a linear curve $y = kx + m$ to get the overhead y at the point x.

Using the methods presented in this chapter, the schedulers were evaluated and the results are presented in the chapter 7.

7

Results

This chapter presents the results acquired, and the structure of the chapter follows that of the method chapter. First is the verification of the schedule generated by the schedulers. Second is the runtime overheads caused by the schedulers, first the overheads on the x86 without batching of simultaneous releases, second the overheads on the x86 with batching, and third the overheads acquired on the ARM board. Third and finally is the schedulability of randomly generated task sets when scheduled by the schedulers.

7.1 Schedule Verification

All four schedulers, EDF-BH, EDF-RBT, FPP, and SCHED_DEADLINE generated global EDF compatible schedules for periodic tasks with correct WCET. The only differences were observed for task sets that were clearly not schedulable, and where jobs missed their deadlines. Under these conditions, the schedules were no longer equal. For example, one scheduler could successfully schedule task A and B, but not task C, while another scheduler successfully scheduled C but not A or B. One reason for this is that the implemented schedulers do not discard a task if its WCET is longer than the remaining time until its deadline. Since task set designers are assumed to use one of the available schedulability tests, and thus do not try to schedule task sets that are definitely not schedulable, we conclude that the generated schedules are identical for the purposes of this thesis.

7.2 Runtime Overheads

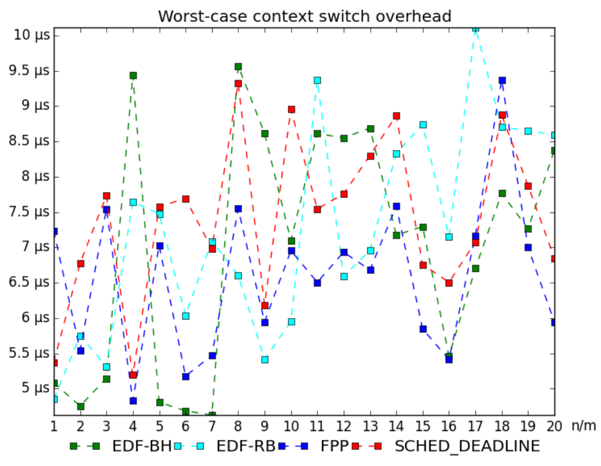
This section presents the runtime overheads of the schedulers observed on the Intel i3-4370 without batching, on the Intel i3-4370 with batching and on the Raspberry Pi 3 model B.

7.2.1 x86

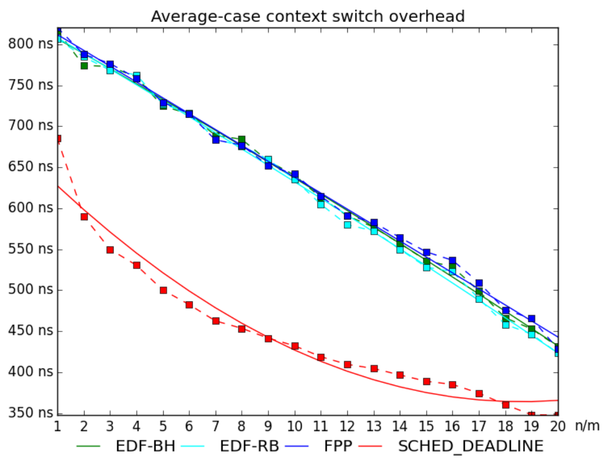
On the x86, approximately 15% of the recorded samples were found to be outliers that had been disturbed by interrupts, in comparison to only 1.5 ppm outliers on the ARM, see section 7.2.3. Thus a very large proportion of the collected samples had to be discarded in order to compare the schedulers. The large number of interrupts disturbing the schedulers cause implications not only for HRT scheduling but also for SRT scheduling, see section 8.2. After filtering all the outliers, it can be stated that the FPP scheduler perform better than the EDF-BH and EDF-RBT schedulers, see figure 7.1. The worst-case overheads, and the average-case overheads for task sets with many tasks, are lower for the FPP scheduler than the EDF-BH and EDF-RBT schedulers.

The worst-case context switch overhead, figure 7.1 (a), is a chaos, most likely caused by the caches. The average-case context switch overheads, figure 7.1 (b), are nearly identical for all implemented schedulers, but there is a small trend for the FPP scheduler to cause a slightly higher context-switch overhead. While the magnitude of the runtime overheads cause by the FPP scheduler are lower than those of the EDF-BH and EDF-RBT schedulers, the claim by Pathan [15, 16, 17] that FPP should perform fewer scheduling decisions and context switches than EDF could not be verified, see figure 7.2. All three implemented schedulers perform the same number of scheduling decisions and context switches.

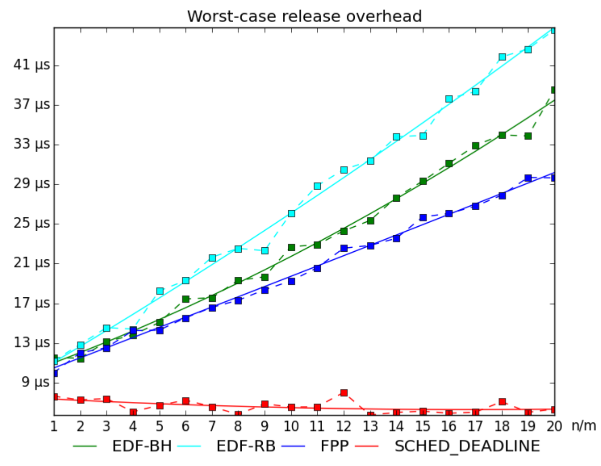
Except for the worst-case context switch overhead, figure 7.1 (a), the implemented schedulers consistently perform worse than SCHED_DEADLINE with respect to the magnitude of the overheads. This comparison is a bit flawed however, see section 8.1. Worth noting is that while the implemented schedulers cause higher runtime overheads than SCHED_DEADLINE, they also perform significantly fewer scheduling decisions than SCHED_DEADLINE, and thus also incur fewer context switches. See section 8 for a discussion.



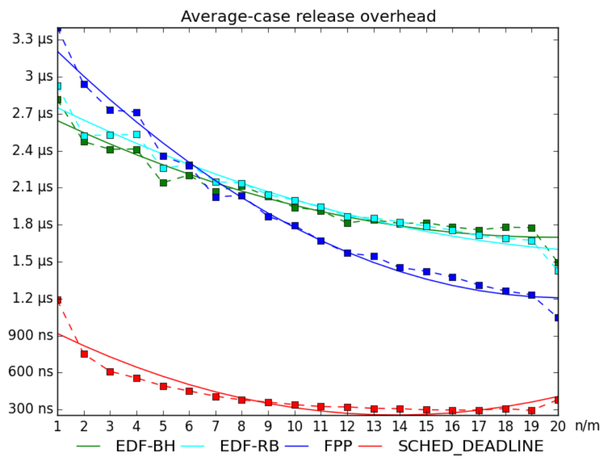
(a)



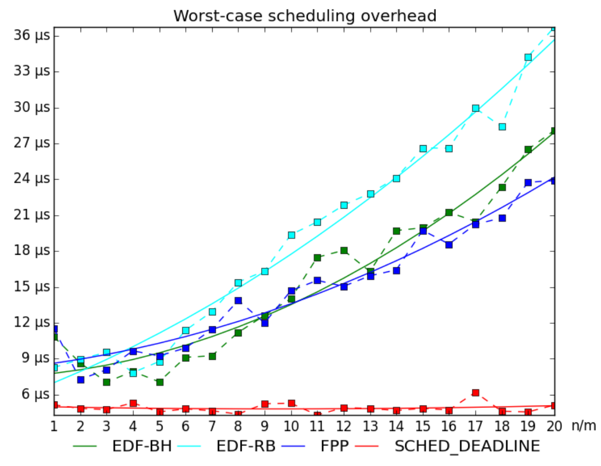
(b)



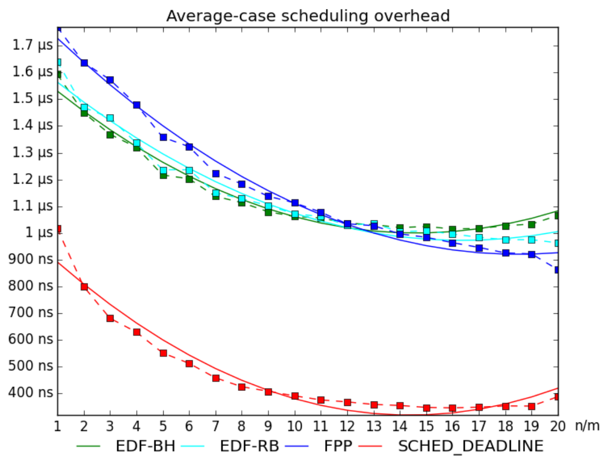
(c)



(d)

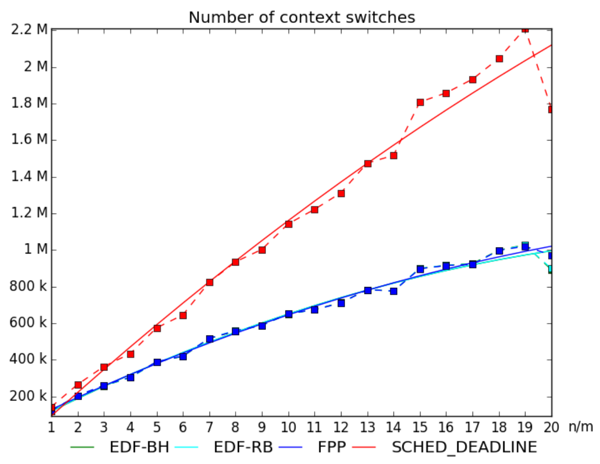


(e)

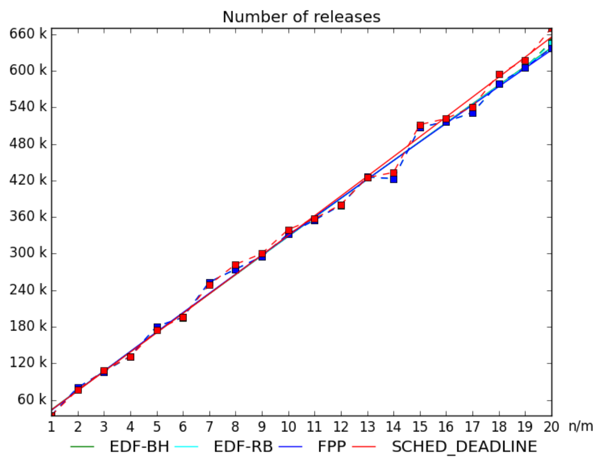


(f)

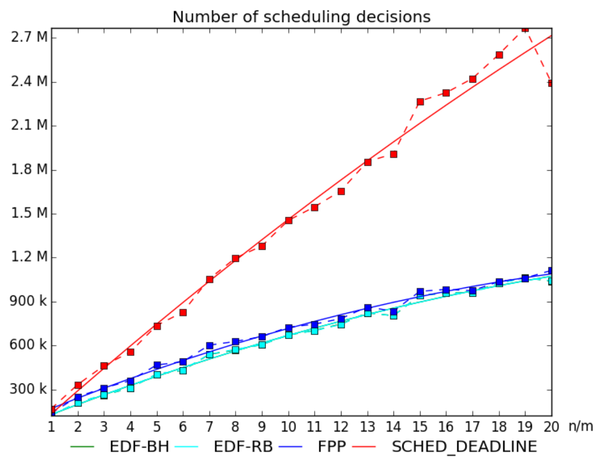
Figure 7.1: Worst-case (left), and average-case overheads (right). Note that outliers have been filtered before computing the worst- and average-case values.



(a)



(b)



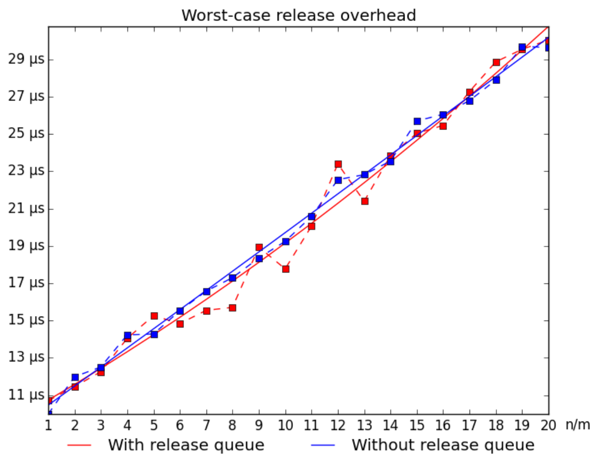
(c)

Figure 7.2: Total number of overheads experienced by the ten task sets. The y-axis represents the total number of overheads experienced by the ten task sets over the course of ten minutes.

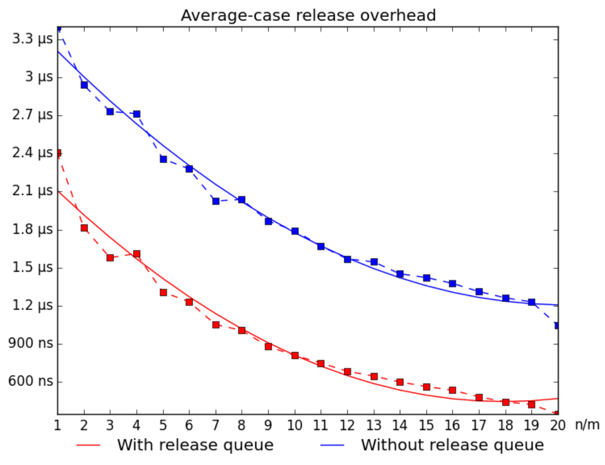
7.2.2 Batching Releases

Using the release queue to batch simultaneous releases dramatically reduce the number of times the release function is called, especially for large task sets, see figure 7.3 (c). This effectively reduces the average-case release overhead, figure 7.3 (b), while the observed worst-case release overhead remained unchanged, figure 7.3 (a). Because it releases a group of tasks instead of releasing the tasks one by one, this also leads to slightly fewer scheduling decisions and context switches, see figure 7.3 (d) and (e). If $2m$ tasks are releases simultaneously, releasing them one by one may cause up to $2m$ preemptions, while batching and releasing them as a group cause at most m preemptions. The scheduling and context switch overheads remained unchanged when batching was turned on.

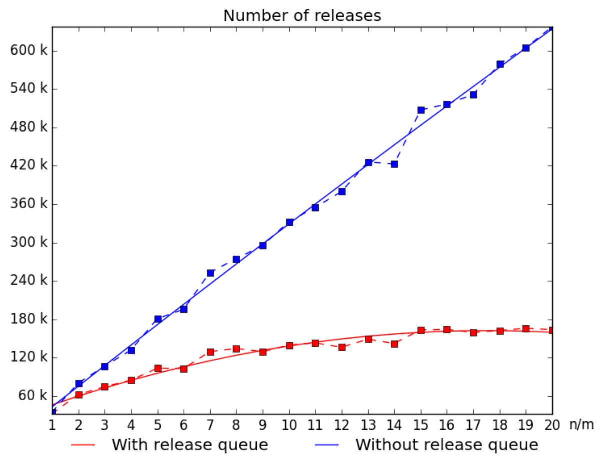
Since it would be misleading to compare the implemented schedulers with this batching functionality against SCHED_DEADLINE which does not have batching, all schedulability experiments were run using the overheads acquired with batching turned off. With batching turned on, the implemented schedulers perform better in the SRT schedulability tests because the average-case release overhead is significantly lower, but they are still slightly worse than SCHED_DEADLINE since the other overheads remain largely unchanged with batching turned on.



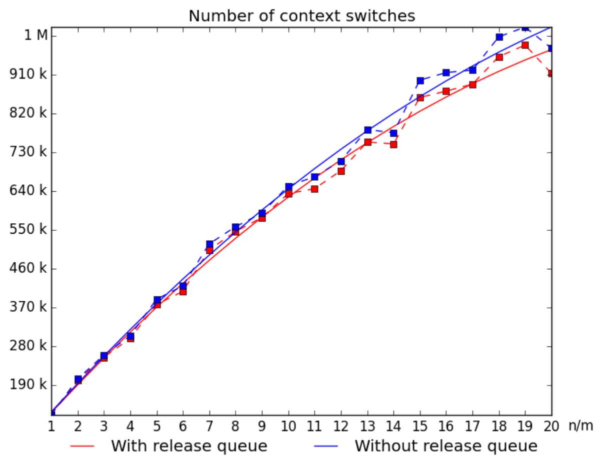
(a)



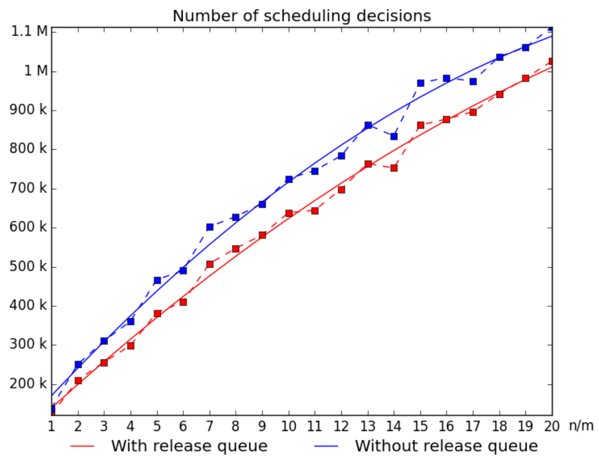
(b)



(c)



(d)



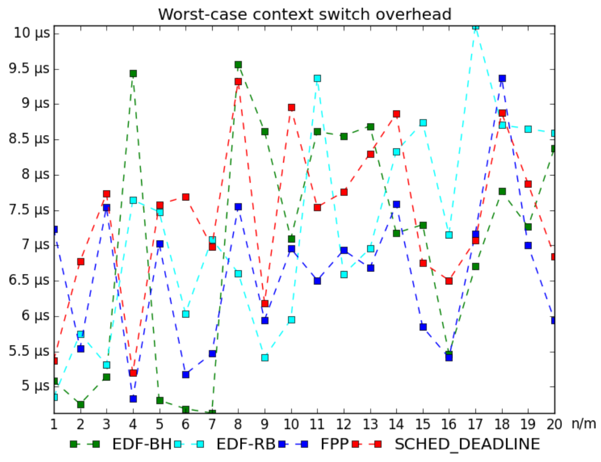
(e)

Figure 7.3: Comparison of the FPP scheduler with and without batching of simultaneous releases. The y-axis represents time in (a) and (b) and the total number of overheads experienced by the ten task sets over the course of ten minutes in (c), (d) and (e). Note that outliers have been filtered before computing the worst- and average-case values.

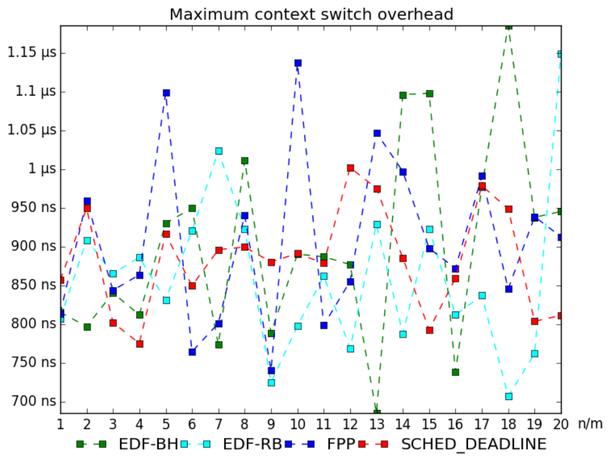
7.2.3 ARM

In contrast to the chaotic overheads obtained on the x86, the overheads observed on the Raspberry Pi were very consistent and with extremely few outliers. Per one million overhead samples recorded, there were only 1.5 outliers observed on the Raspberry Pi and the magnitude of these outliers were around one minute so they were clearly separated from the rest of the values.

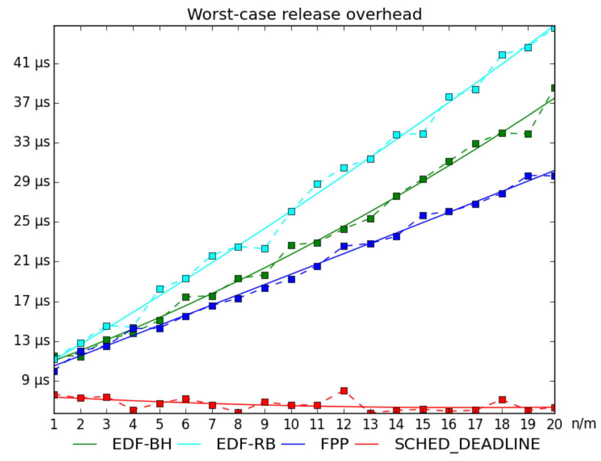
Another significant difference between the x86 and the ARM processor used in the Raspberry Pi is that the overheads observed on the x86 are in microseconds, while the overheads on the ARM are in nanoseconds! Despite that a huge number of outliers were filtered for the x86, the worst-case context switch overhead is still 8.5 times larger, the worst-case release overhead is approximately 60 times larger, and the worst-case scheduling overhead is 40 timer larger than the corresponding on the ARM. For the average-case overheads, the difference is less dramatic, the context switch overhead is 12 timer larger on the x86, the release overheads is 16.5 times larger, and the scheduling overhead is 8 times larger than the ARM.



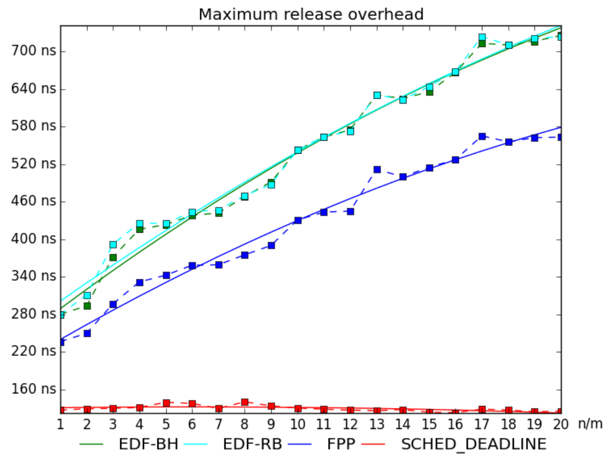
(a)



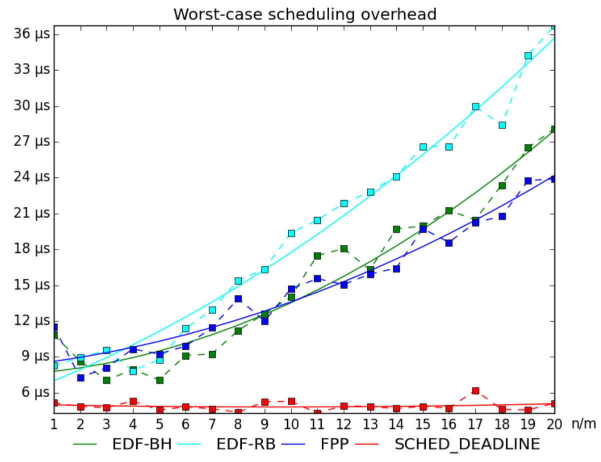
(b)



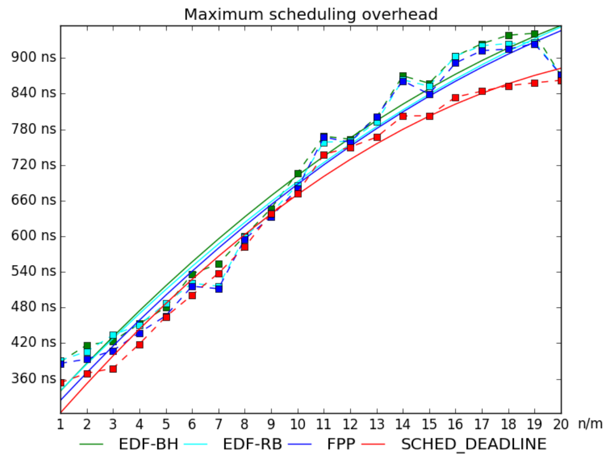
(c)



(d)

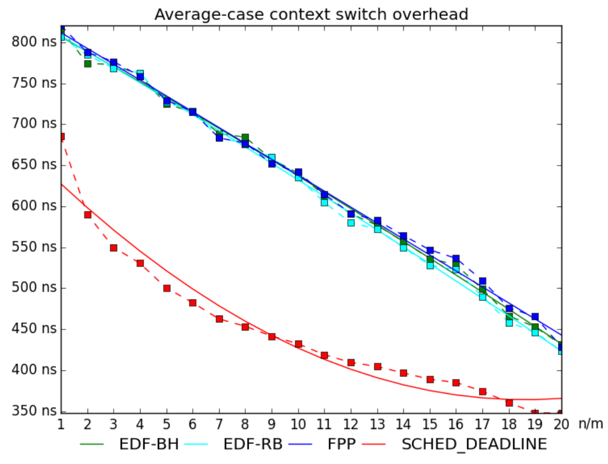


(e)

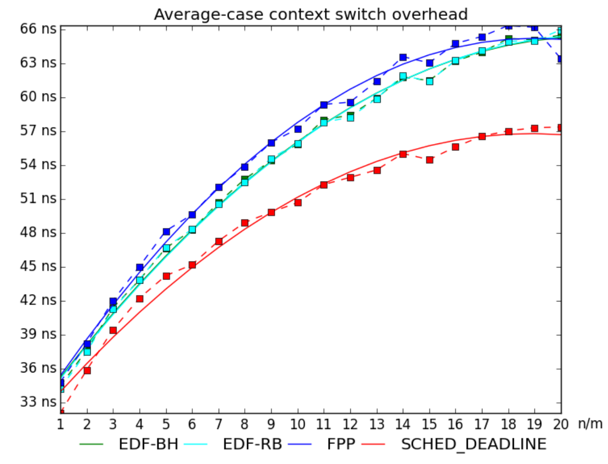


(f)

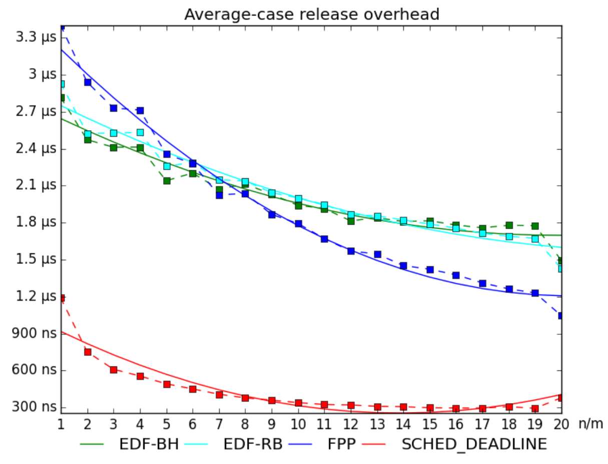
Figure 7.4: Worst-case overheads, on the x86 (left), on the Raspberry Pi 3 Model B (right). Note that the graphs have different scales and that outliers have been filtered before computing the worst-case values.



