Evaluation of scheduling algorithms for mixed criticality workloads on multi-core embedded systems

Robin Klusman

August 16, 2016

Supervisors: Andy Pimentel, Jun Xiao
Signed:
Abstract

This paper presents an evaluation of scheduling algorithms for use with mixed criticality workloads on multi-core systems using LitmusRT. By performing experiments on multiple different types of schedulers, including Pfair, EDF, FP, FIFO and RR, we determine what the advantages and disadvantages of different schedulers are. In these experiments we find the maximum feasible load with our specific taskset for each scheduler, while also gathering information on the total amount of context switches and the response time of non-real-time, best effort tasks. In addition to testing the schedulers we validate the correctness of the real-time operating system simulator, SysRT. Showing that it can correctly simulate real-time systems and schedulers, by comparing results in SysRT to results in LitmusRT.
## Contents

1 Introduction .......................... 7
   1.1 Introduction .......................... 7
   1.2 Research question