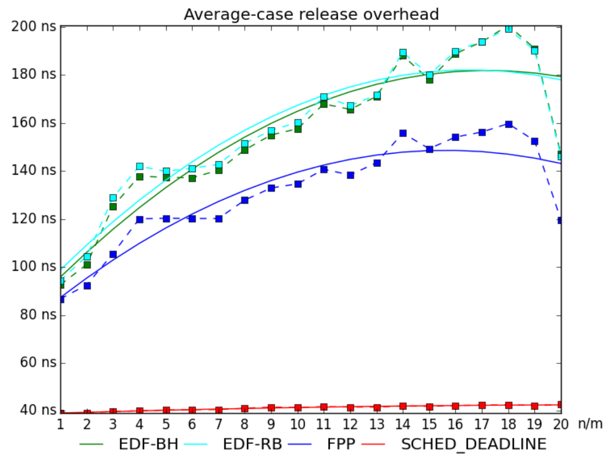
(a)



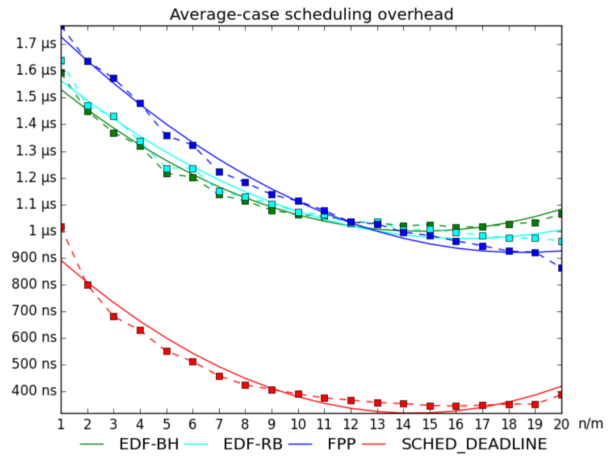
(b)



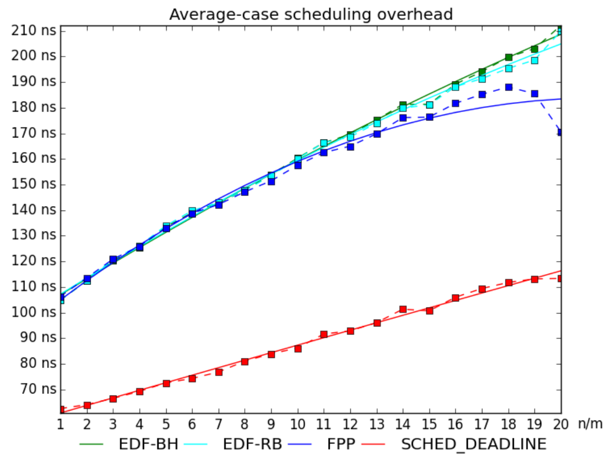
(c)



(d)



(e)



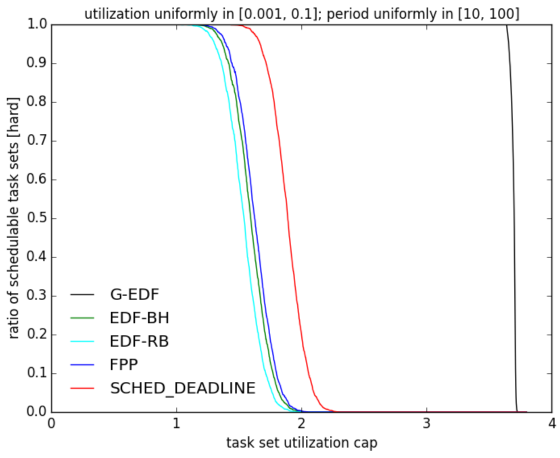
(f)

Figure 7.5: Average-case overheads, on the x86 (left), on the Raspberry Pi 3 Model B (right). Note that the graphs have different scales and that outliers have been filtered before computing the worst-case values.

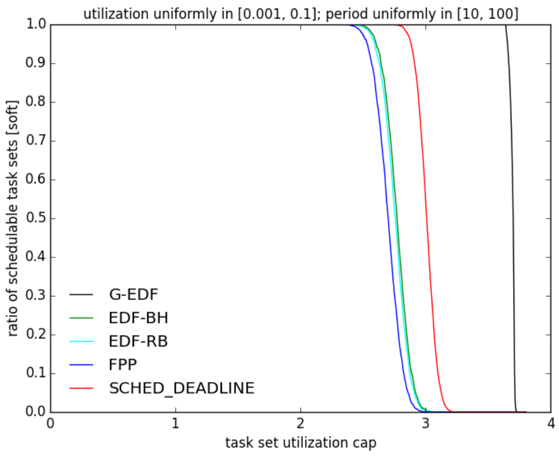
7.3 Schedulability

The thesis generated a total of 36 schedulability graphs for the x86 processor and another 36 graphs for the Raspberry Pi. Due to space constraints, figure 7.6 and 7.7 only present the schedulability results for task sets with uniform moderate periods obtained using the overhead data from the x86. The schedulability graphs for the ARM follow this same trend, except that the difference between the schedulers is so small that no difference can be seen, see appendix B. It is already difficult to see a difference between the schedulers in terms of schedulability with the overheads obtained on the x86.The graphs for uniform moderate periods were selected since most tasks at Svenskt Stål AB (SSAB) have periods in the range $[10, 100]$ ms. All schedulability graphs can be found in appendix A and B for the x86 and the Raspberry Pi respectively.

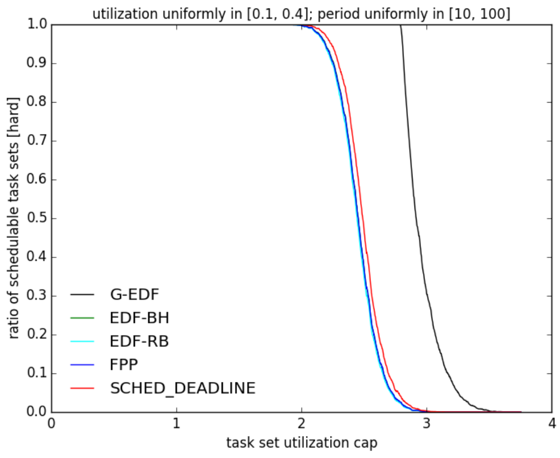
As expected, the implemented schedulers perform slightly worse than SCHED_DEADLINE since the runtime overheads cause by the implemented schedulers were higher than those of SCHED_DEADLINE, this is most obvious in figure 7.6 (a) and (b). Worth noting is that the overheads caused by the schedulers primarily reduce the schedulability of task sets with many low-utilization tasks, where the scheduler is invoked very frequently. For task sets with few high-utilization tasks, the runtime overheads caused by the scheduler barely affect the schedulability of the task set. It is also worth nothing that there is barely any difference between the schedulers in terms of schedulability, not even between the implemented schedulers and SCHED_DEADLINE, for task sets where the utilization of the tasks follow a bimodal distribution (i.e. a mix of light and heavy tasks), see figure 7.7. That the FPP scheduler performs worst for SRT scheduling of uni-light utilization and uni-moderate periods, figure 7.6 (b), can be explained by the slightly higher average-case context-switch overhead of the FPP scheduler, see figure 7.1 (b). Since figure 7.6 (b) shows the schedulability of task sets with very many low-utilization tasks, there may be a lot of context-switches and thus a small increase in context-switch overhead makes a large difference in schedulability.



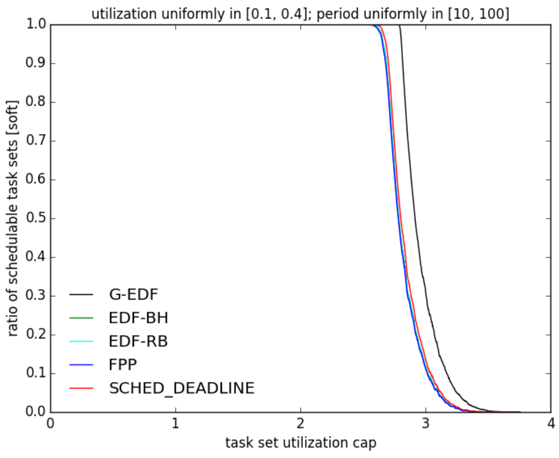
(a)



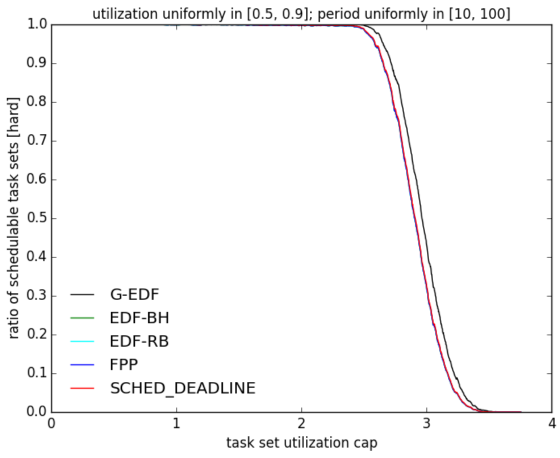
(b)



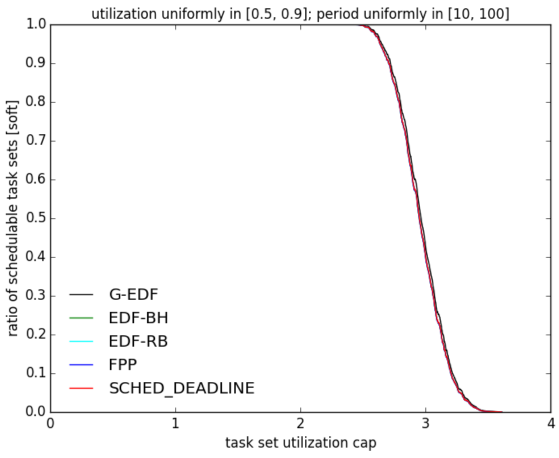
(c)



(d)

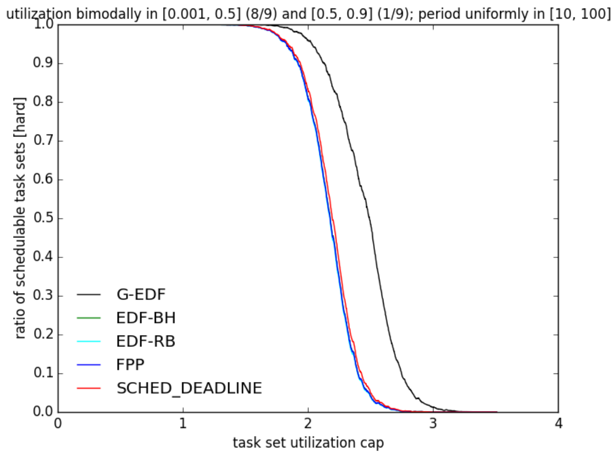


(e)

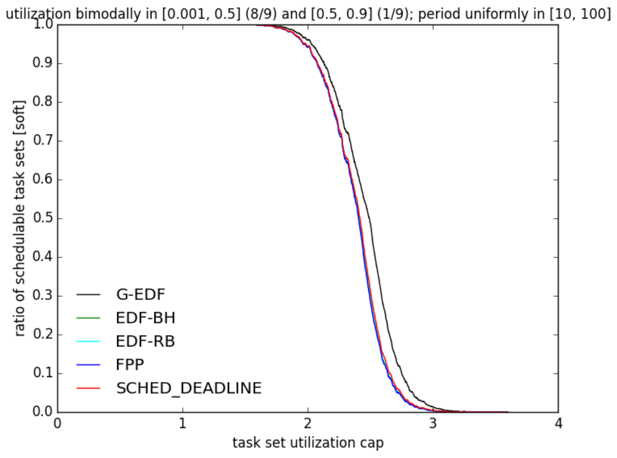


(f)

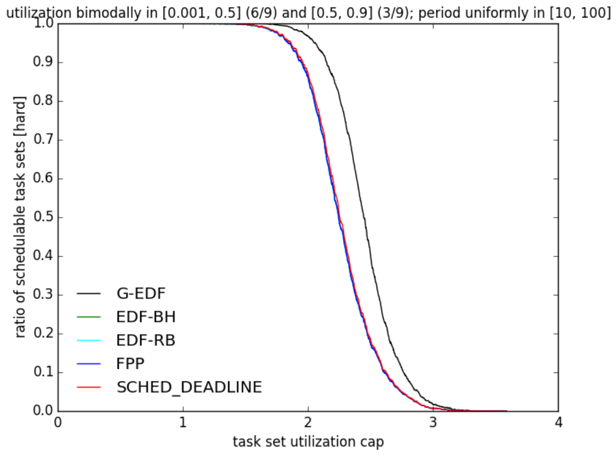
Figure 7.6: Schedulability for task sets with moderate periods and uniformly distributed utilization. Hard real-time to the left, soft real-time to the right. The black line (G-EDF) represent the theoretical bound without overheads.



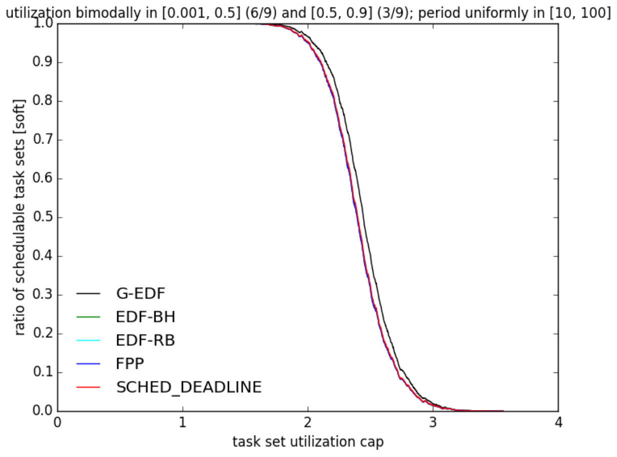
(a)



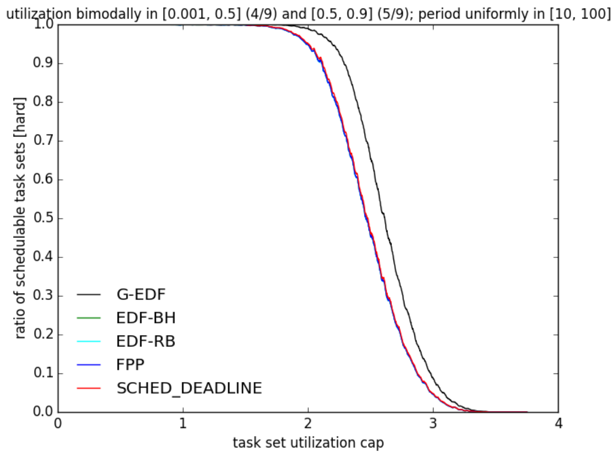
(b)



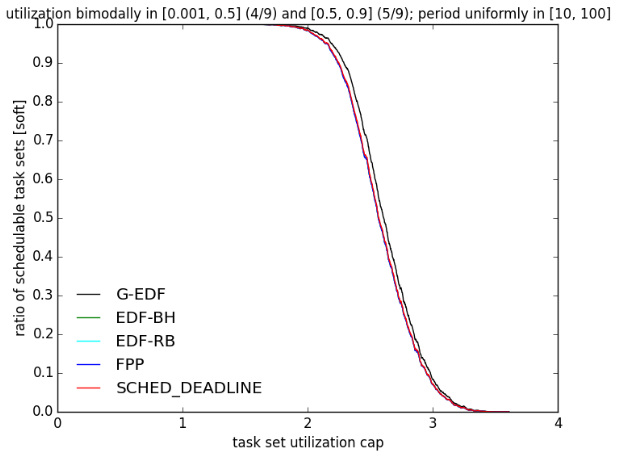
(c)



(d)



(e)



(f)

Figure 7.7: Schedulability for task sets with moderate periods and bimodally distributed utilization. Hard real-time to the left, soft real-time to the right. The black line (G-EDF) represent the theoretical bound without overheads.

8
Discussion

Before concluding the thesis, there are a few things that need to be discussed. First is the chaotic runtime overheads acquired on the x86, the effect of batching simultaneous job releases, and the astonishing overheads acquired on the ARM board. Second is the schedulability of randomly generated task sets when scheduled by the schedulers, which differs significantly between the x86 and the ARM. Finally, we present future work that could be done to improve and evaluate the schedulers, as well as some recommendations regarding x86 versus ARM when using Linux.

8.1 Runtime overheads

The claim by Pathan [15, 16, 17] that FPP should cause fewer context switches than EDF could not be verified, figure 7.2, since all three implemented schedulers caused roughly the same number of context switches. This may be attributed to the design of the implemented EDF-BH and EDF-RBT schedulers. These schedulers were designed with FPP in mind, and thus their design is largely influenced by the description of the FPP scheduler by Pathan. All three implemented schedulers, including the FPP scheduler, are event-driven EDF schedulers using a coarse-grained lock to synchronize access to the global ready queue. These design decisions, event-driven versus quantum-driven, coarse-grained versus fine-grained locking, with or without dedicated interrupt handling, may influence the number of preemptions and migrations. The main difference between the implemented schedulers is only the data structure used as the ready queue, where the EDF-BH and EDF-RBT use a binary heap and a red-black tree respectively, while the FPP scheduler use an array of linked lists together with the FPP ready queue operations. The conclusion that must be drawn is that the ready queue of the FPP scheduler does not by itself reduce the number of context switches, but it does however reduce the magnitude of the runtime overheads caused by the scheduler, figure 7.1.

The comparison against SCHED_DEADLINE is not entirely fair since the implemented schedulers perform all scheduling decisions in the five functions in listing 2. There are another 15 functions that a scheduling class can implement and SCHED_

DEADLINE fully utilize all of these instead of squeezing all logic into a subset of them. When measuring for example the scheduling overhead, we only measure the time taken to execute the `pick_next_task` function. This works for the implemented schedulers where all scheduling decisions are made in the measured functions, but SCHED_DEADLINE performs many decisions in the other function that were not measured, and thus these decisions are not part of the observed runtime overhead.

8.1.1 x86

The raw overheads observed on the Linux x86 system varied immensely, with fluctuations that were too big to be the result of caching. A complete cache miss in all three cache levels incur a main memory access that takes 63 ns, but the fluctuations in observed overheads were a magnitude 1000 timer larger than that. The cause of these massive fluctuations were traced back to interrupts in between the recorded timestamps and it was found that approximately 15% of the recorded overhead samples had been disturbed by external interrupts and had to be filtered. The number and magnitude of interrupts disturbing the schedulers cause implications not only for HRT scheduling but also for SRT scheduling, see section 8.2.

8.1.2 Batching releases

Batching simultaneous job releases with the release queue effectively reduce the average-case release overhead experienced by a job, while the worst-case release overhead remain unchanged, figure 7.3. Since real-time schedulability analysis is done without taking the observed number of overheads into account, batching does not increase the schedulability of HRT task sets, and may even decrease the schedulability slightly. For SRT scheduling however, it effectively reduces the average-case release overhead while keeping the worst-case overheads bounded, and thus improve the schedulability of SRT task sets.

8.1.3 ARM

In contrast to the overheads obtained on the x86, there were very few outliers in the overhead samples obtained on the Raspberry Pi. These outliers were also very large in their magnitude and may be caused by an insufficient power supply since the Raspberry Pi was fed with exactly the minimum power required. When CPU utilization goes up and the Raspberry Pi becomes warm, the minimum power required to boot the Raspberry Pi may become insufficient and cause the board to hang. The Raspberry Pi not only caused very few outliers compared to the x86, but after filtering out the outliers, the overheads observed on the Raspberry Pi were

$8 - 75$ times smaller than those observed on the x86, see section 7.2.3 and figure 7.4 and 7.5.

8.2 Schedulability

In hindsight, it would have been better to focus on the Raspberry Pi and evaluate the schedulers there. The large number of outliers that had to be filtered on the x86 processor make the schedulability results unreliable and they do not reflect the true schedulability of a four core x86 processor running Linux since we do not account for these external interrupts in the schedulability tests. Unfortunately, we do not know the actual worst-case overheads since we do not know how if the number of interrupts that may disturb the scheduler can be bounded. It may even be the case that these interrupts can starve the scheduler, in which case both the HRT *and* the SRT schedulability is zero on the x86. We believe that this problem may be alleviated to some degree on multicore processors by using dedicated interrupt handling, see section 2.1.3. With dedicated interrupt handling, the schedulability results in this thesis may be applicable. For a system with five processor cores, where one of the cores is dedicated to interrupt handling, the remaining four cores may exhibit a schedulability that match the schedulability graphs in figure 7.6 and 7.7.

The Raspberry Pi is a completely different story, the runtime overheads caused by the schedulers are very small and thus affect the schedulability of the analyzed task sets to a much lower extent. In order for the runtime overheads of the schedulers to significantly reduce the schedulability of a task set on the Raspberry Pi, the task set must contain a very large number of tasks and the tasks must have very short periods, compare figure B.1 (many tasks and short periods) to figure B.3 (few tasks and short periods) and figure B.13 (many tasks and long periods).

At first it may seem peculiar that the implemented schedulers perform comparably to SCHED_DEADLINE and not better since they cause fewer overheads, figure 7.2. However, when inflating the parameters of a task set to verify the schedulability of the task set, the actual number of overheads is not taken into account, only the magnitude of the overheads. Instead, a worst-case number of overheads is calculated and multiplied by the magnitude of the overheads. Given that the magnitude of the runtime overheads were higher for the implemented schedulers compared to SCHED_DEADLINE, figure 7.1, they are expected to perform worse in the schedulability tests. It is interesting however that the difference between the schedulers is primarily seen for task sets with many tasks, figure 7.6 (a) and (b). Since each task in these task sets have a low utilization, many tasks have to be generated in order to reach the fixed utilization cap, and the scheduler is invoked very frequently for task sets with many tasks. In figure 7.6 (c), (d), (e) and (f), the utilization of each task is higher, thus fewer tasks are needed to reach the utilization threshold, and for task sets with few tasks the overhead of the scheduler barely matters. It is also interesting that no difference between the schedulers can be seen for task sets where the utilization of the tasks follow a bimodal distribution (i.e. a mix of light and heavy tasks), figure 7.7.

8.3 Future work

The implemented FPP scheduler can be further improved in two ways. First, the FPP scheduler is limited to 512 unique relative deadlines. This limitation can be more or less removed by switching from a sequential bitmap to hierarchical bitmaps, which would allow for a much larger data structure without increasing the runtime overhead of the scheduler [14].

The second step is to take the current implementation of SCHED_DEADLINE, replace the red-black tree with the FPP ready queue, and evaluate it against the original SCHED_DEADLINE implementation. The work in this thesis suggests that FPP is an effective strategy for reducing the magnitude of the runtime overheads caused by EDF schedulers, even for rather mature and optimized implementations of EDF such as the SCHED_DEADLINE scheduler. Thus the modified SCHED_DEADLINE using the FPP ready queue is expected to perform slightly better than the original SCHED_DEADLINE implementation. Since the data structure used by FPP is already used by the SCHED_RT scheduler, this modification should be relatively small since it only involves the addition of the FPP ready queue operations and the removal of red-black tree related code. The main bulk of the work is to evaluate the modified SCHED_DEADLINE against the original implementation to verify that it actually performs better.

Finally, future research on evaluating real-time schedulers should be done on cheap ARM boards such as the Arduino or the Raspberry Pi. These ARM boards provide relatively high performance while still retaining the predictability that is crucial to real-time scheduling. Given the low price of these boards, they will quickly become the new standard for real-time systems. This thesis used a x86 because SSAB have traditionally used low-cost x86 consumer processors in their SRT systems. With the release of the Arduino and Raspberry Pi boards, there are now better alternatives available and this thesis used a Raspberry Pi since SSAB are evaluating them for use in future real-time systems. A recommendation to the company would be to start deploying Raspberry Pis as their new real-time systems, and to enable dedicated interrupt handling on their current real-time systems. Without dedicated interrupt handling, the x86 processors may cause a possible unbounded number of interrupts in which case they cannot be safely used for real-time systems since they cannot guarantee a schedulability larger than zero.

9 Conclusion

The purpose of this thesis was to investigate if FPP was an effective strategy for reducing the overhead of EDF schedulers when implemented on a real platform. To investigate, a FPP scheduler and a “standard” EDF scheduler based on a binary heap, EDF-BH, were implemented. The FPP scheduler was benchmarked against the EDF-BH scheduler, but also against the rather mature and optimized EDF scheduler in the Linux kernel, SCHED_DEADLINE, which uses an red-black tree as ready queue. The hypothesis was that the FPP scheduler would perform better than the binary-heap based EDF-BH scheduler, but that it would be inferior to SCHED_DEADLINE since this scheduler has been developed and improved for several years by many very competent developers.

From the first results it could be concluded that the implemented FPP and EDF-BH schedulers were incomparable to SCHED_DEADLINE. Thus, a second EDF scheduler equal to the EDF-BH scheduler, but with the red-black tree from SCHED_DEADLINE as the ready queue, was developed. The results showed that the FPP scheduler outperform both the heap based and red-black tree based EDF schedulers implemented in this thesis. It did not cause fewer context switches compared to the two EDF schedulers, but the magnitude of the runtime overheads caused by the scheduler were significantly smaller. The results also showed that the performance of SCHED_DEADLINE is unrelated to the data structure used as the ready queue, and that the FPP ready queue would most likely be an improvement over the red-black tree used by SCHED_DEADLINE.

Evaluating the schedulers on a x86 Linux system gave a very surprising result. The potentially unbounded number of interrupts cause implications for both SRT and HRT scheduling, and Linux on x86 processors should only be used for real-time scheduling with dedicated interrupt handling. Since cheap ARM boards such as the Arduino and the Raspberry Pi are becoming increasingly common and will be the future of real-time systems, future work should use and evaluate real-time schedulers on these ARM boards instead. They provide good performance while still retaining the predictability that is crucial to real-time scheduling.

Based on the runtime overheads observed both on the Intel i3-4370 (x86) and on the Raspberry Pi 3 model B (ARM), the conclusion is that FPP does not reduce the number of context switches compared to similar EDF schedulers. It is however an effective strategy for reducing the magnitude of the runtime overheads caused by EDF schedulers, probably even for rather mature and optimized implementations of EDF such as the SCHED_DEADLINE scheduler.

Bibliography

[1] L. Abeni, J. Lelli, T. Cucinotta, D. Faggioli, and C. Scordino. *Deadline Task Scheduling*, 2017. `https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt` [Accessed: 2017-11-06].

[2] L. Abeni, G. Lipari, and J. Lelli. Constant bandwidth server revisited. *SIGBED Rev.*, 11(4):19–24, January 2015.

[3] ARM. Arm cortex-a53 mpcore processor - technical reference manual, 2018. `https://developer.arm.com/docs/ddi0500/latest/` [Accessed: 2018-05-20].

[4] ARM. Technologies | big.little, 2018. `https://developer.arm.com/technologies/big-little` [Accessed: 2018-06-04].

[5] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In *Proc. 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, Brussels, Belgium, July 2010.

[6] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 47–56, Aug 2007.

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

[8] B. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 141–152, April 2013.

[9] B. B. Brandenburg and J. H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *In Proceedings of the Third International Workshop on Operating Systems Platformsfor Embedded Real-Time Applications (OSPERT'07*, pages 61–70, 2007.

[10] B. B. Brandenburg and J. H. Anderson. On the implementation of global real-time schedulers. In *2009 30th IEEE Real-Time Systems Symposium*, pages 214–224, Dec 2009.

[11] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual, 2018. `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf` [Accessed: 2018-05-20].

[12] Ulrich Drepper. What every programmer should know about memory. 2007. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`.

[13] J. Lelli. Design and development of deadline based scheduling mechanisms for multiprocessor systems. PhD thesis, University of Pisa, 2010.

[14] Mikolaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos, and Yannis Manolopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *ADBIS*, 2003.

[15] R. M. Pathan. Unifying fixed- and dynamic-priority scheduling based on priority promotion and an improved ready queue management technique. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 209–220, April 2015.

[16] R. M. Pathan. Design of an efficient ready queue for earliest-deadline-first (edf) scheduler. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 293–296, March 2016.

[17] R. M. Pathan. Real-time scheduling on uni- and multiprocessors based on priority promotions. *Leibniz Transactions on Embedded Systems*, 3(1):02–1–02:29, June 2016.

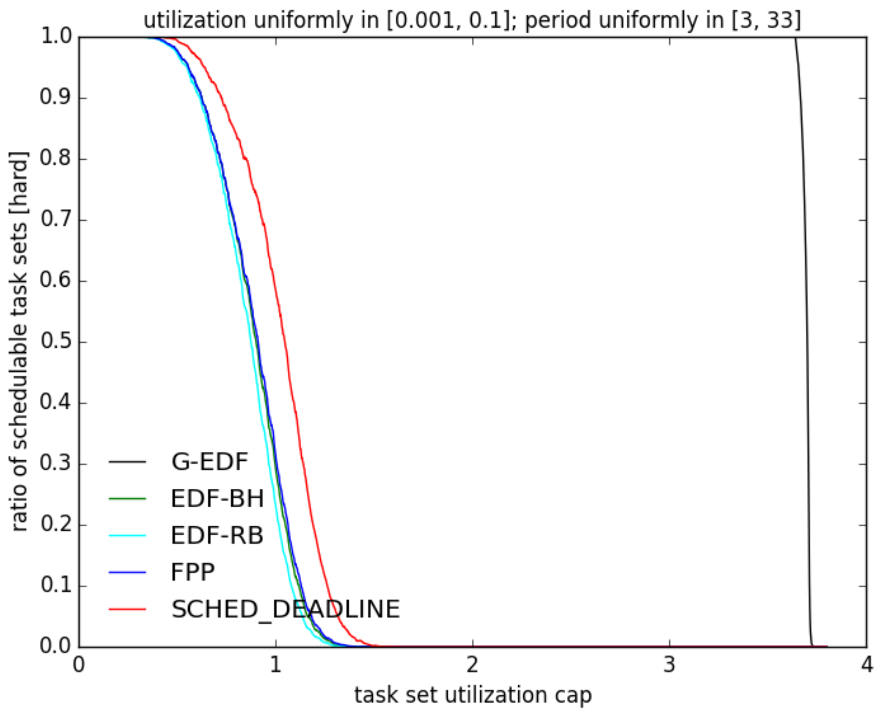
[18] A. Pinkus, S. Winnitzky, and G. Mazur. *Adaptive Function Plotting - Yacas*, 2018. `http://yacas.readthedocs.io/en/latest/book_of_algorithms/basic.html` [Accessed: 2018-04-12].

A

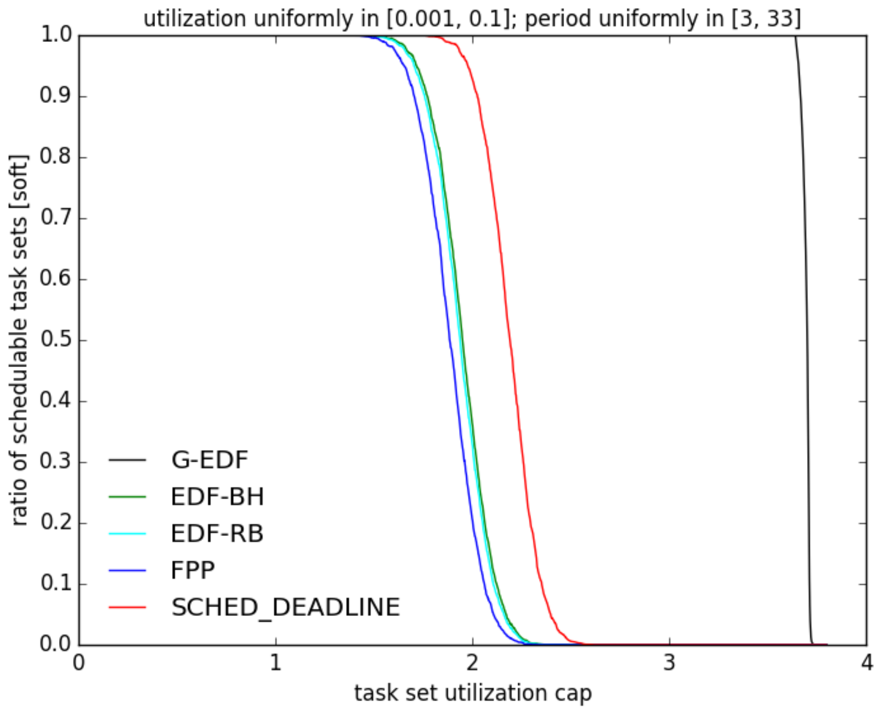
Complete Schedulability Results for x86

This appendix contains the complete schedulability results for the x86. The following 18 figures depict the schedulability results obtained for each combination of the period and utilization distributions listed in section 4.2.

- Figure A.1 shows schedulability for uniform light utilizations and short periods.
- Figure A.2 shows schedulability for uniform medium utilizations and short periods.
- Figure A.3 shows schedulability for uniform heavy utilizations and short periods.
- Figure A.4 shows schedulability for bimodal light utilizations and short periods.
- Figure A.5 shows schedulability for bimodal medium utilizations and short periods.
- Figure A.6 shows schedulability for bimodal heavy utilizations and short periods.
- Figure A.7 shows schedulability for uniform light utilizations and moderate periods.
- Figure A.8 shows schedulability for uniform medium utilizations and moderate periods.
- Figure A.9 shows schedulability for uniform heavy utilizations and moderate periods.
- Figure A.10 shows schedulability for bimodal light utilizations and moderate periods.
- Figure A.11 shows schedulability for bimodal medium utilizations and moderate periods.
- Figure A.12 shows schedulability for bimodal heavy utilizations and moderate periods.
- Figure A.13 shows schedulability for uniform light utilizations and long periods.
- Figure A.14 shows schedulability for uniform medium utilizations and long periods.
- Figure A.15 shows schedulability for uniform heavy utilizations and long periods.
- Figure A.16 shows schedulability for bimodal light utilizations and long periods.
- Figure A.17 shows schedulability for bimodal medium utilizations and long periods.
- Figure A.18 shows schedulability for bimodal heavy utilizations and long periods.

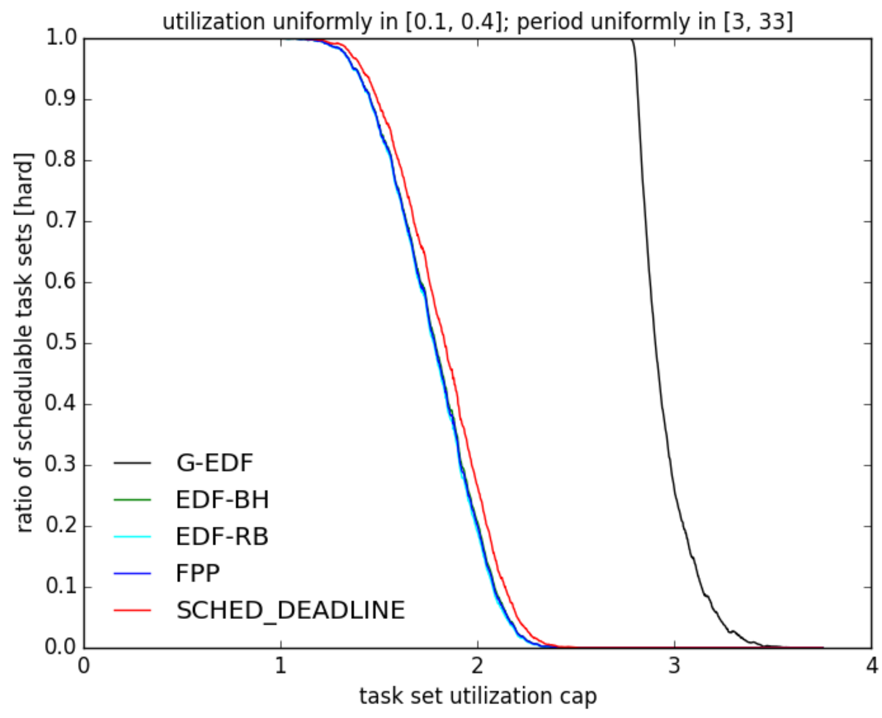


(a) Hard real-time

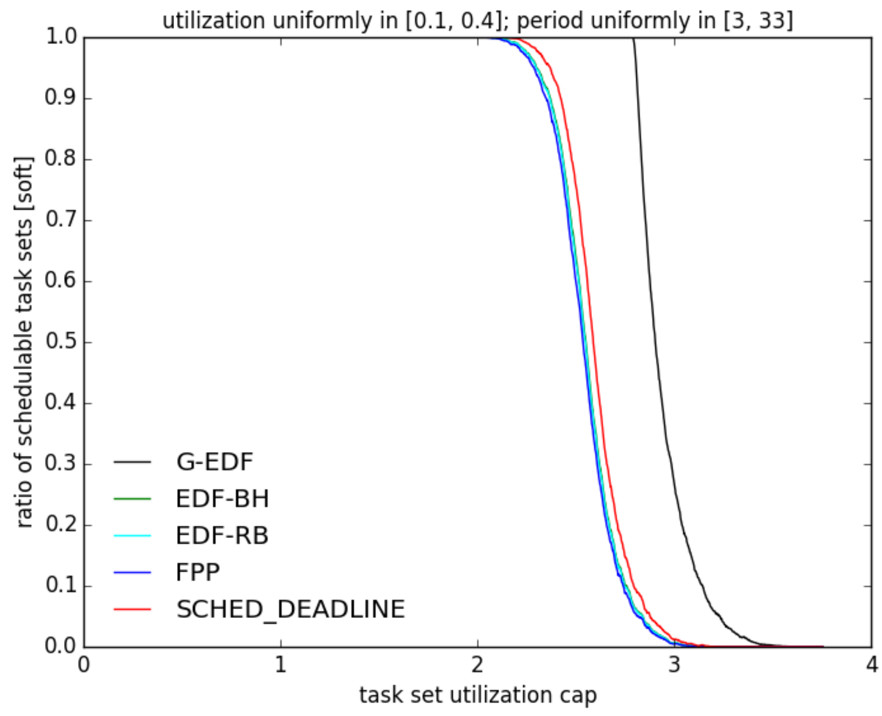


(b) Soft real-time

Figure A.1: Schedulability for task sets with uniformly distributed short periods and uniformly distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

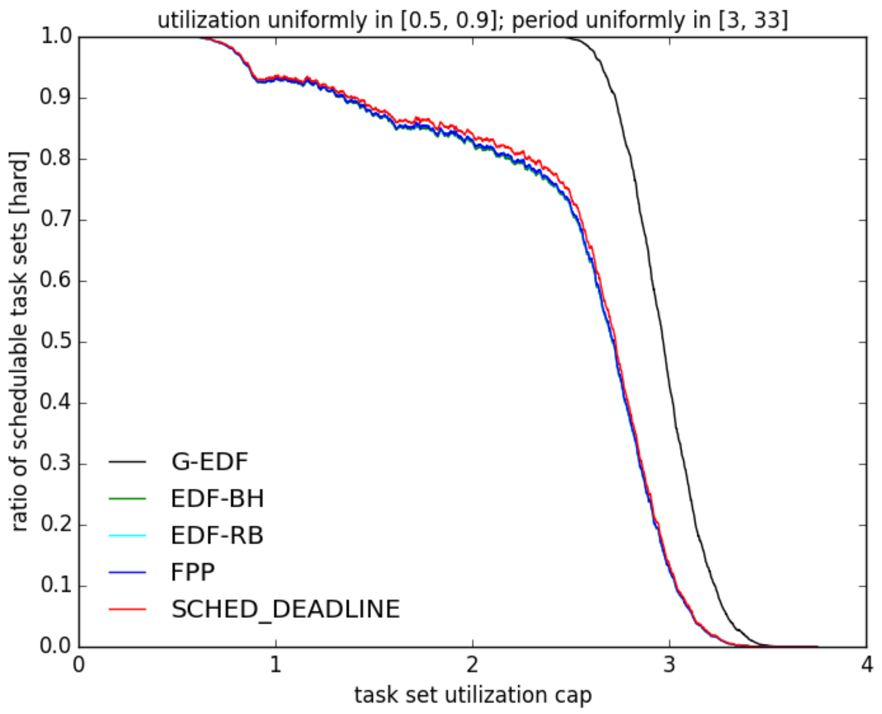


(a) Hard real-time

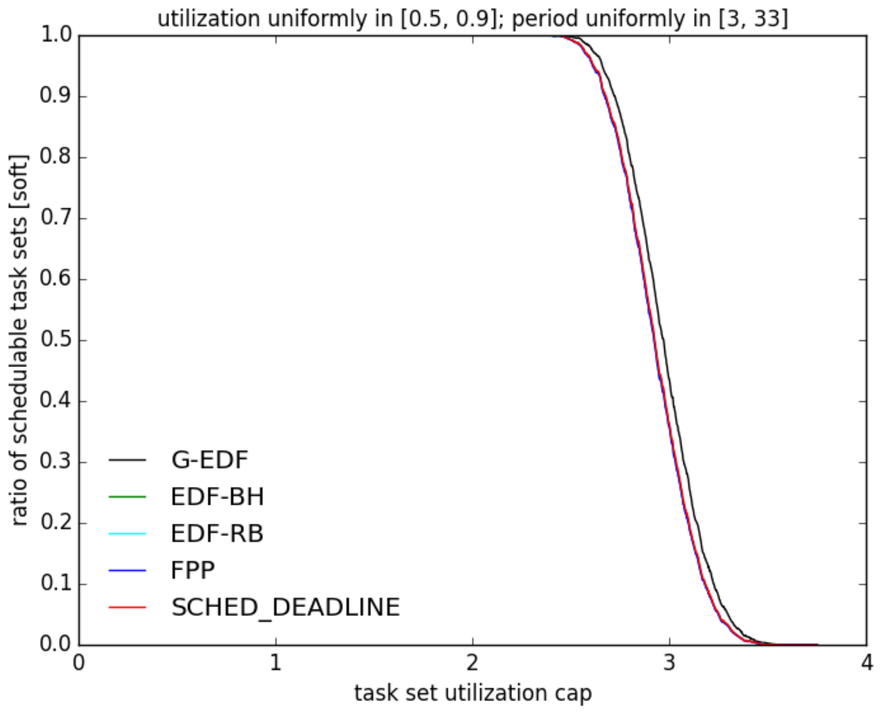


(b) Soft real-time

Figure A.2: Schedulability for task sets with uniformly distributed short periods and uniformly distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

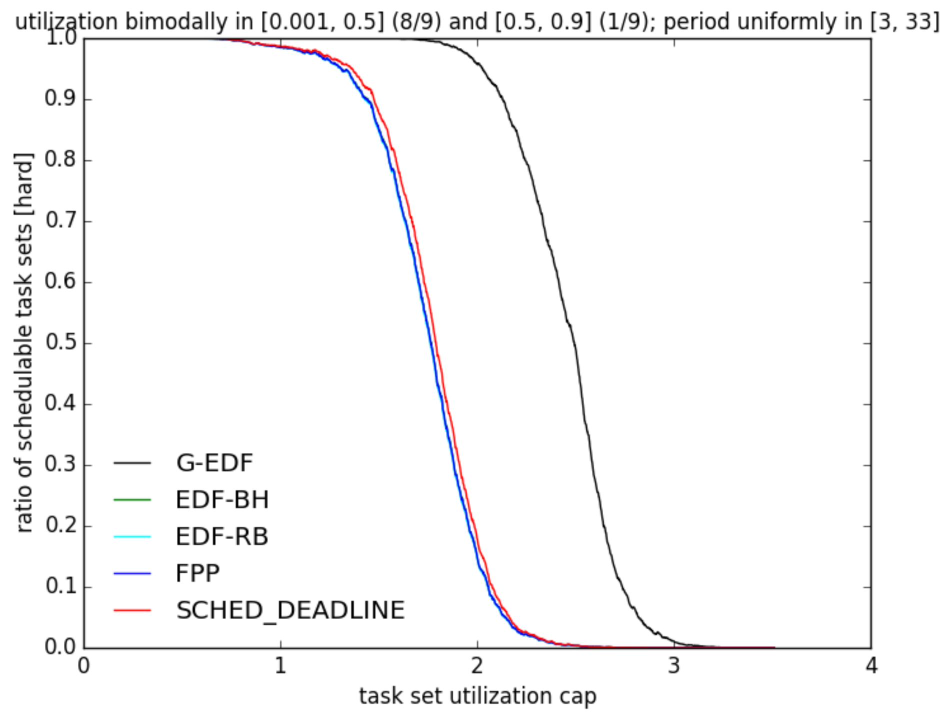


(a) Hard real-time

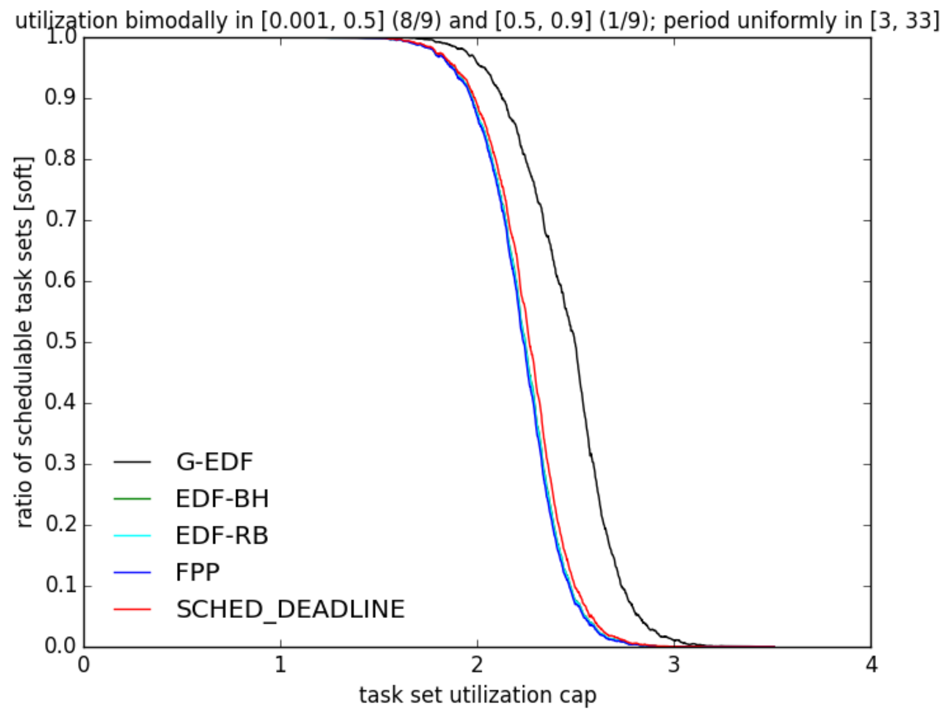


(b) Soft real-time

Figure A.3: Schedulability for task sets with uniformly distributed short periods and uniformly distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

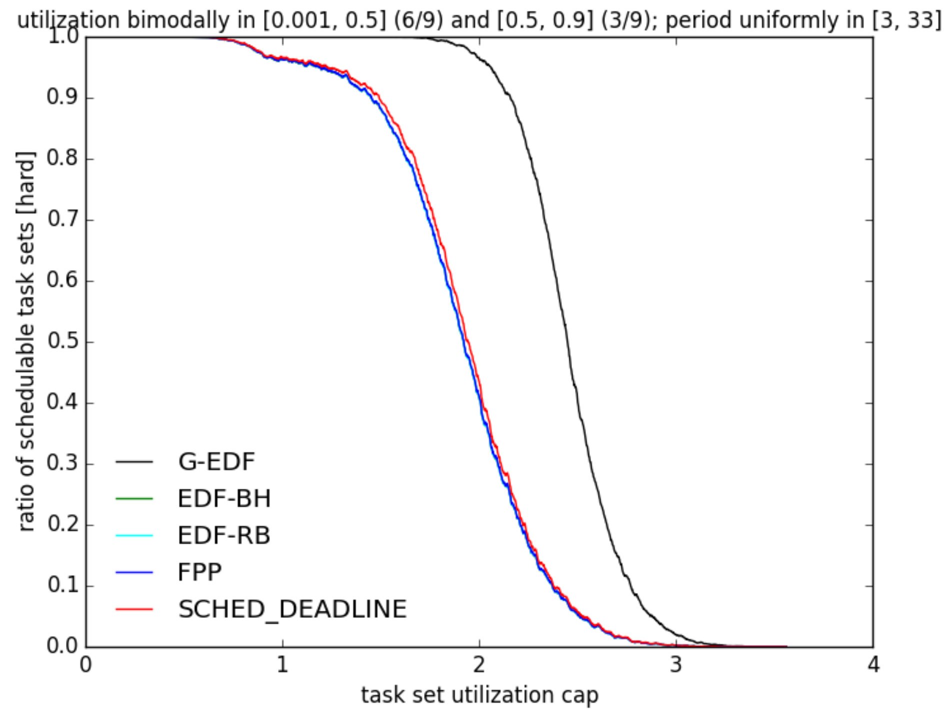


(a) Hard real-time

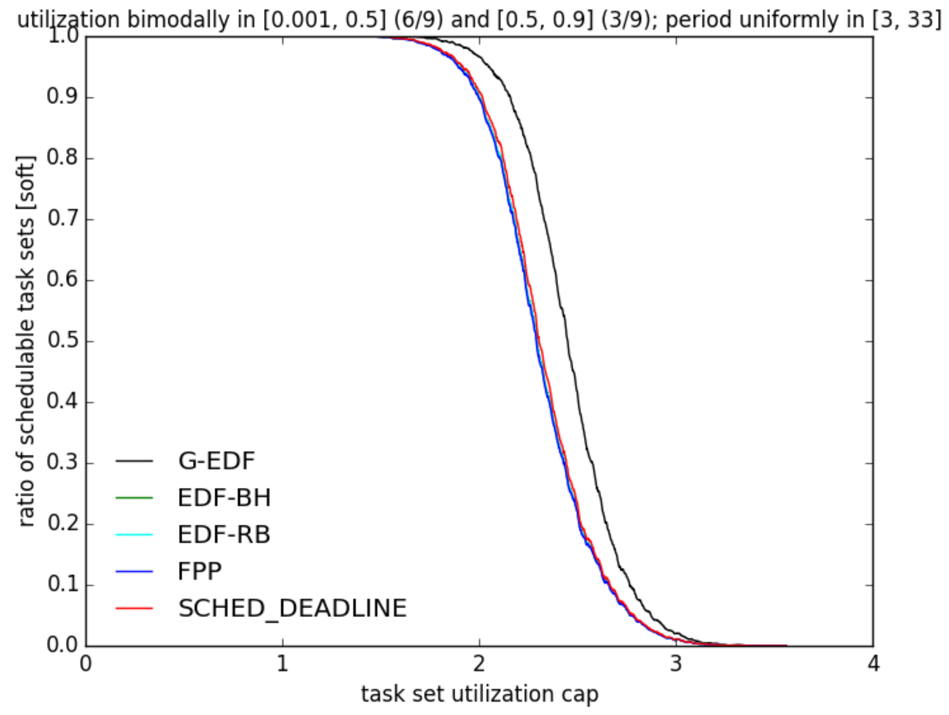


(b) Soft real-time

Figure A.4: Schedulability for task sets with uniformly distributed short periods and bimodally distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

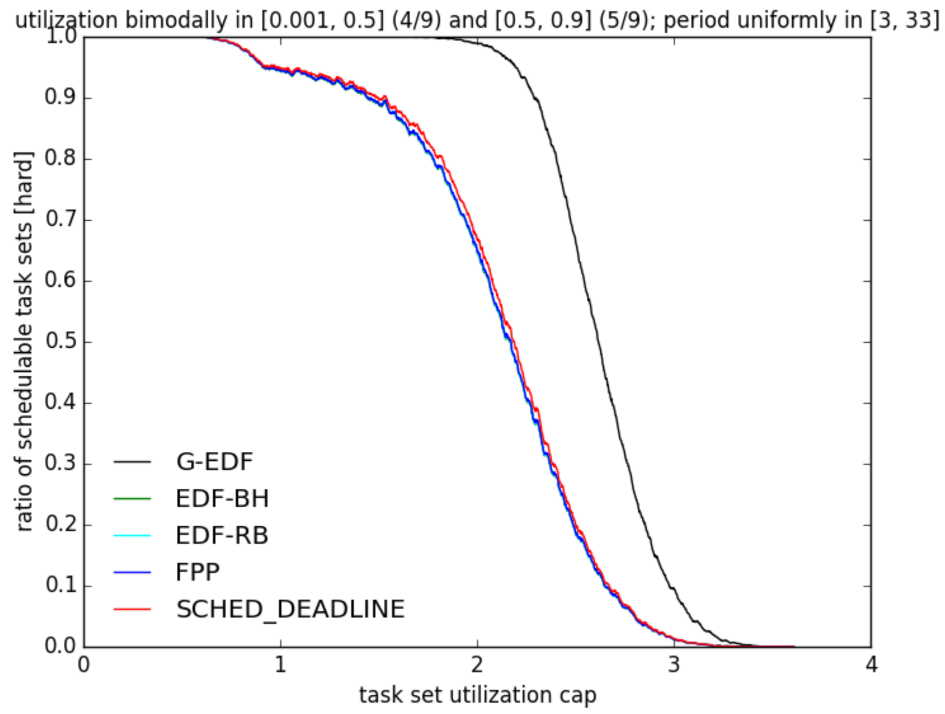


(a) Hard real-time

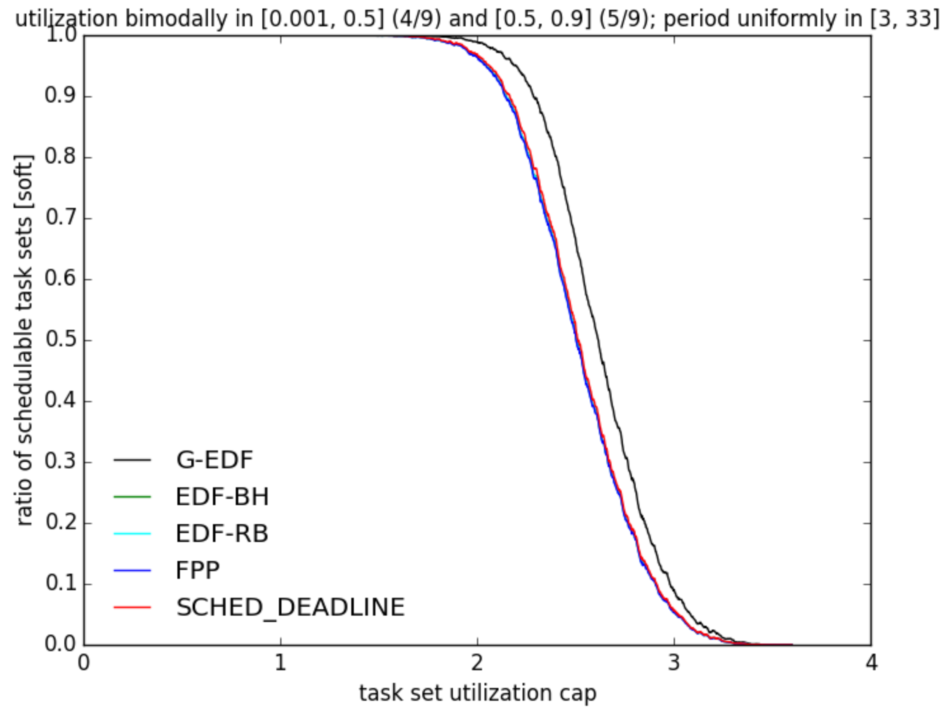


(b) Soft real-time

Figure A.5: Schedulability for task sets with uniformly distributed short periods and bimodally distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

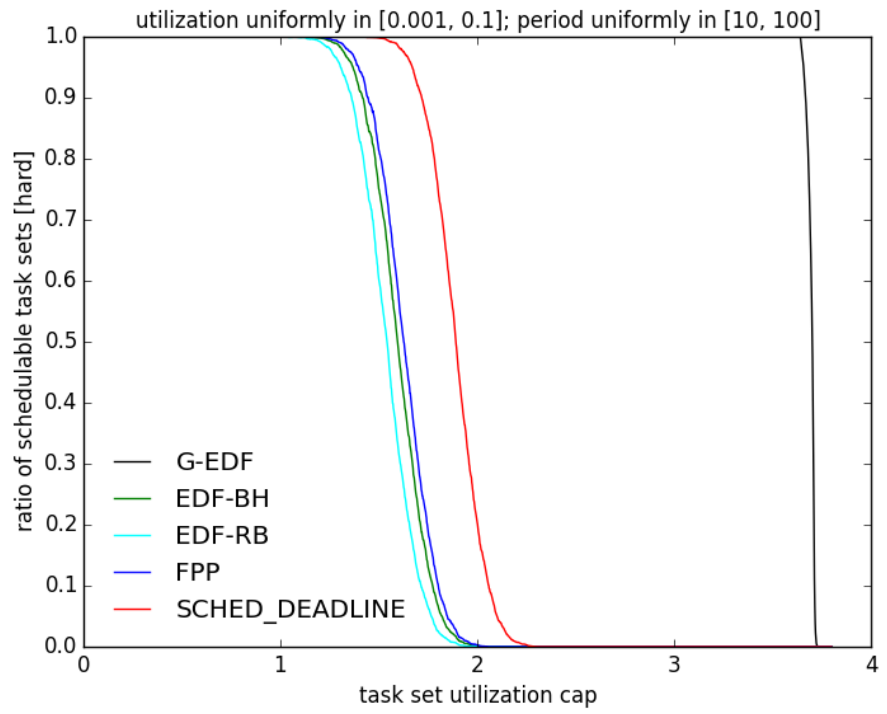


(a) Hard real-time

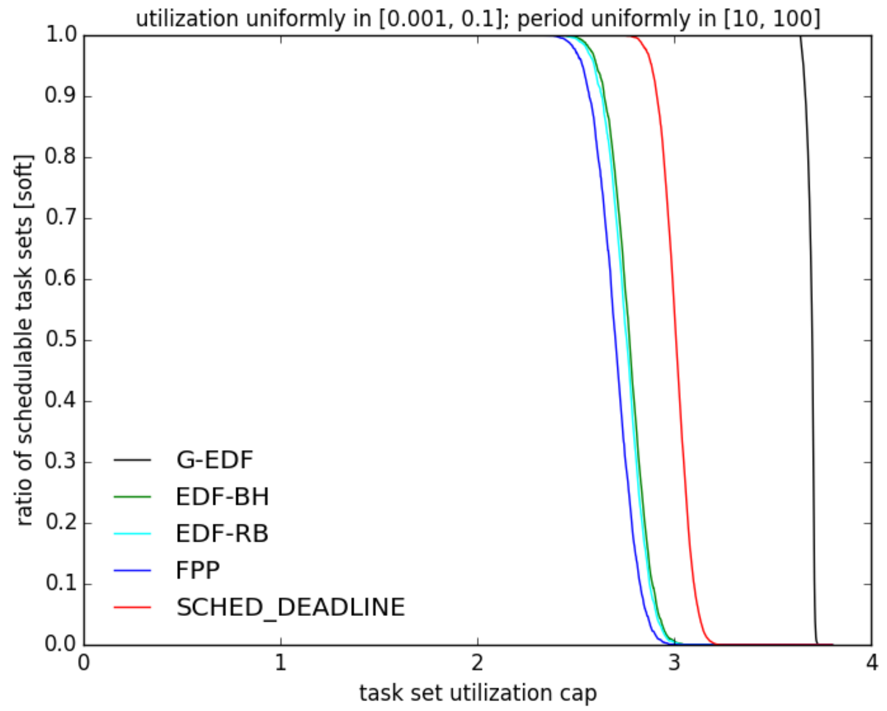


(b) Soft real-time

Figure A.6: Schedulability for task sets with uniformly distributed short periods and bimodally distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

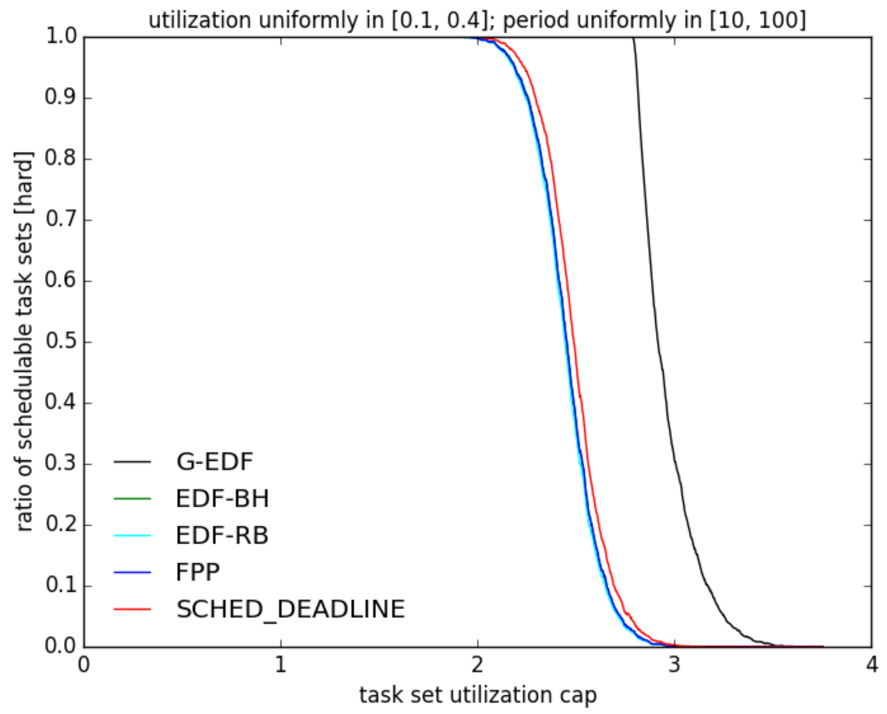


(a) Hard real-time

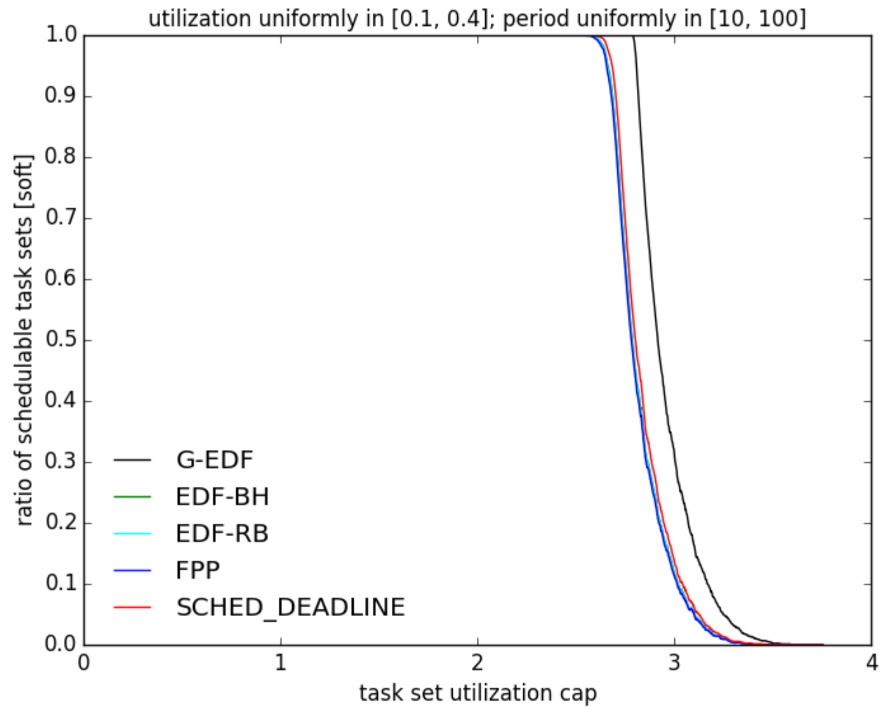


(b) Soft real-time

Figure A.7: Schedulability for task sets with uniformly distributed moderate periods and uniformly distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

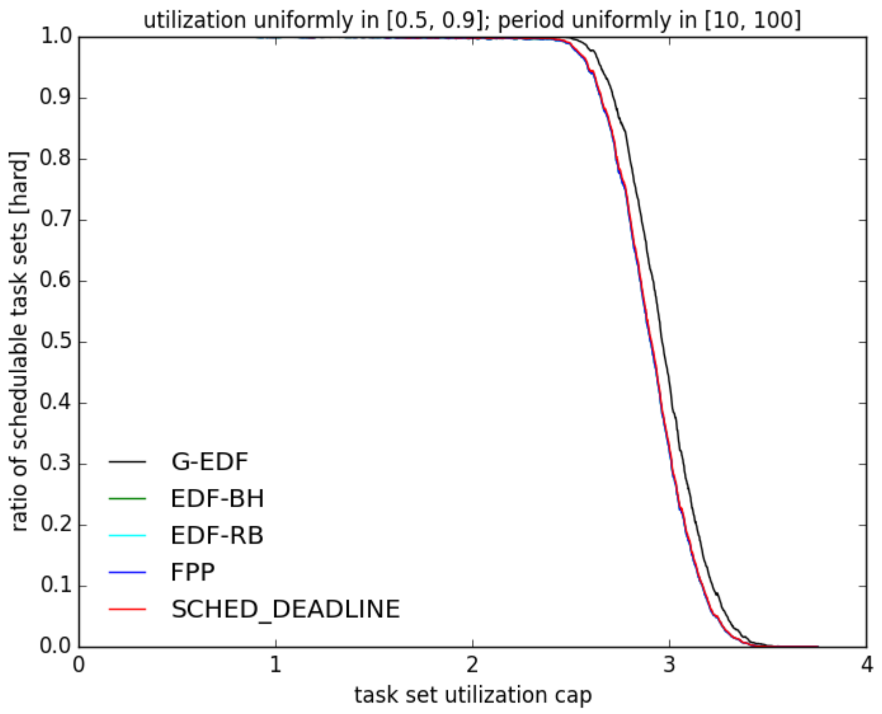


(a) Hard real-time

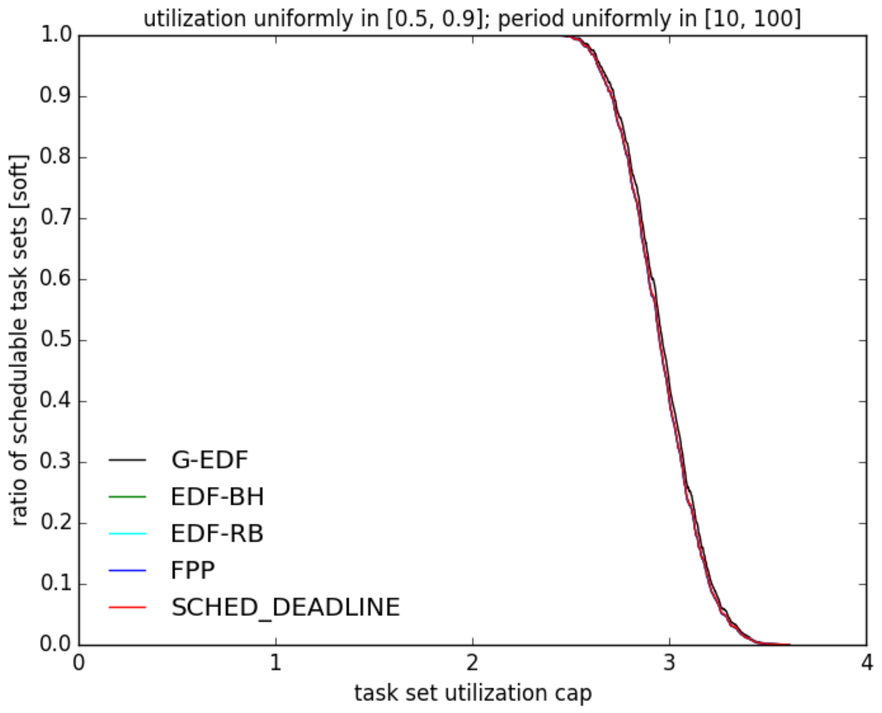


(b) Soft real-time

Figure A.8: Schedulability for task sets with uniformly distributed moderate periods and uniformly distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

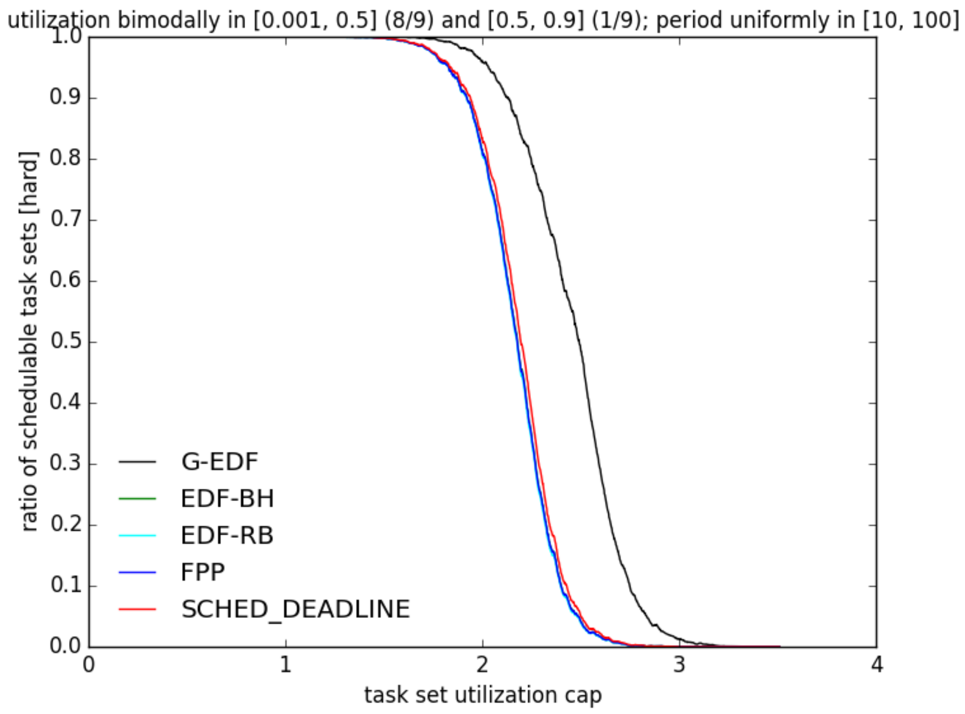


(a) Hard real-time

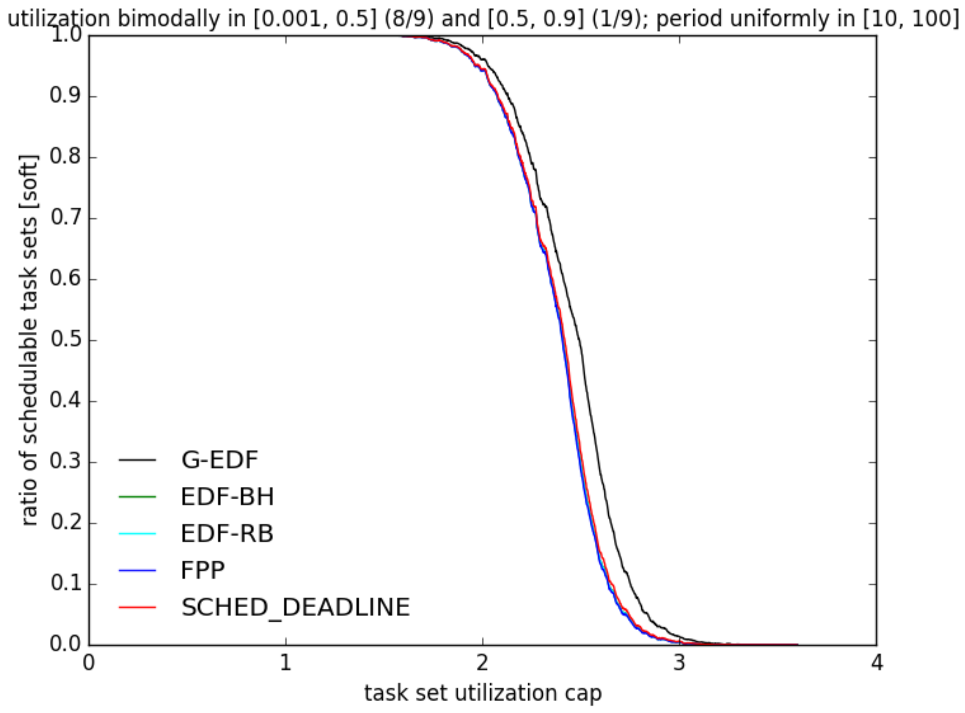


(b) Soft real-time

Figure A.9: Schedulability for task sets with uniformly distributed moderate periods and uniformly distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

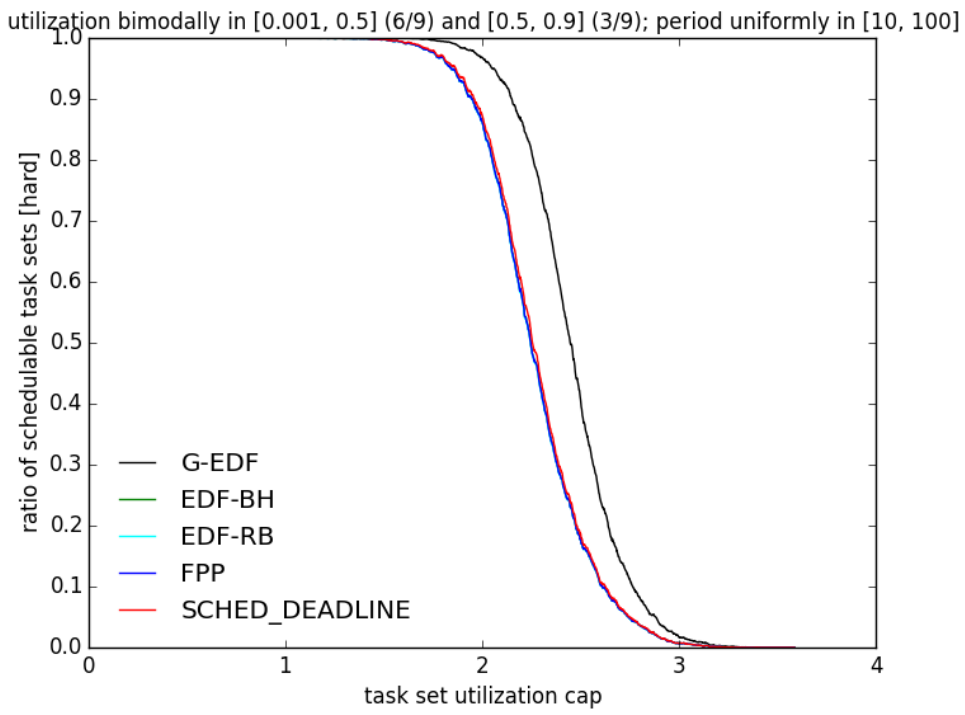


(a) Hard real-time

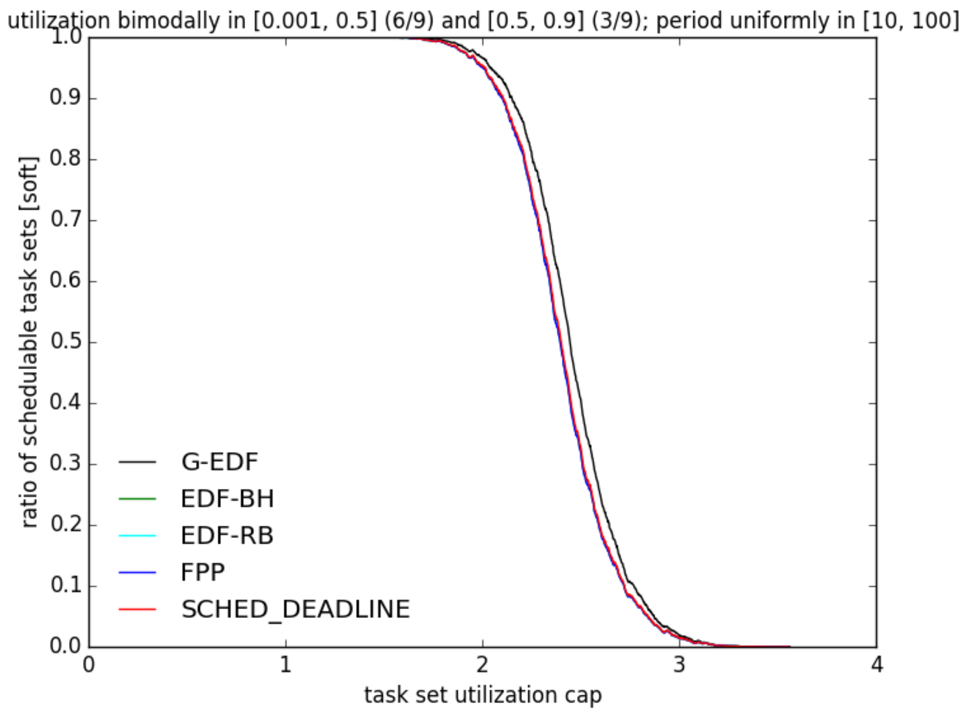


(b) Soft real-time

Figure A.10: Schedulability for task sets with uniformly distributed moderate periods and bimodally distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

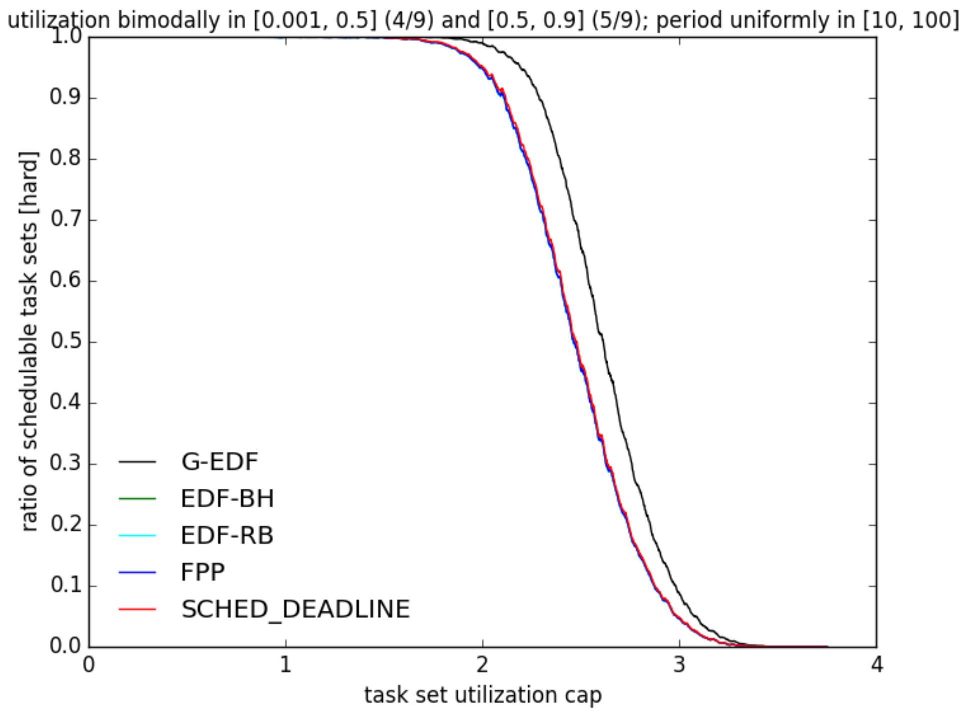


(a) Hard real-time

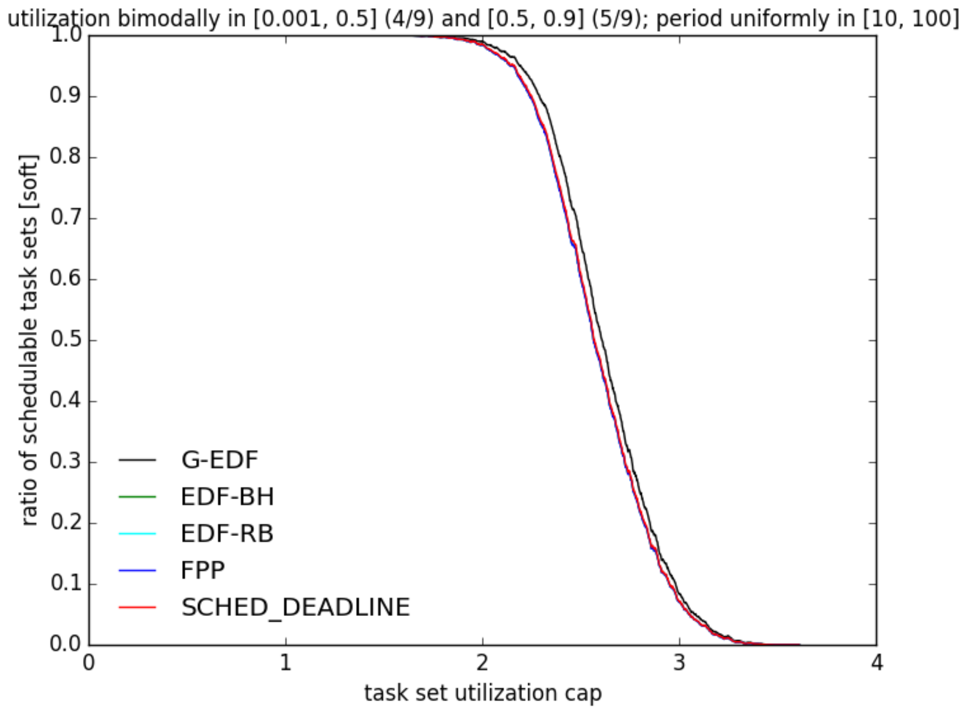


(b) Soft real-time

Figure A.11: Schedulability for task sets with uniformly distributed moderate periods and bimodally distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

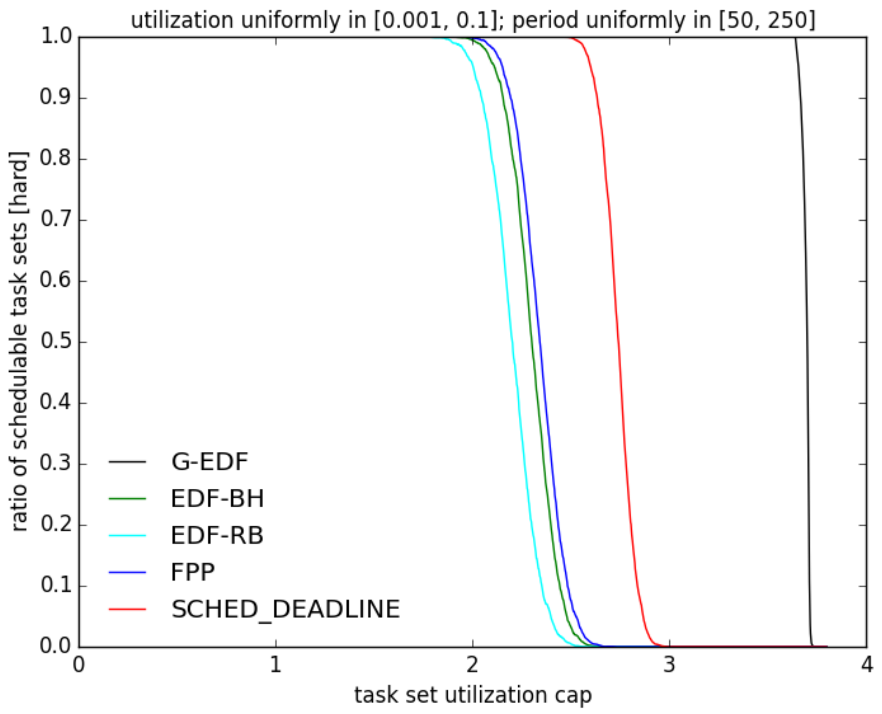


(a) Hard real-time

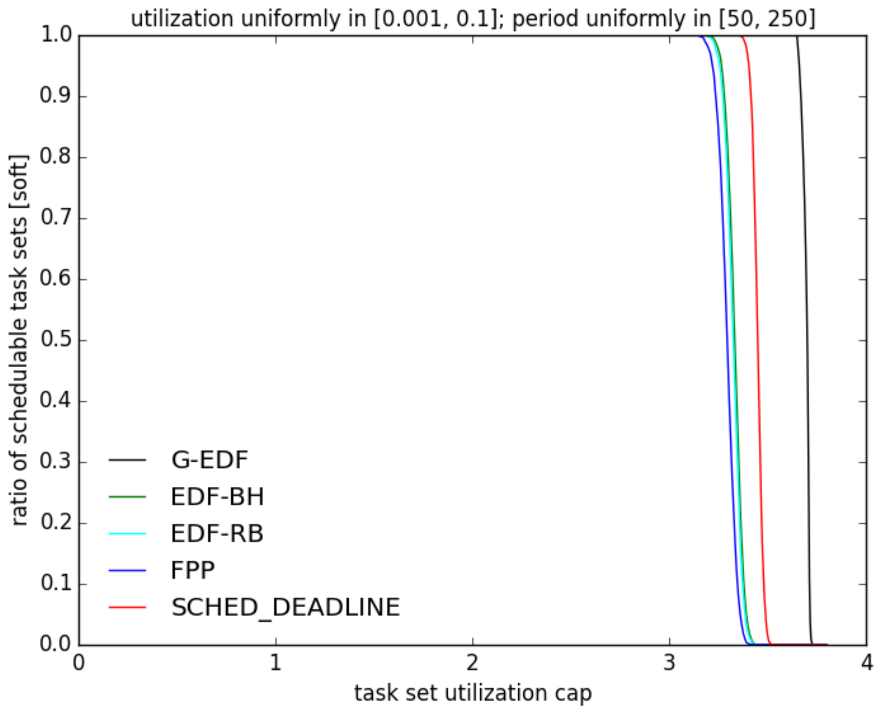


(b) Soft real-time

Figure A.12: Schedulability for task sets with uniformly distributed moderate periods and bimodally distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

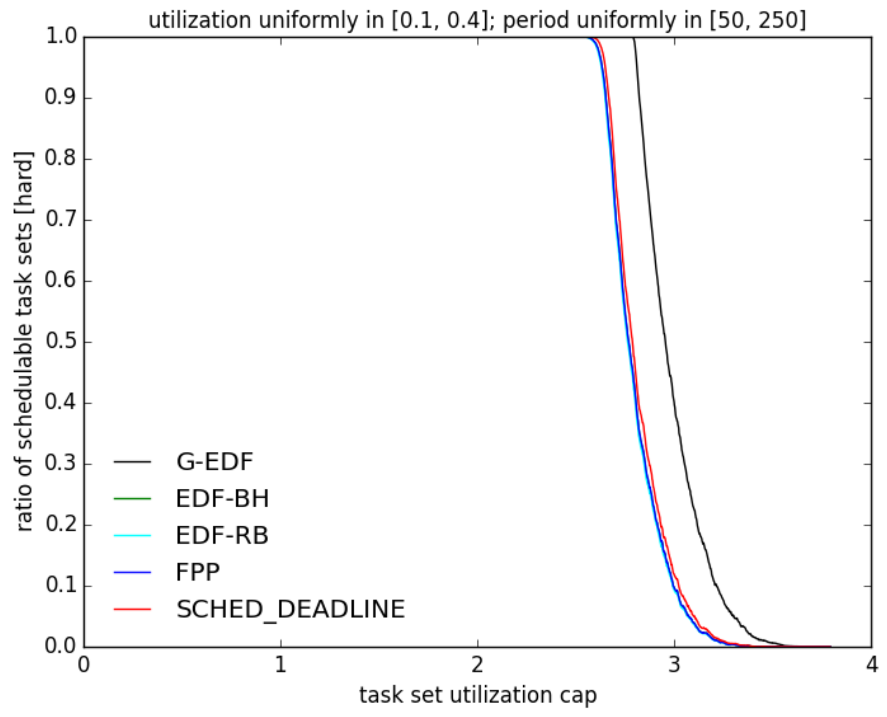


(a) Hard real-time

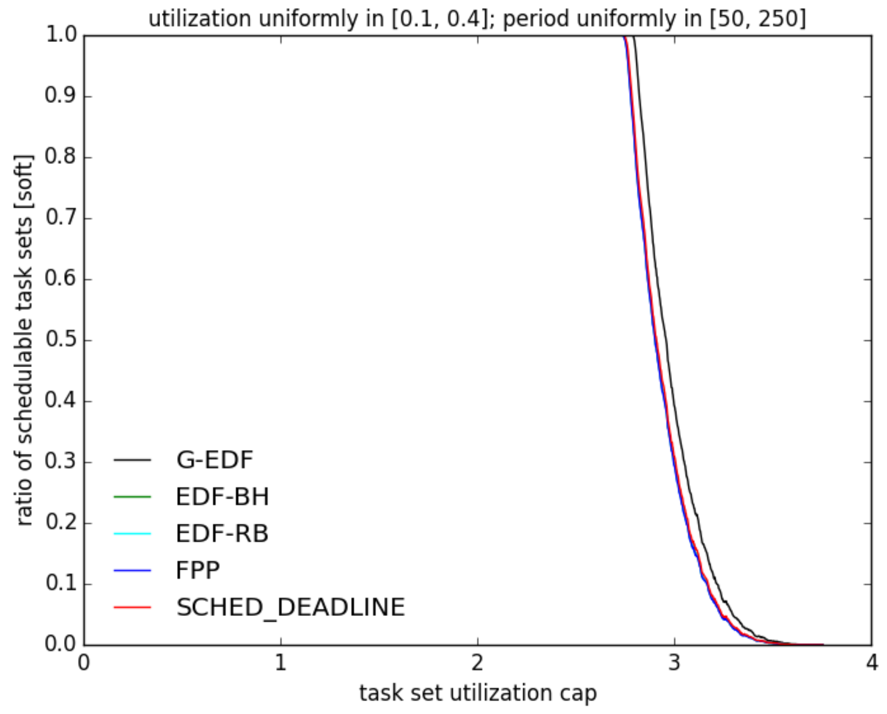


(b) Soft real-time

Figure A.13: Schedulability for task sets with uniformly distributed long periods and uniformly distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

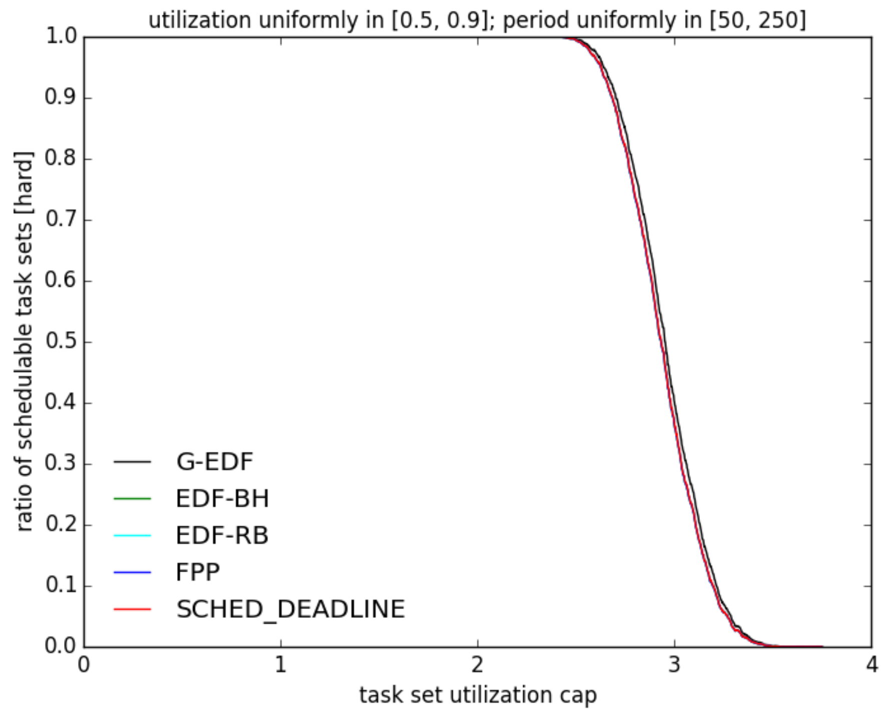


(a) Hard real-time

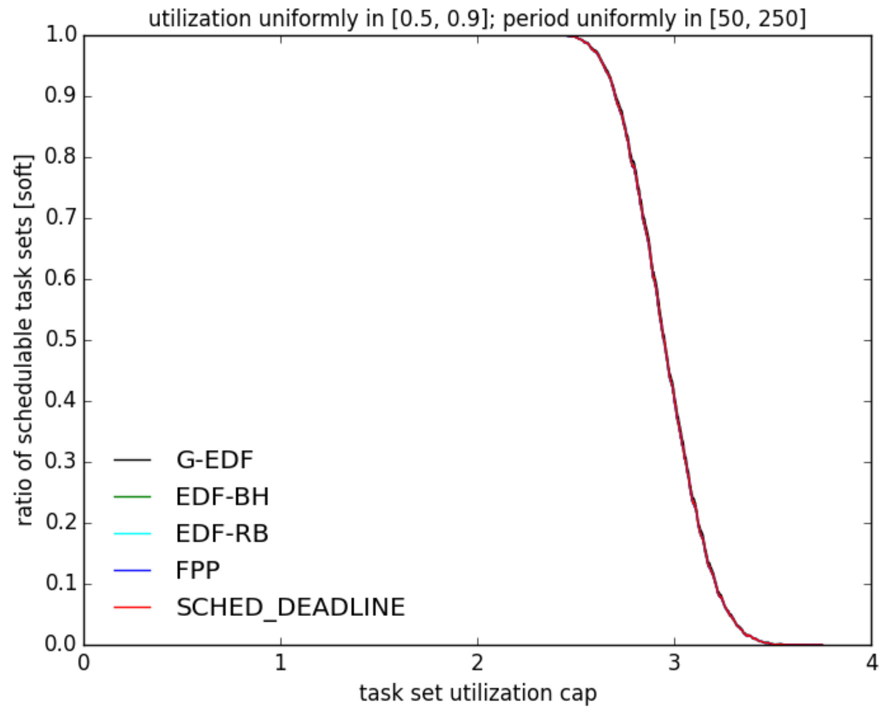


(b) Soft real-time

Figure A.14: Schedulability for task sets with uniformly distributed long periods and uniformly distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

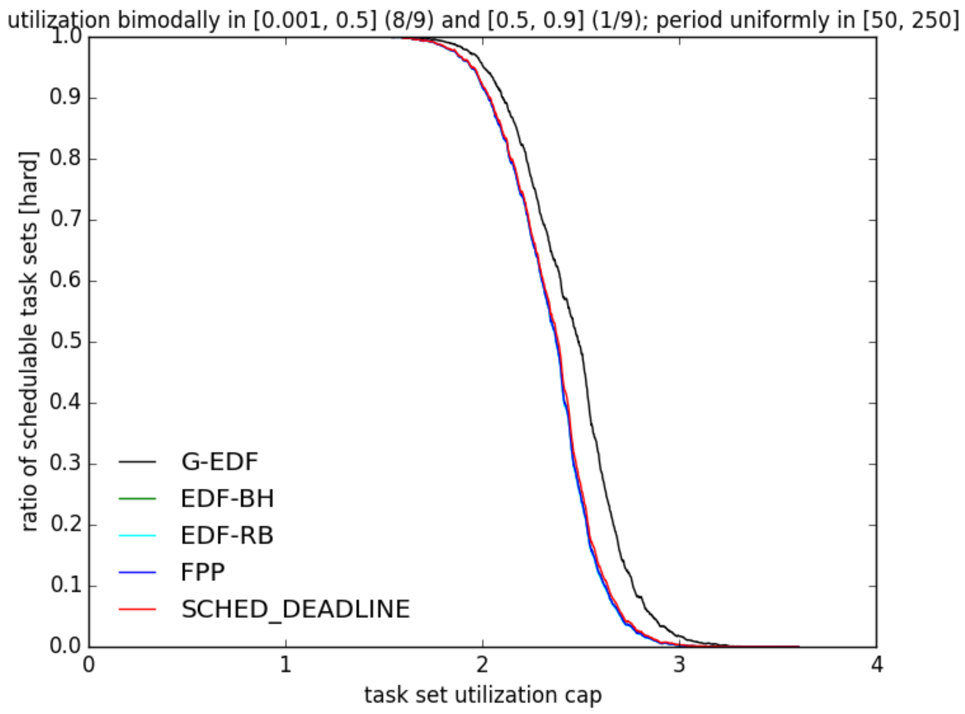


(a) Hard real-time

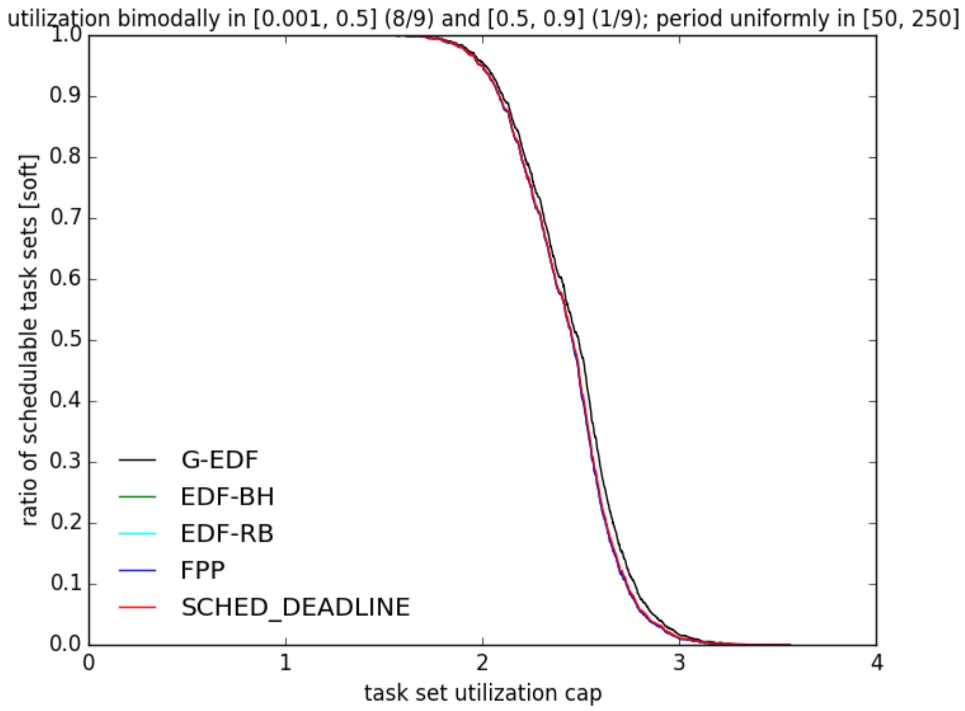


(b) Soft real-time

Figure A.15: Schedulability for task sets with uniformly distributed long periods and uniformly distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

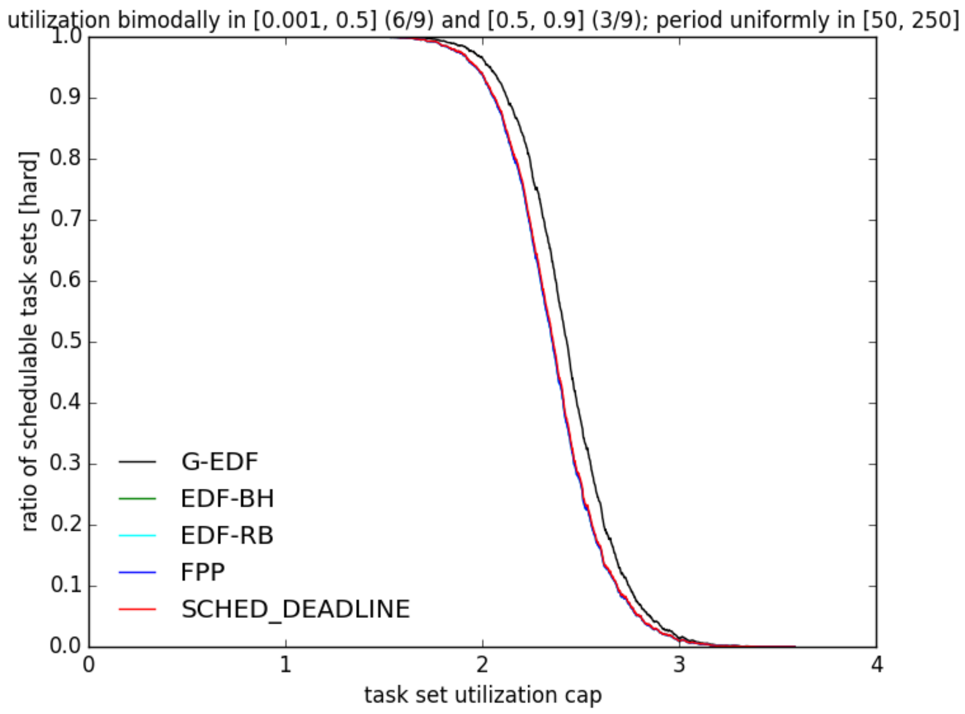


(a) Hard real-time

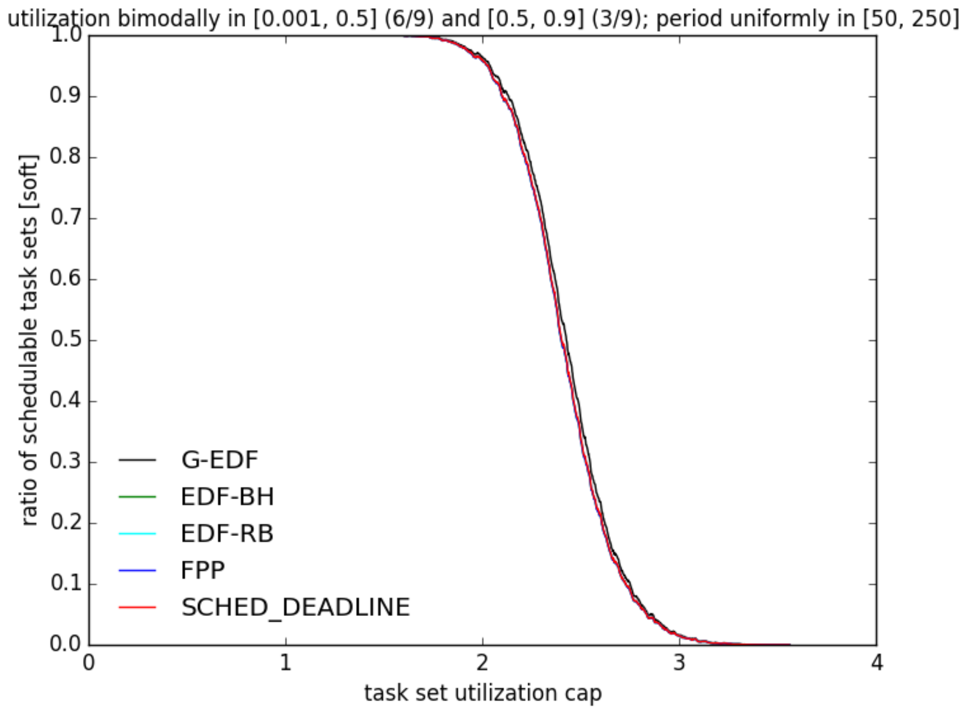


(b) Soft real-time

Figure A.16: Schedulability for task sets with uniformly distributed long periods and bimodally distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

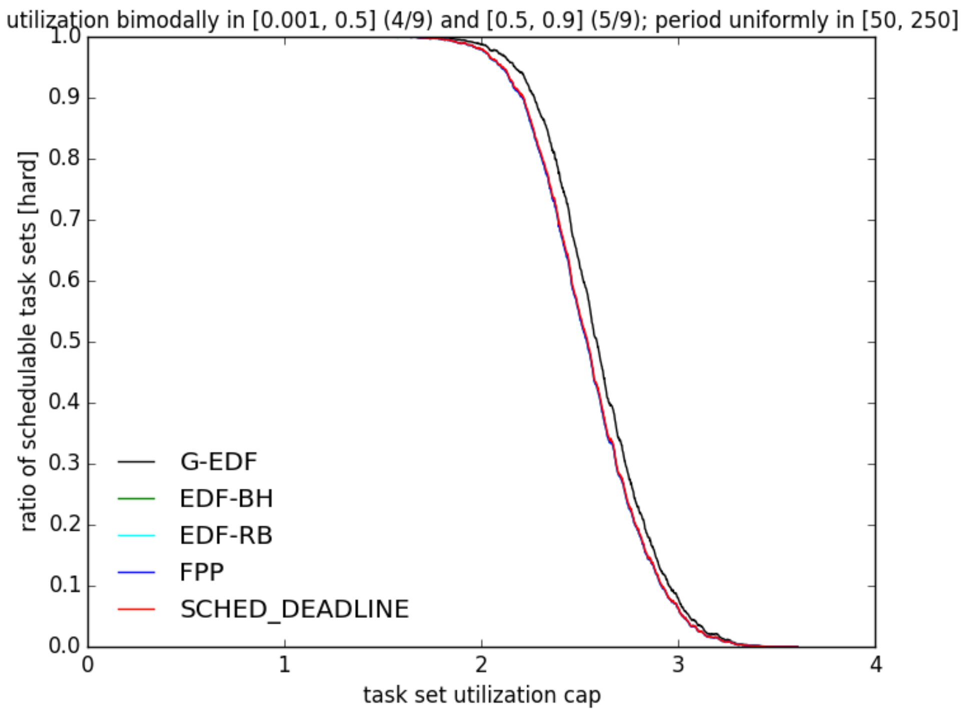


(a) Hard real-time

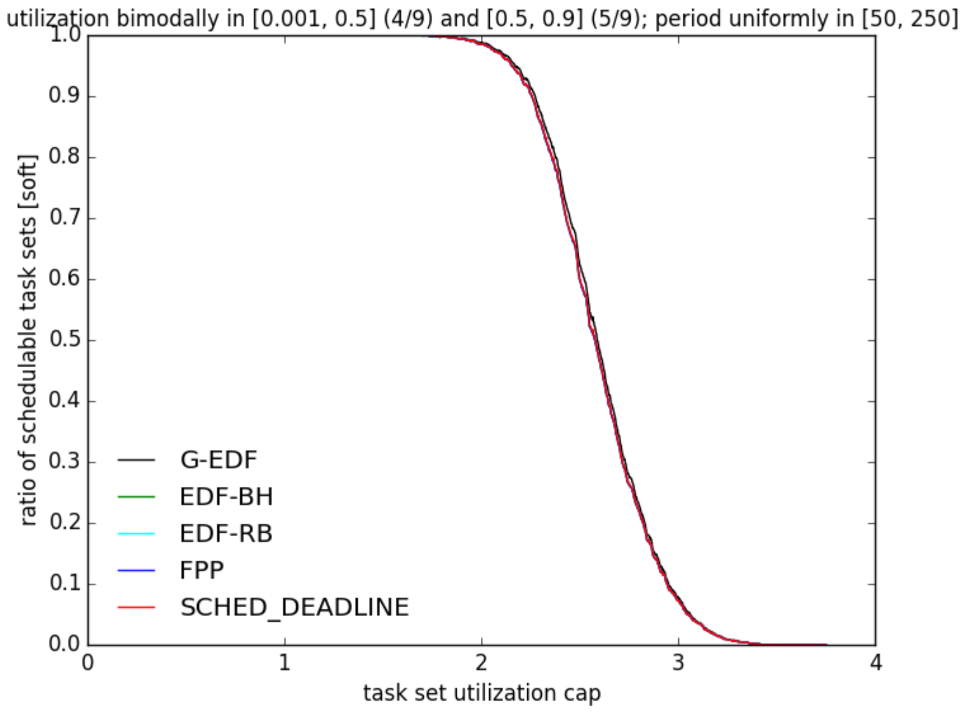


(b) Soft real-time

Figure A.17: Schedulability for task sets with uniformly distributed long periods and bimodally distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.



(a) Hard real-time



(b) Soft real-time

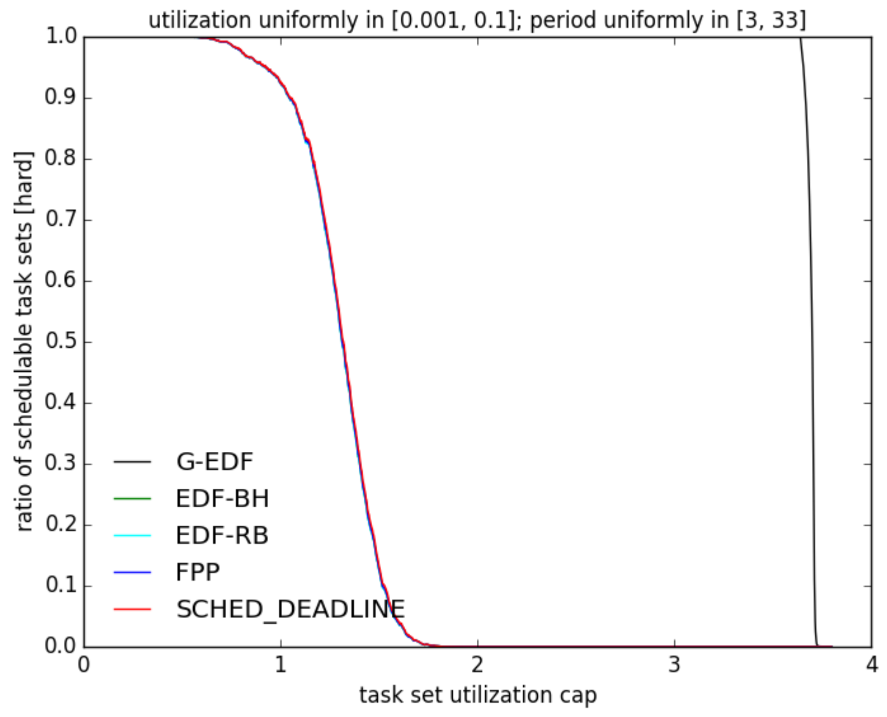
Figure A.18: Schedulability for task sets with uniformly distributed long periods and bimodally distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

B

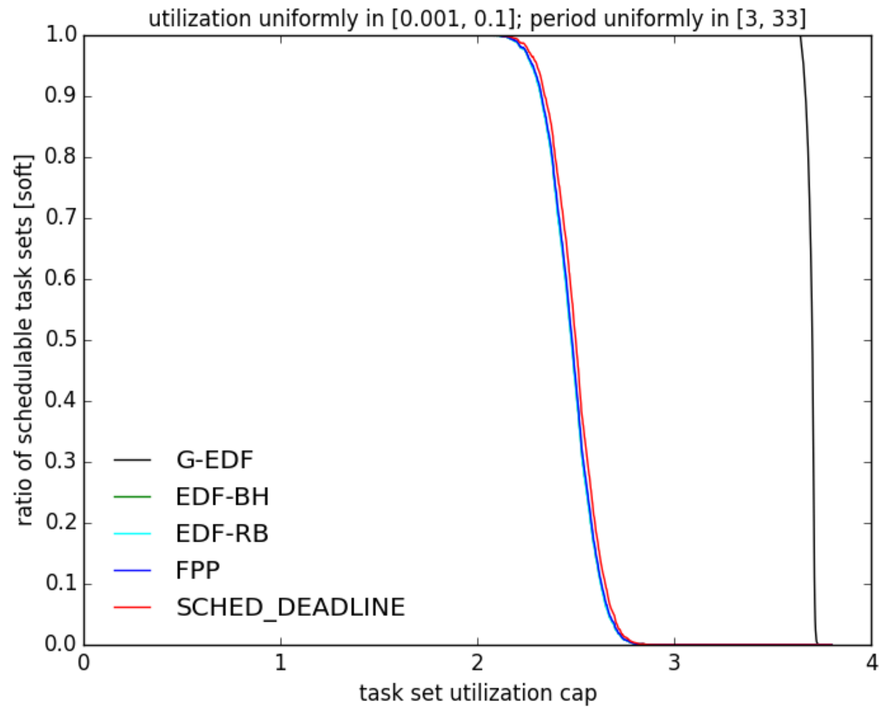
Complete Schedulability Results for ARM

This appendix contains the complete schedulability results for the Raspberry Pi 3 model B. The following 18 figures depict the schedulability results obtained for each combination of the period and utilization distributions listed in section 4.2.

- Figure B.1 shows schedulability for uniform light utilizations and short periods.
- Figure B.2 shows schedulability for uniform medium utilizations and short periods.
- Figure B.3 shows schedulability for uniform heavy utilizations and short periods.
- Figure B.4 shows schedulability for bimodal light utilizations and short periods.
- Figure B.5 shows schedulability for bimodal medium utilizations and short periods.
- Figure B.6 shows schedulability for bimodal heavy utilizations and short periods.
- Figure B.7 shows schedulability for uniform light utilizations and moderate periods.
- Figure B.8 shows schedulability for uniform medium utilizations and moderate periods.
- Figure B.9 shows schedulability for uniform heavy utilizations and moderate periods.
- Figure B.10 shows schedulability for bimodal light utilizations and moderate periods.
- Figure B.11 shows schedulability for bimodal medium utilizations and moderate periods.
- Figure B.12 shows schedulability for bimodal heavy utilizations and moderate periods.
- Figure B.13 shows schedulability for uniform light utilizations and long periods.
- Figure B.14 shows schedulability for uniform medium utilizations and long periods.
- Figure B.15 shows schedulability for uniform heavy utilizations and long periods.
- Figure B.16 shows schedulability for bimodal light utilizations and long periods.
- Figure B.17 shows schedulability for bimodal medium utilizations and long periods.
- Figure B.18 shows schedulability for bimodal heavy utilizations and long periods.

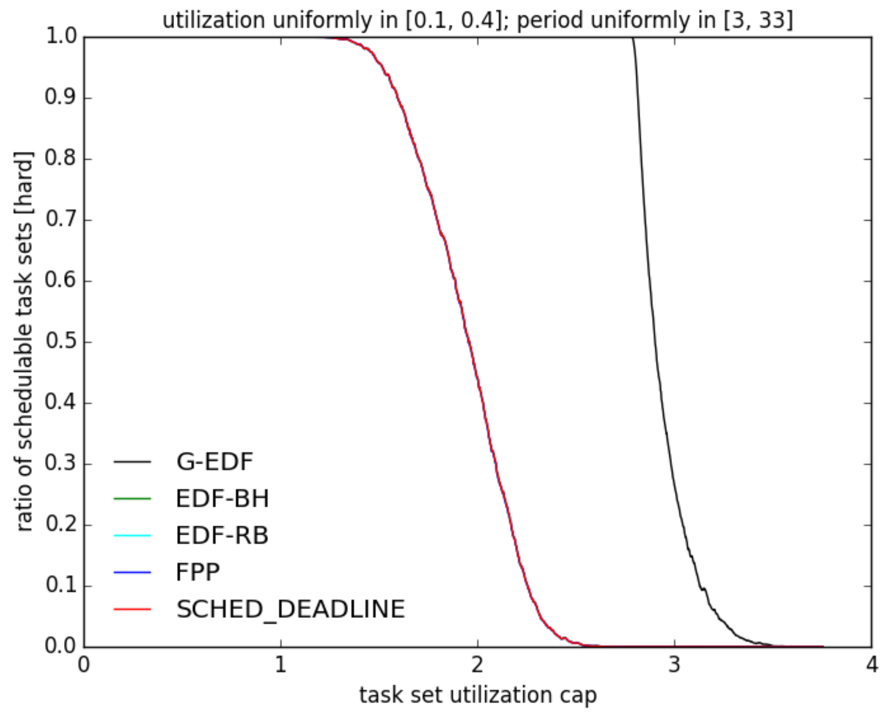


(a) Hard real-time

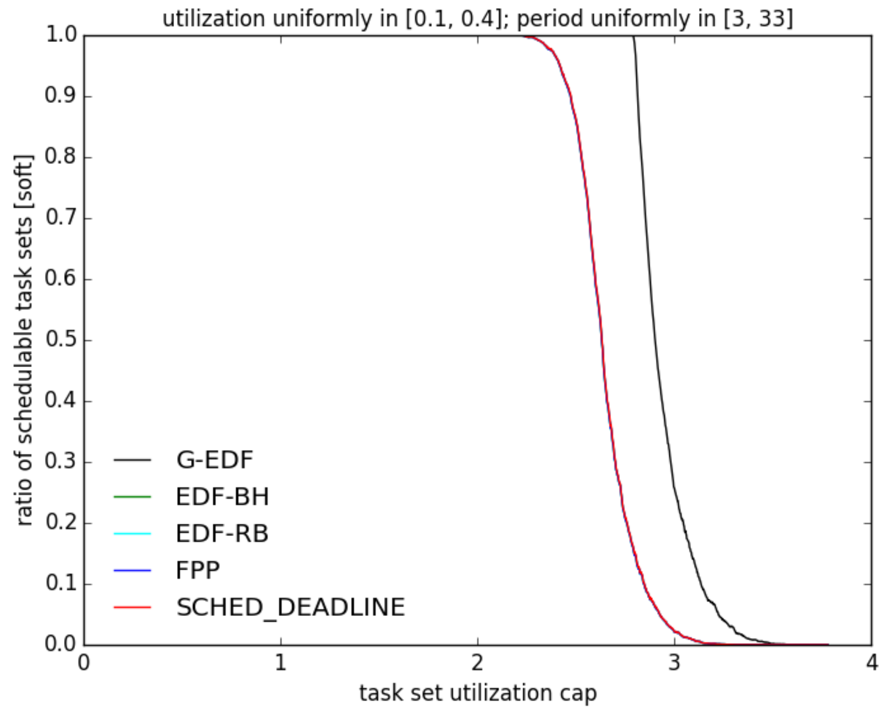


(b) Soft real-time

Figure B.1: Schedulability for task sets with uniformly distributed short periods and uniformly distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

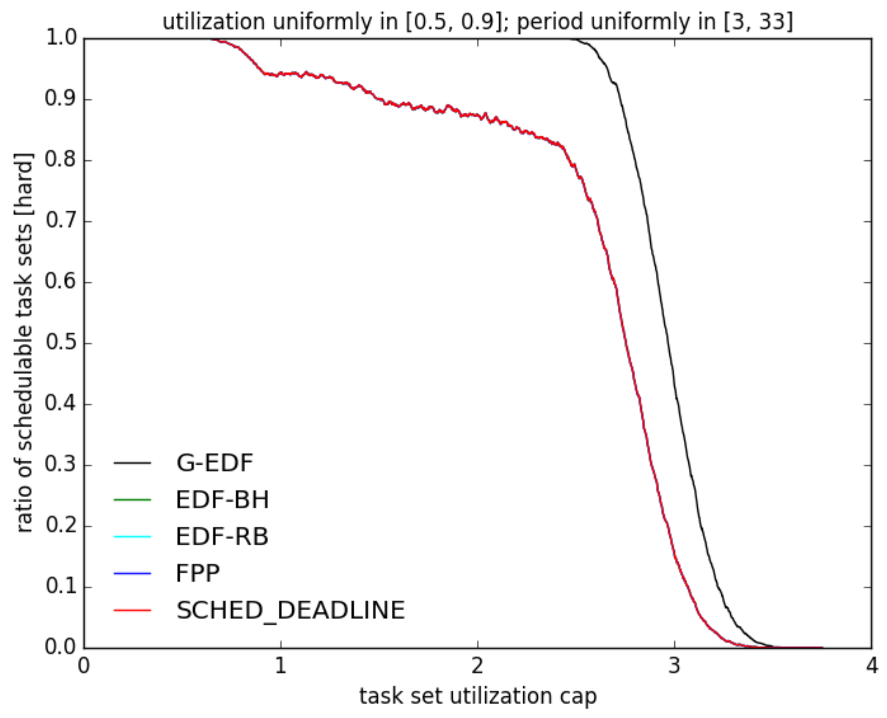


(a) Hard real-time

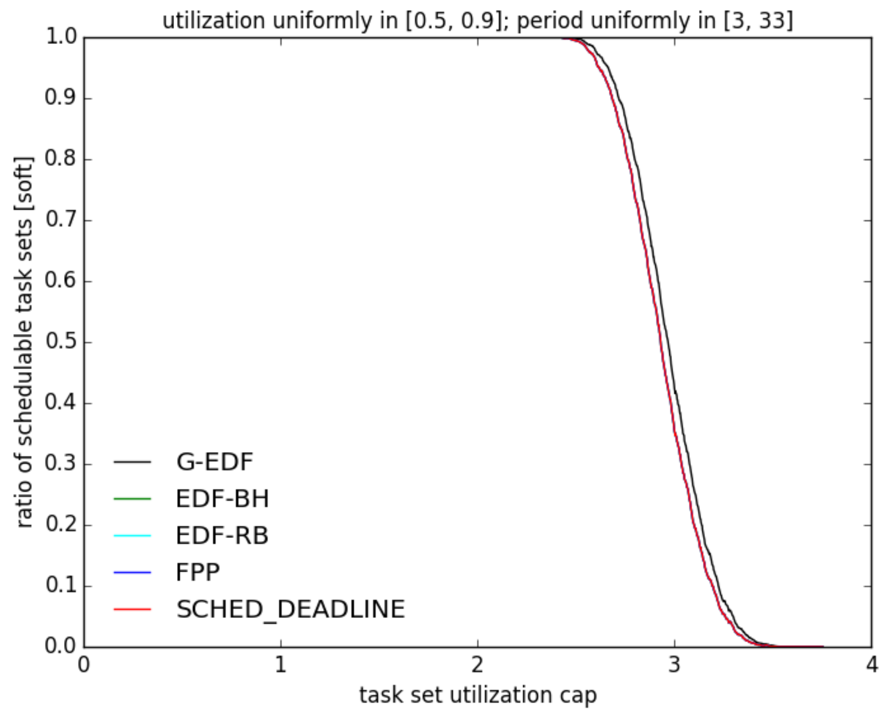


(b) Soft real-time

Figure B.2: Schedulability for task sets with uniformly distributed short periods and uniformly distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

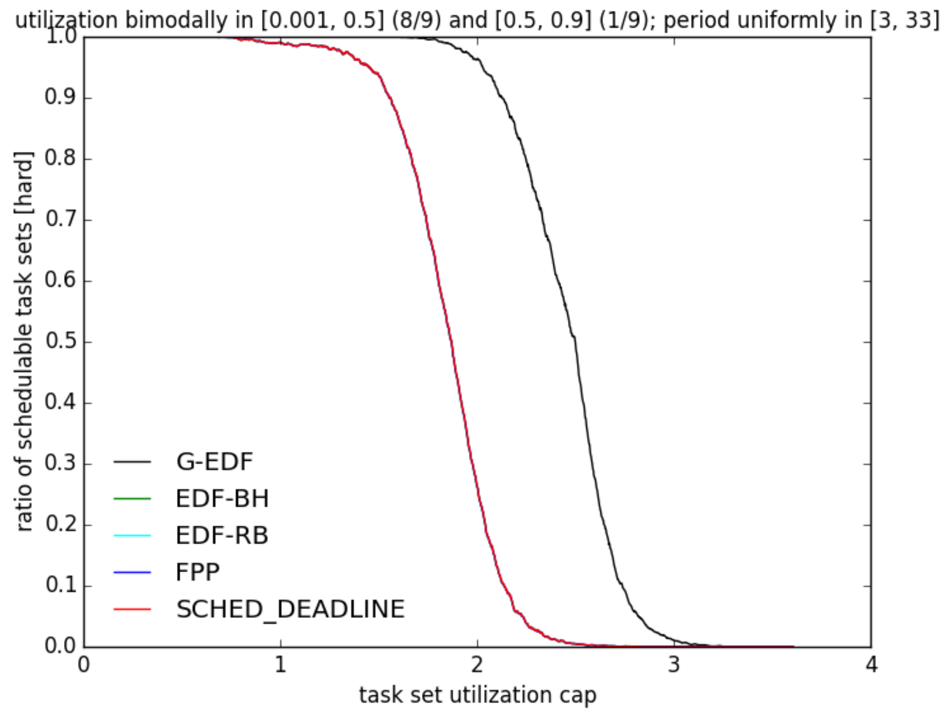


(a) Hard real-time

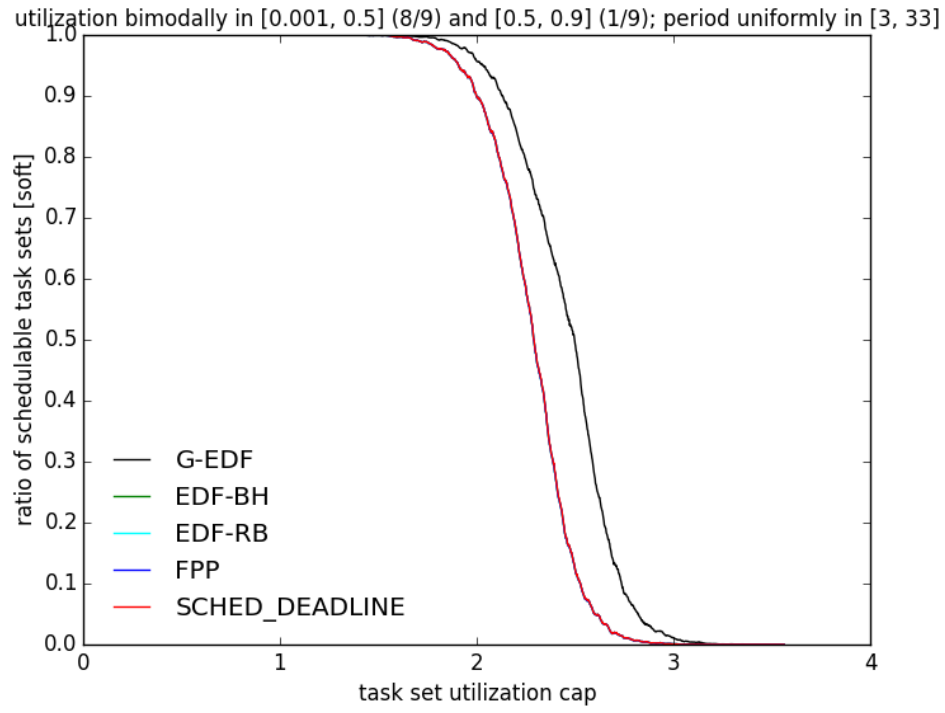


(b) Soft real-time

Figure B.3: Schedulability for task sets with uniformly distributed short periods and uniformly distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

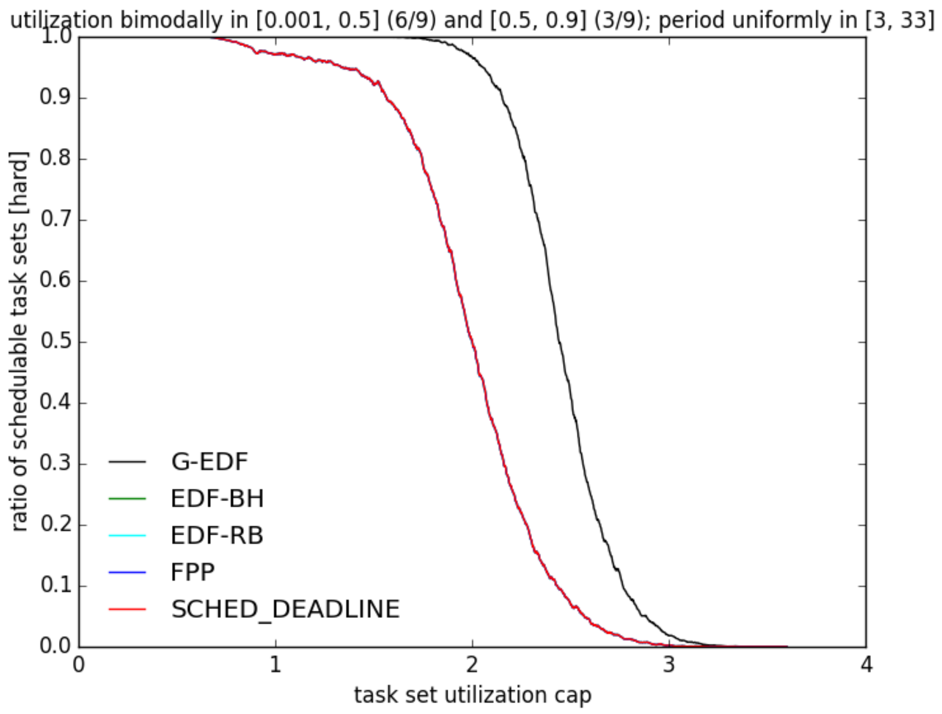


(a) Hard real-time

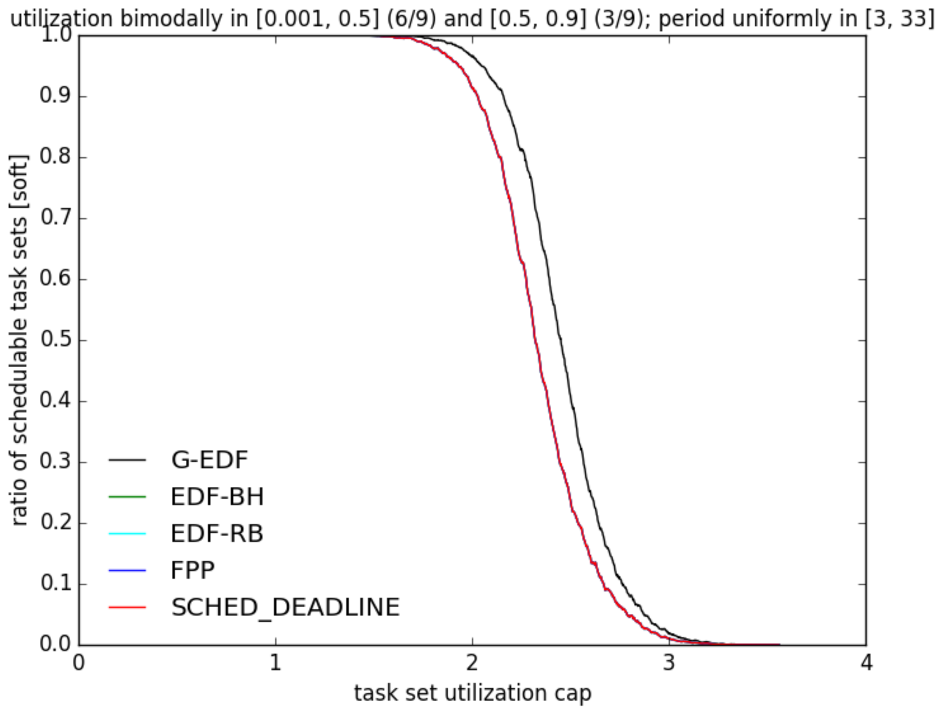


(b) Soft real-time

Figure B.4: Schedulability for task sets with uniformly distributed short periods and bimodally distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

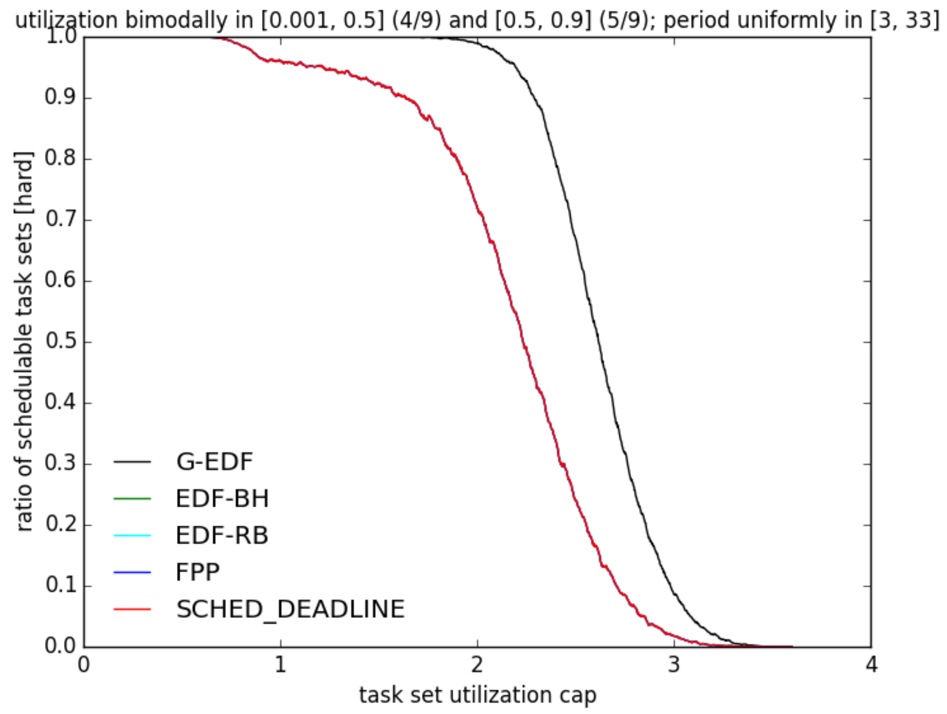


(a) Hard real-time

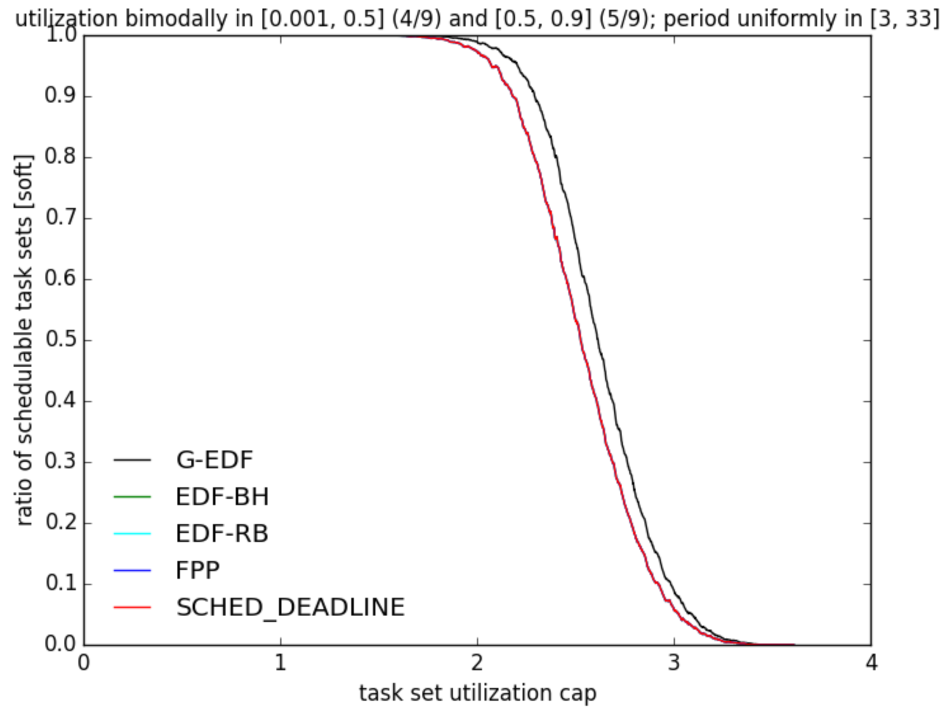


(b) Soft real-time

Figure B.5: Schedulability for task sets with uniformly distributed short periods and bimodally distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

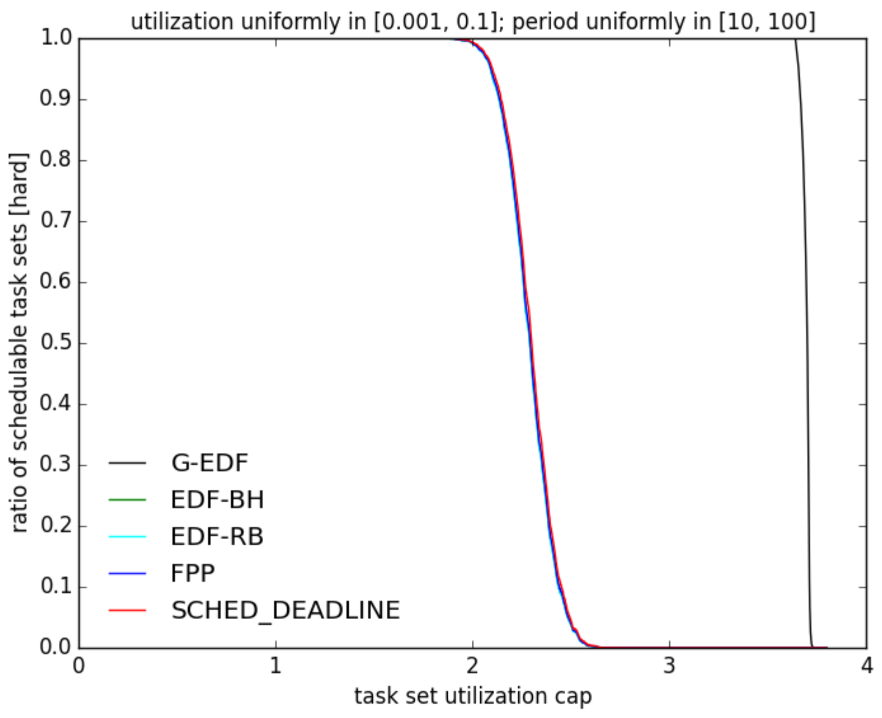


(a) Hard real-time

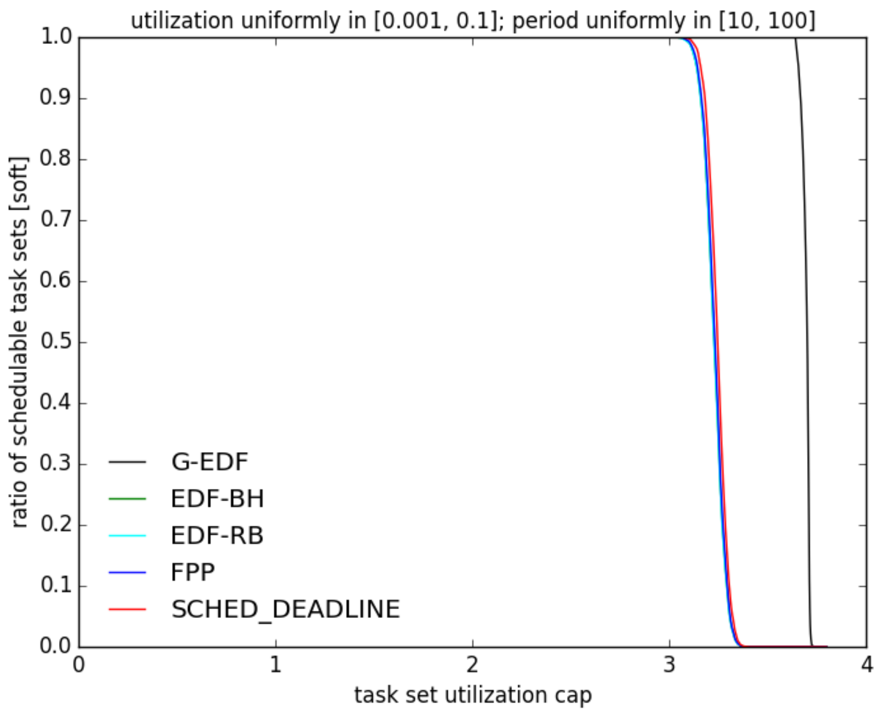


(b) Soft real-time

Figure B.6: Schedulability for task sets with uniformly distributed short periods and bimodally distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

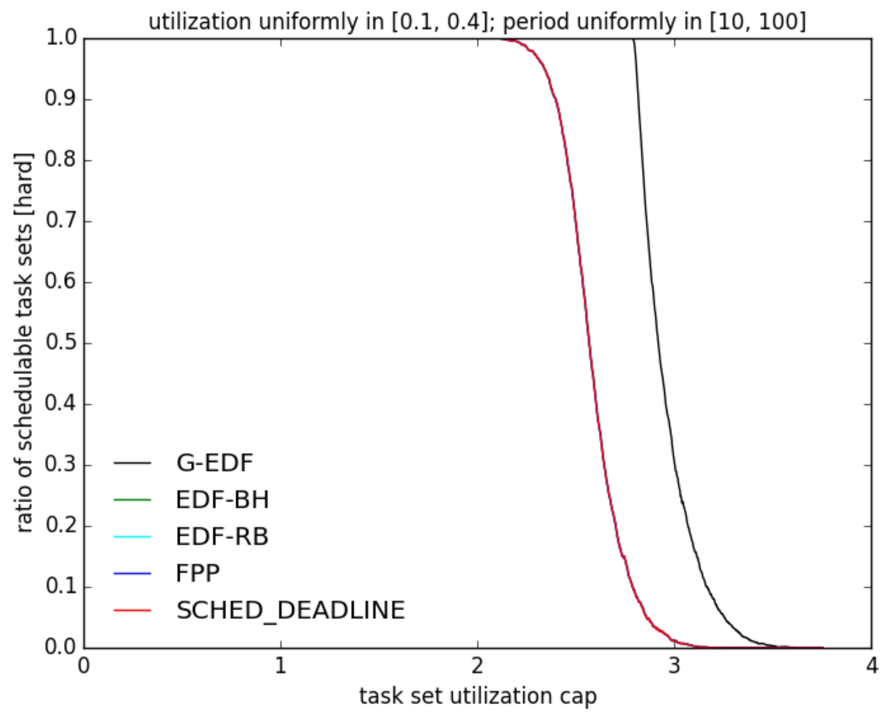


(a) Hard real-time

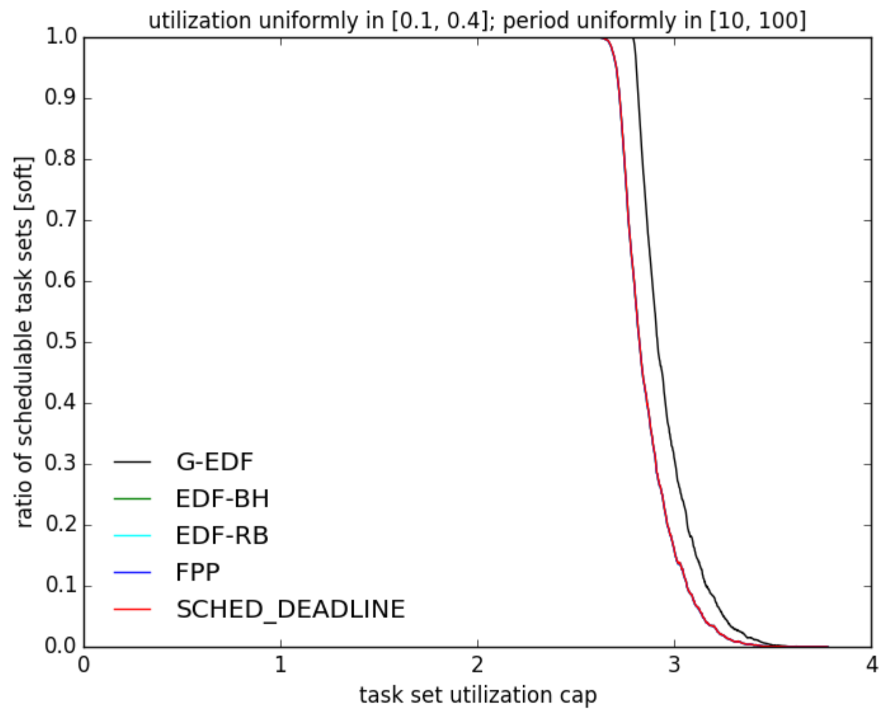


(b) Soft real-time

Figure B.7: Schedulability for task sets with uniformly distributed moderate periods and uniformly distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

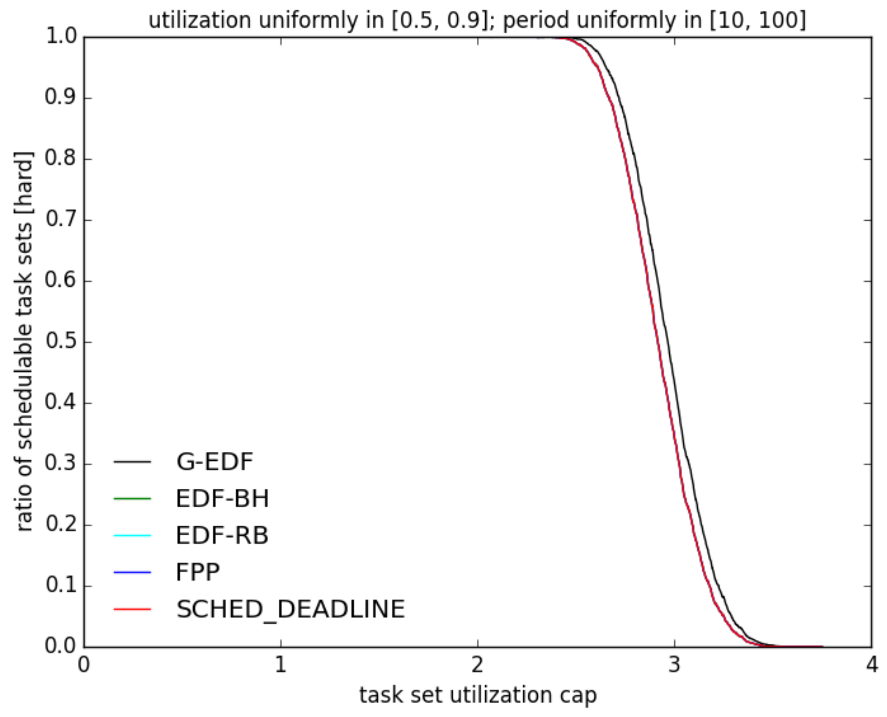


(a) Hard real-time

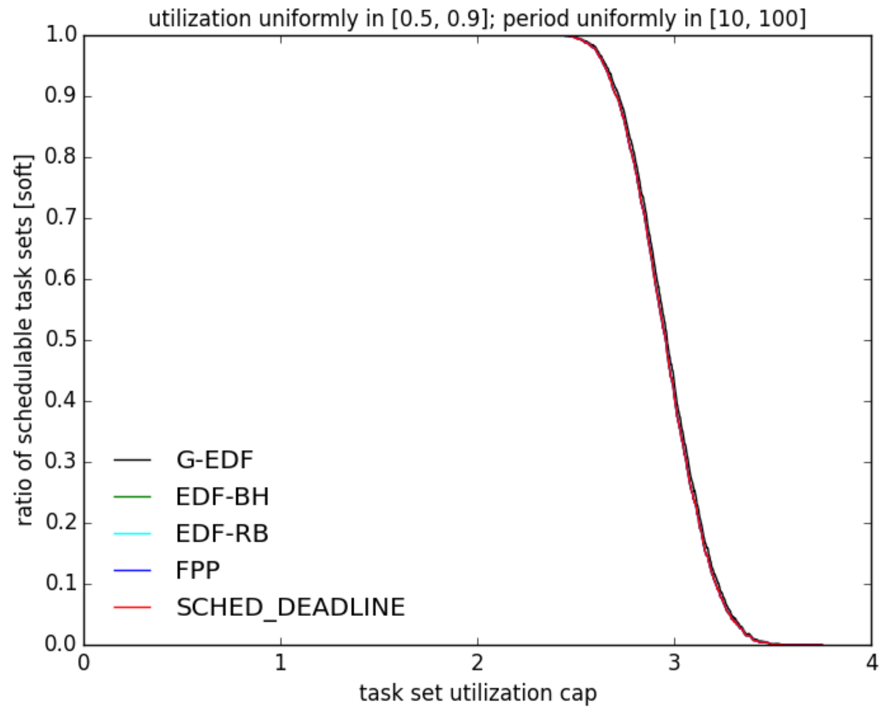


(b) Soft real-time

Figure B.8: Schedulability for task sets with uniformly distributed moderate periods and uniformly distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

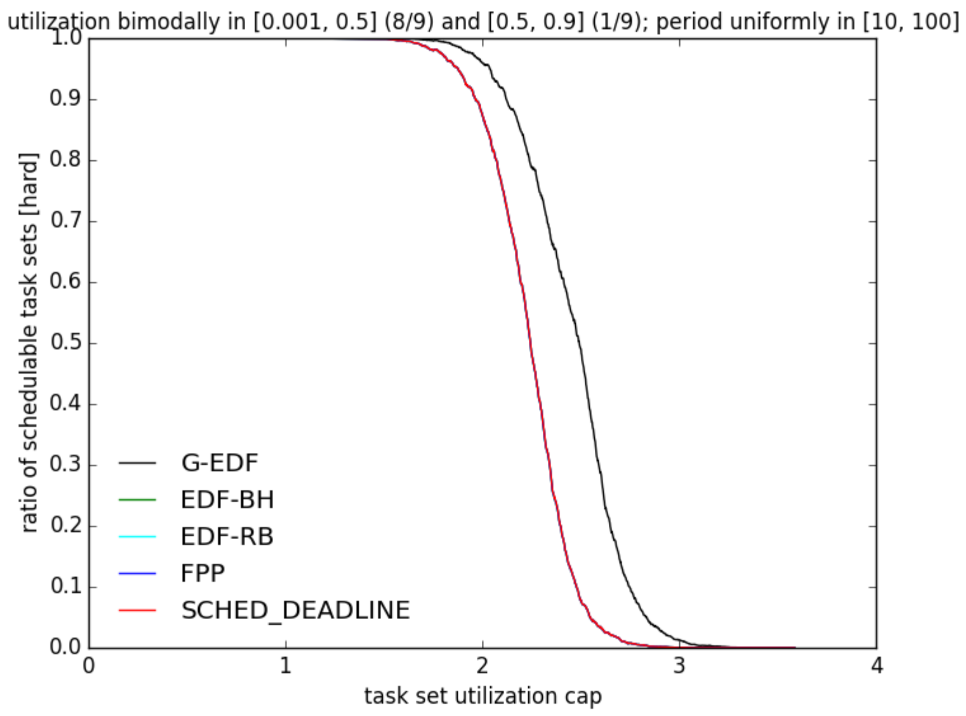


(a) Hard real-time

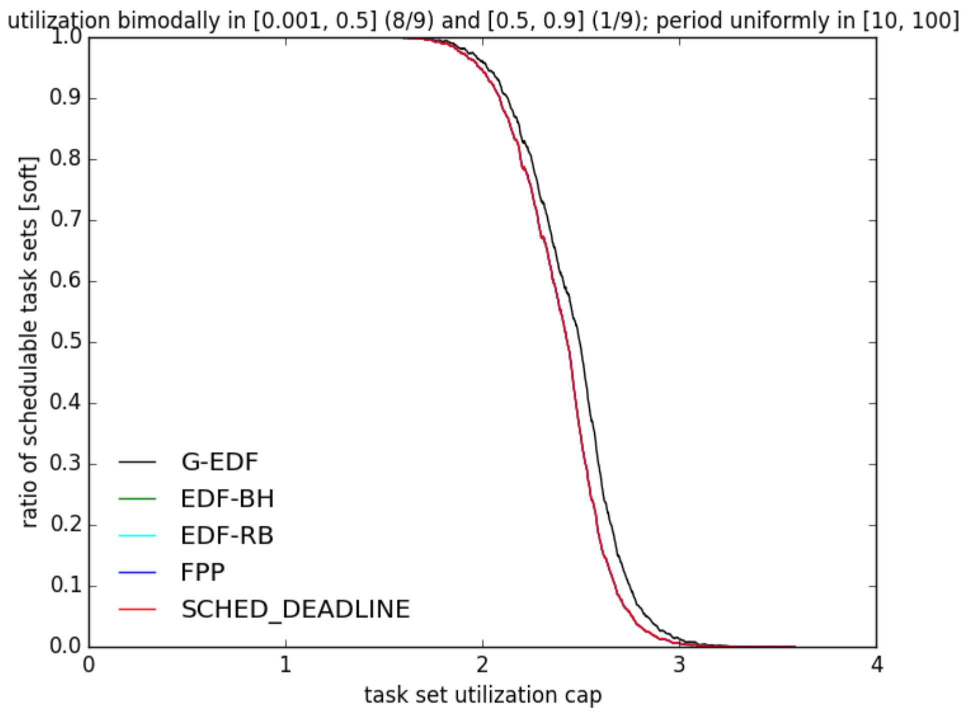


(b) Soft real-time

Figure B.9: Schedulability for task sets with uniformly distributed moderate periods and uniformly distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

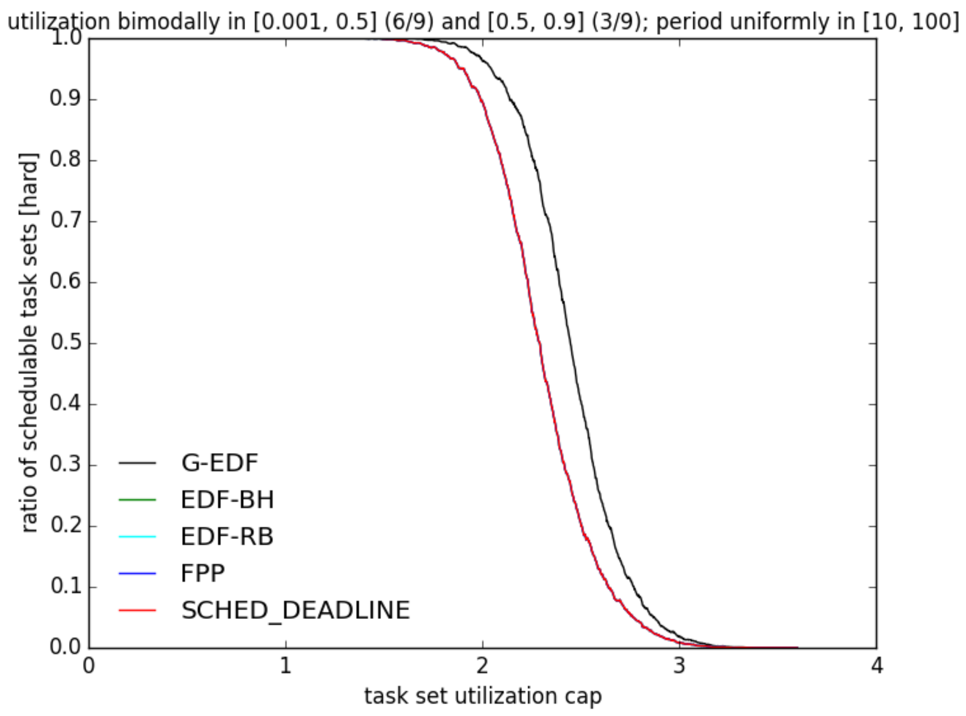


(a) Hard real-time

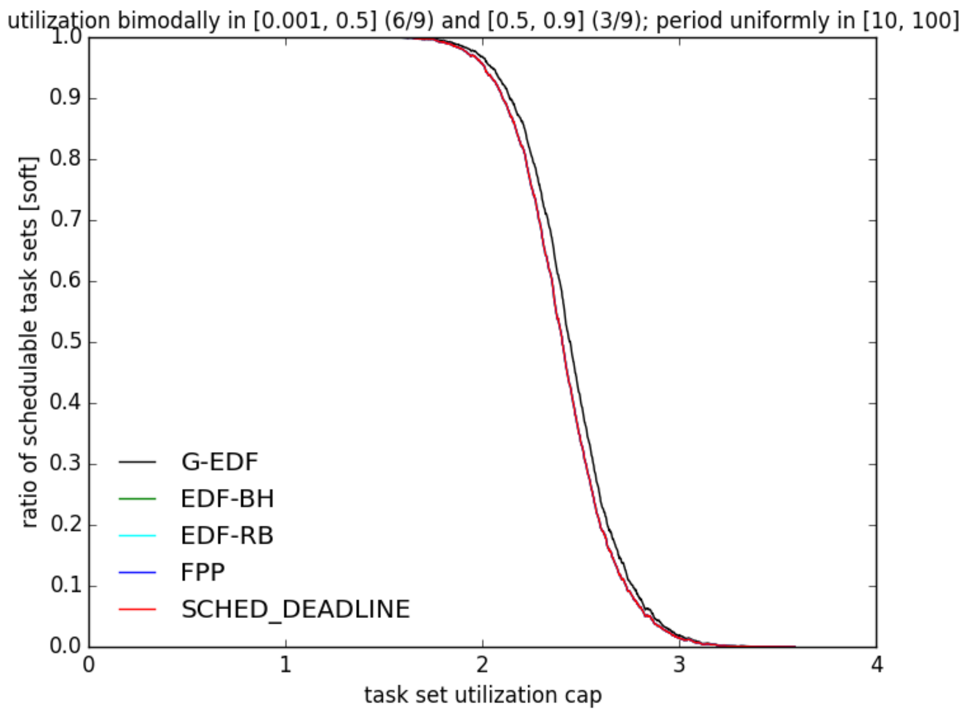


(b) Soft real-time

Figure B.10: Schedulability for task sets with uniformly distributed moderate periods and bimodally distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

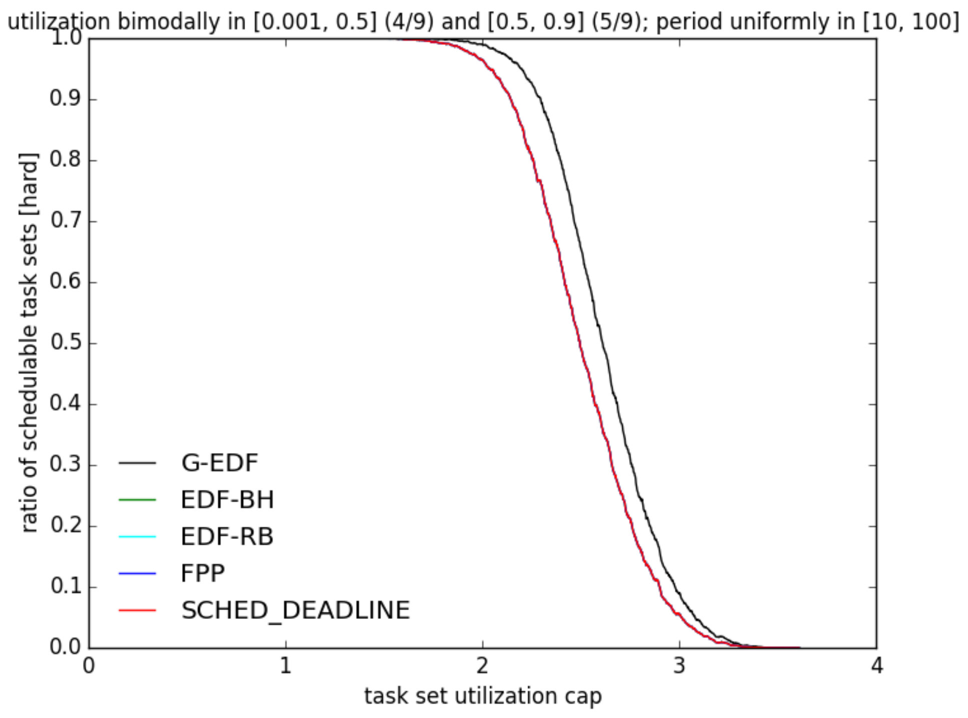


(a) Hard real-time

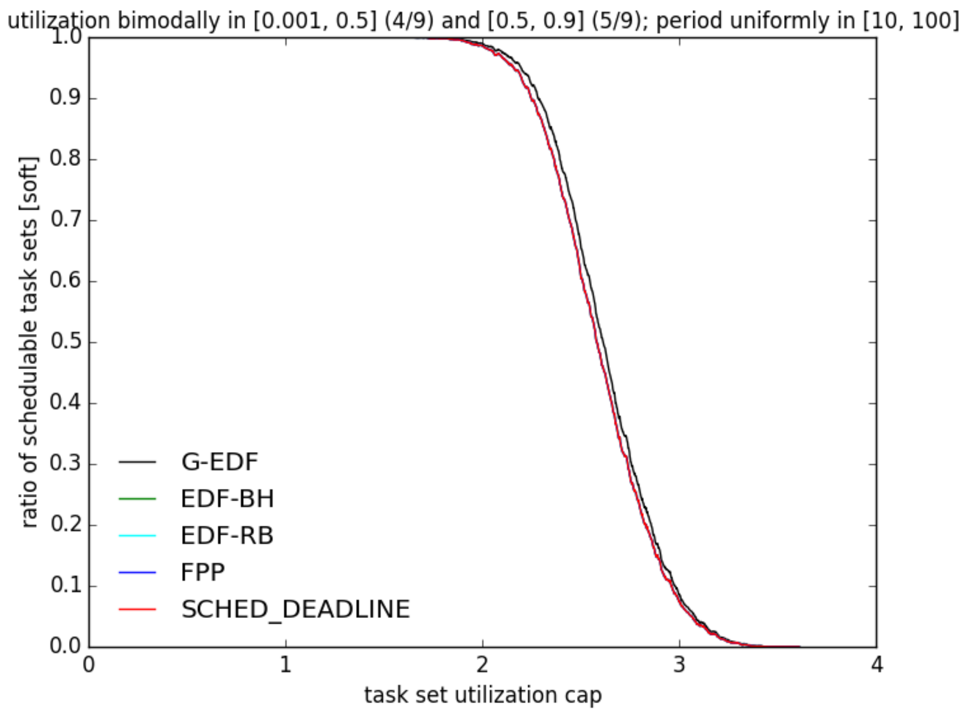


(b) Soft real-time

Figure B.11: Schedulability for task sets with uniformly distributed moderate periods and bimodally distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

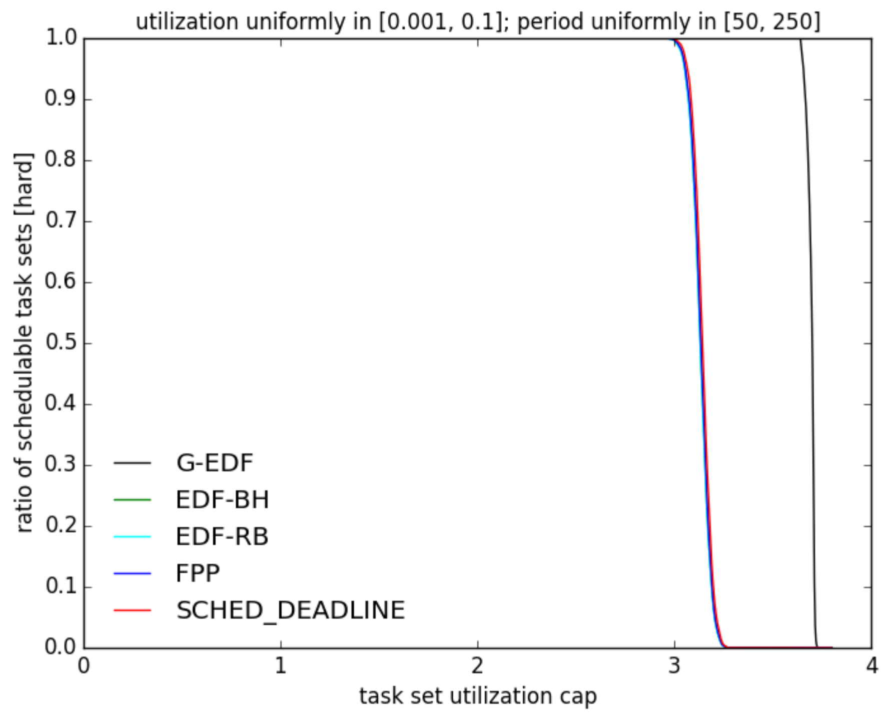


(a) Hard real-time

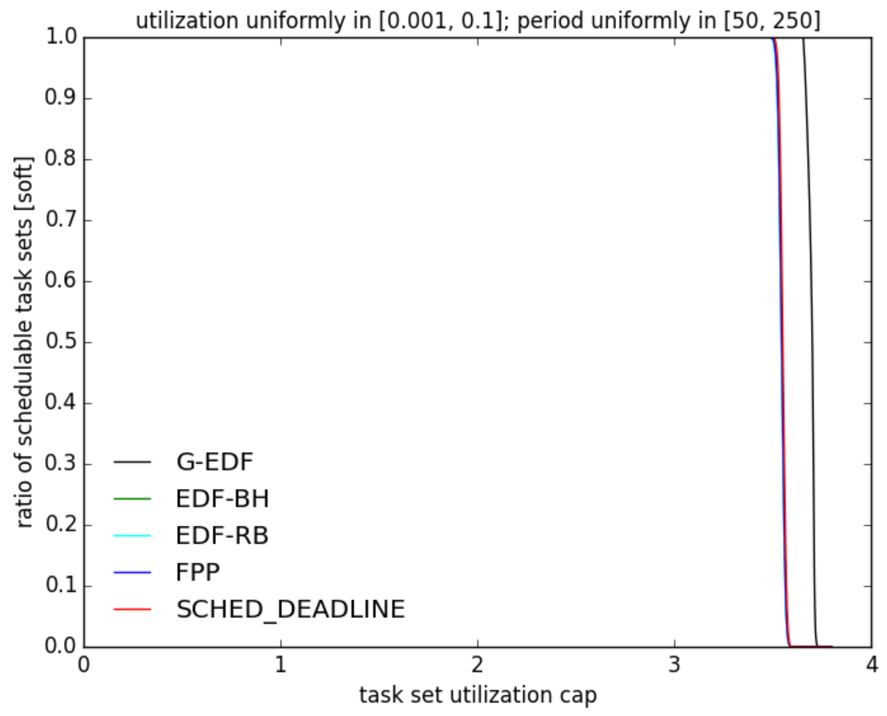


(b) Soft real-time

Figure B.12: Schedulability for task sets with uniformly distributed moderate periods and bimodally distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

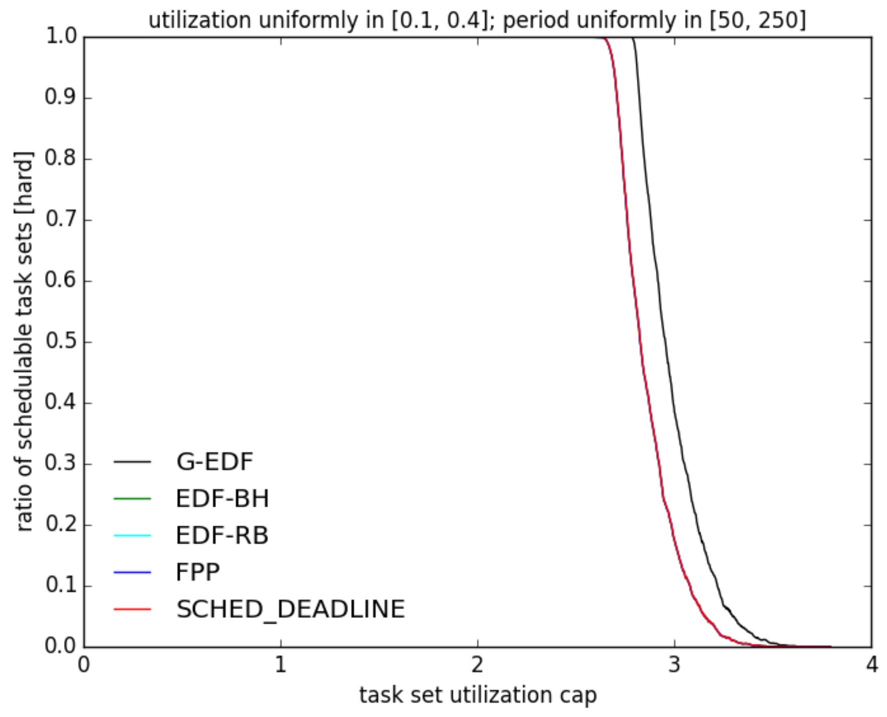


(a) Hard real-time

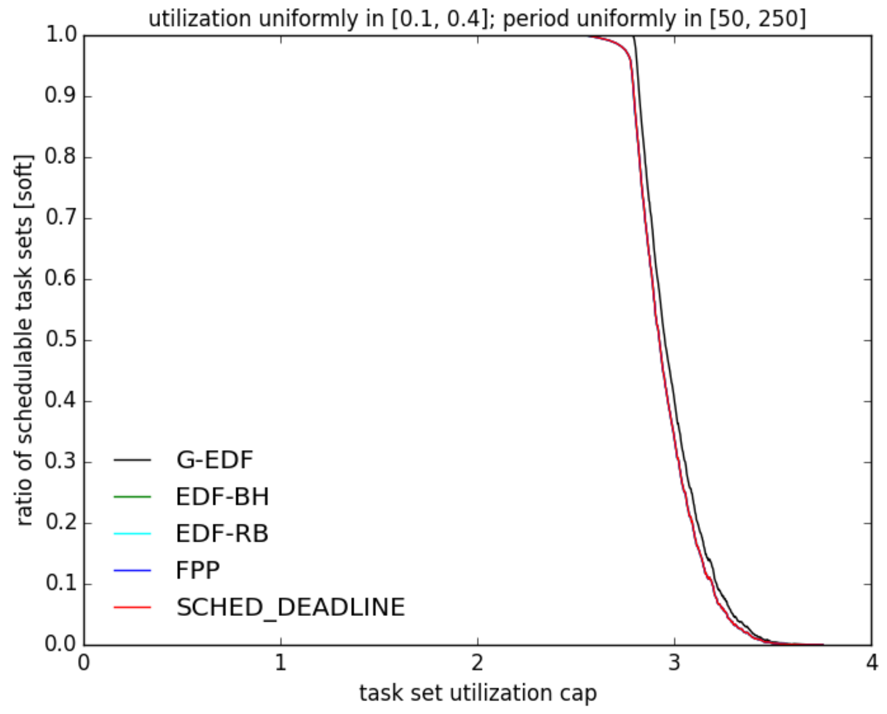


(b) Soft real-time

Figure B.13: Schedulability for task sets with uniformly distributed long periods and uniformly distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

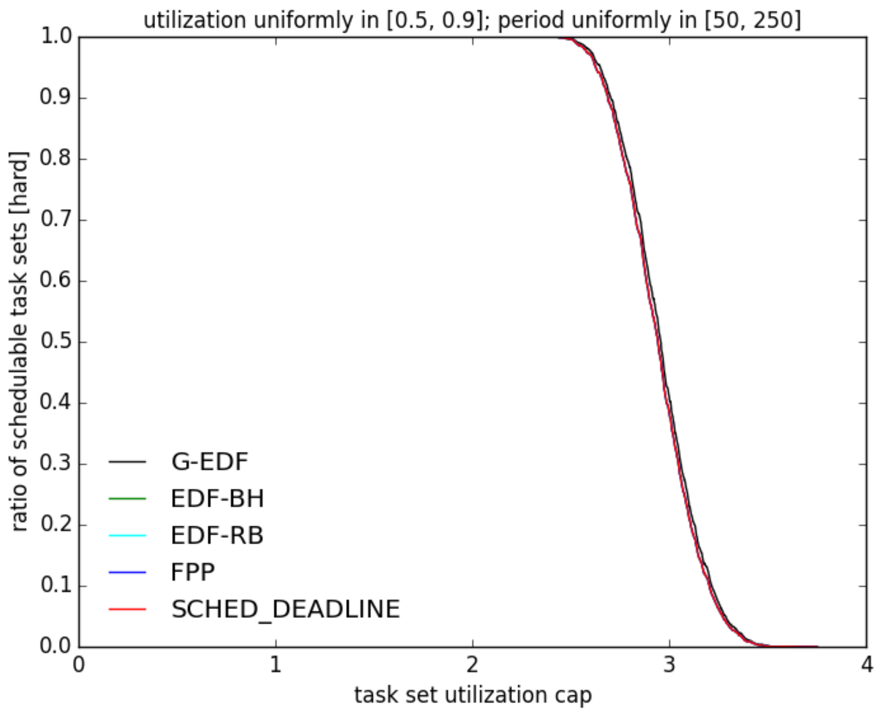


(a) Hard real-time

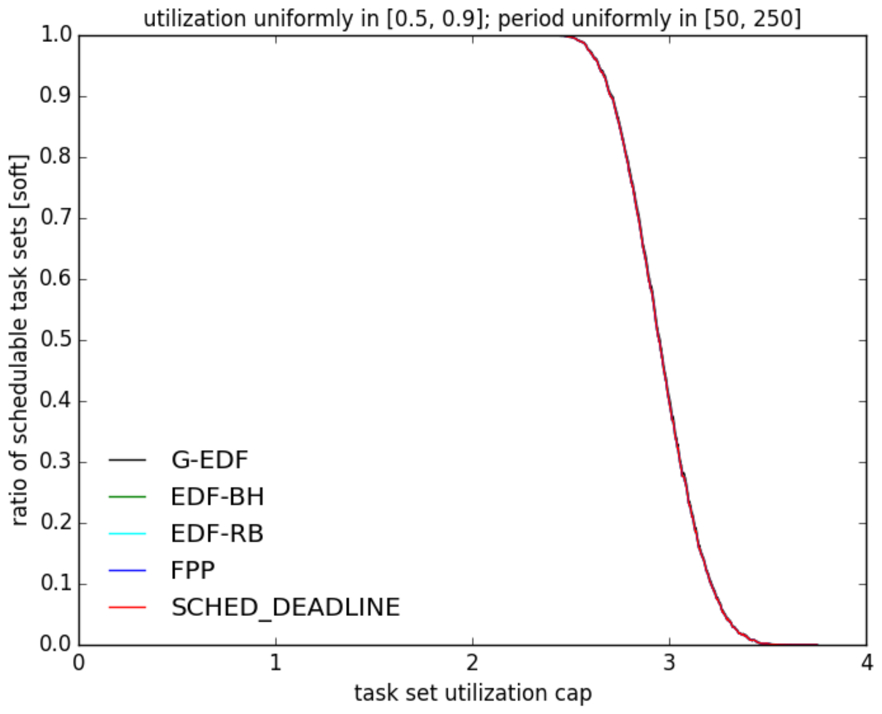


(b) Soft real-time

Figure B.14: Schedulability for task sets with uniformly distributed long periods and uniformly distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

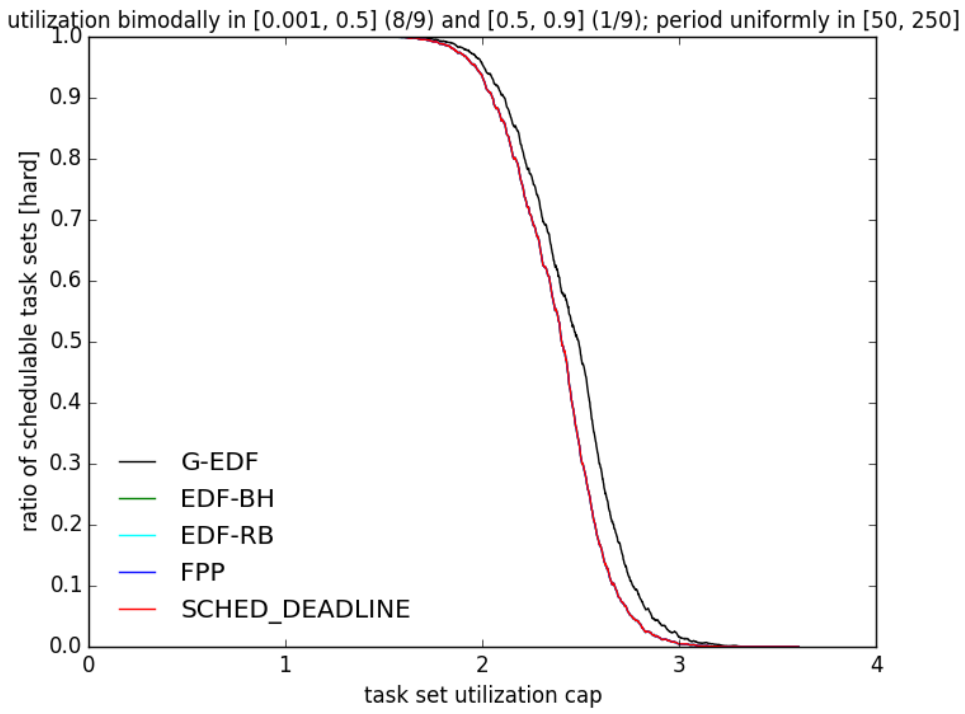


(a) Hard real-time

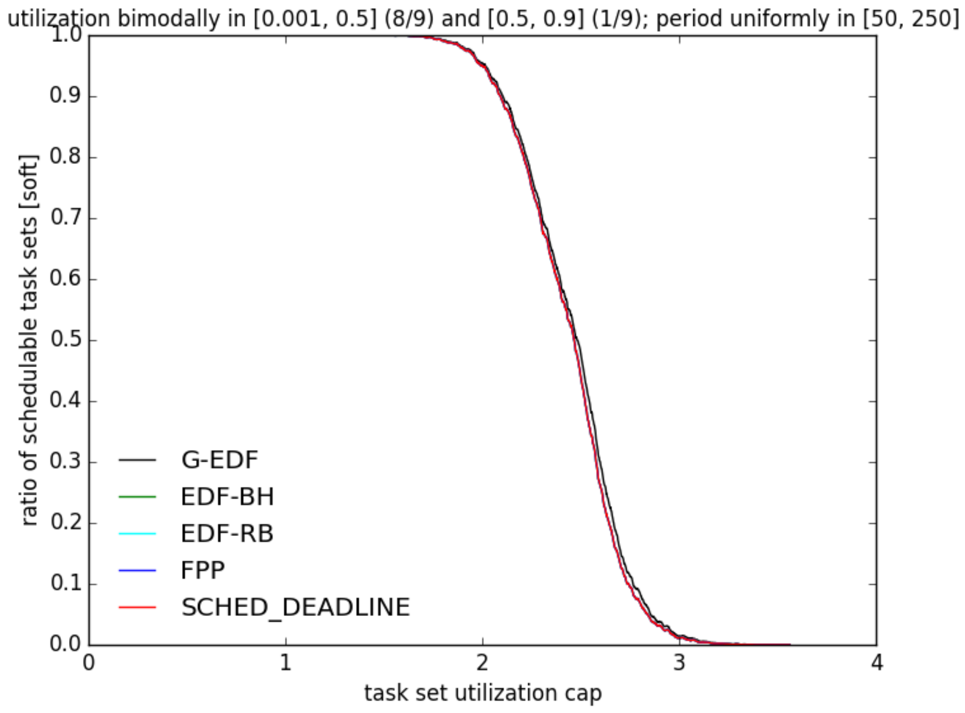


(b) Soft real-time

Figure B.15: Schedulability for task sets with uniformly distributed long periods and uniformly distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

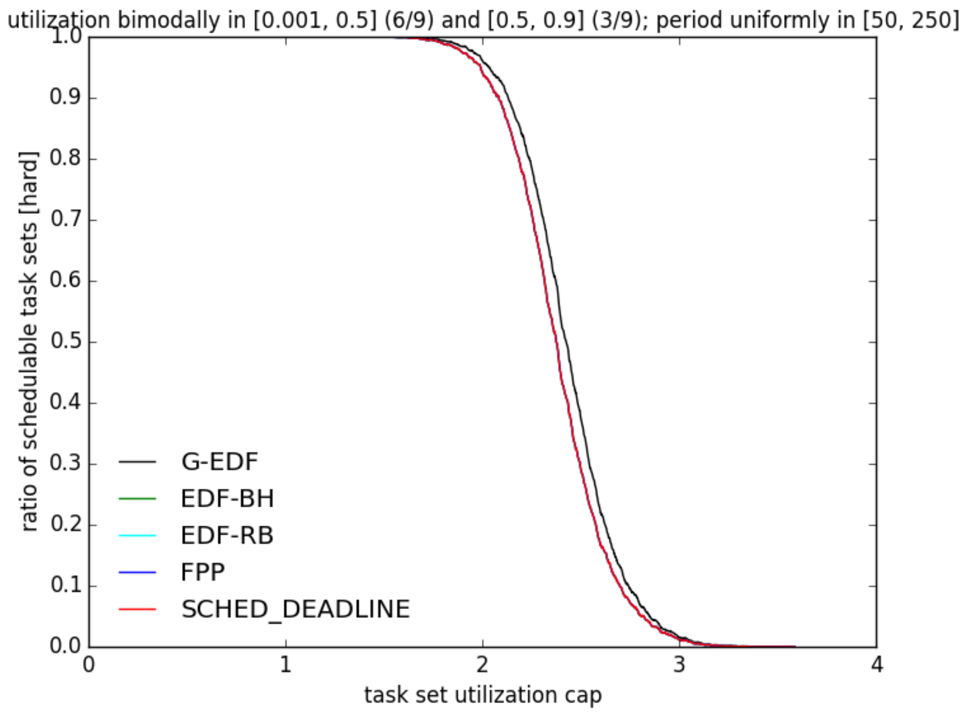


(a) Hard real-time

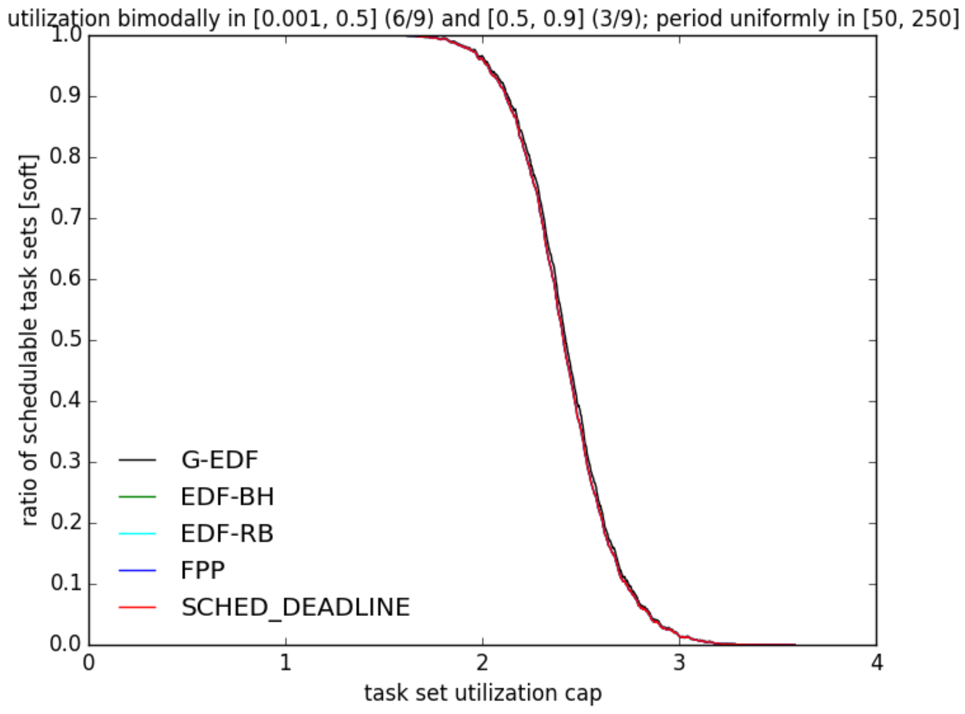


(b) Soft real-time

Figure B.16: Schedulability for task sets with uniformly distributed long periods and bimodally distributed light utilizations. The black line (G-EDF) represent the theoretical bound without overheads.

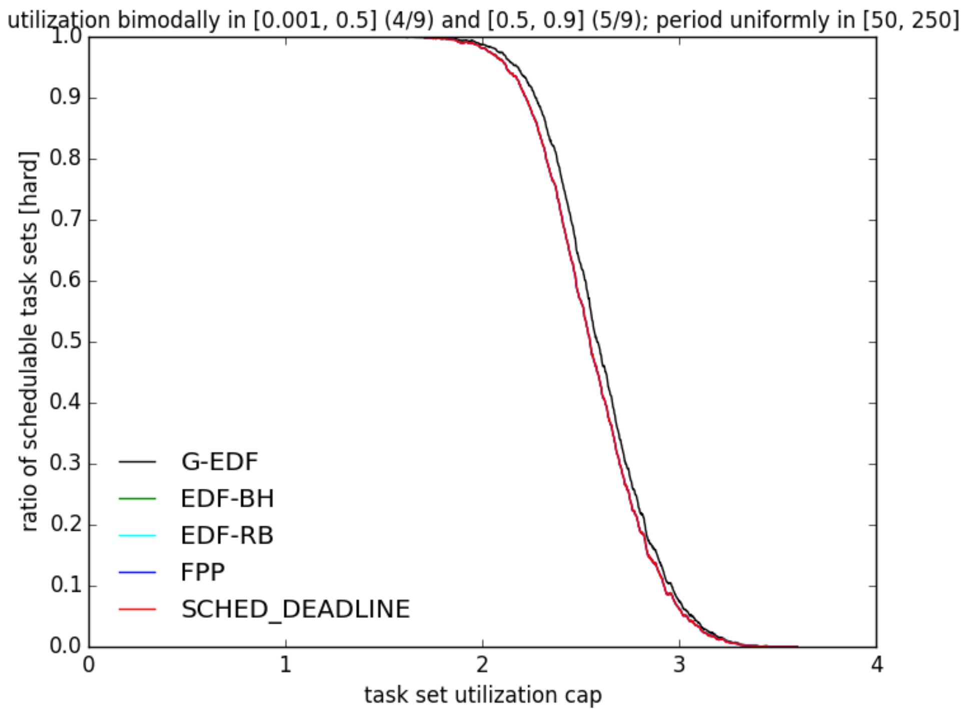


(a) Hard real-time

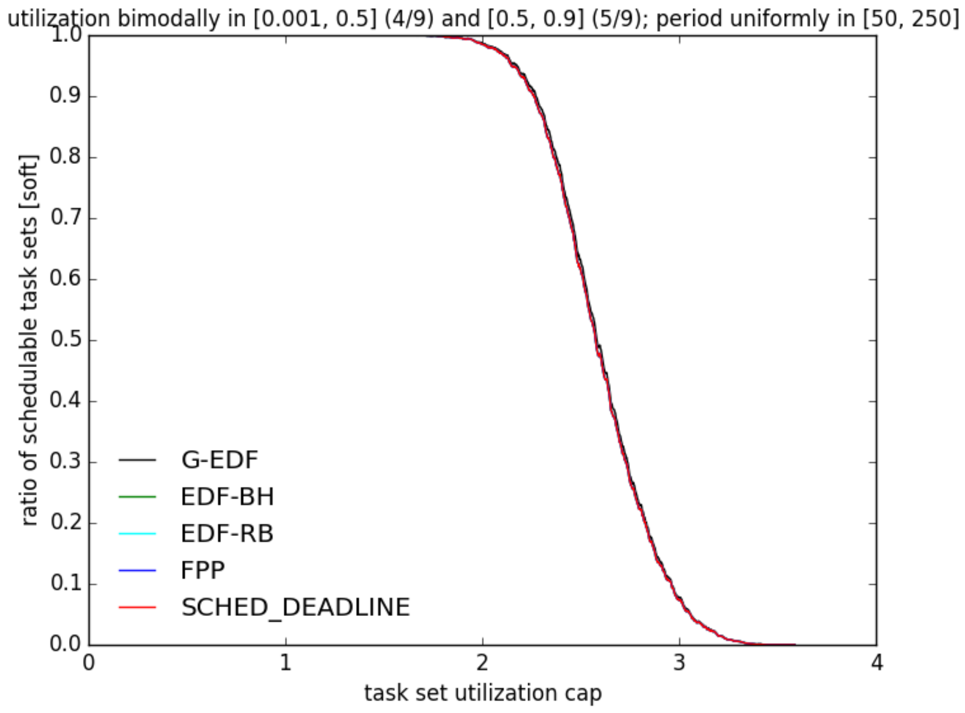


(b) Soft real-time

Figure B.17: Schedulability for task sets with uniformly distributed long periods and bimodally distributed medium utilizations. The black line (G-EDF) represent the theoretical bound without overheads.



(a) Hard real-time



(b) Soft real-time

Figure B.18: Schedulability for task sets with uniformly distributed long periods and bimodally distributed heavy utilizations. The black line (G-EDF) represent the theoretical bound without overheads.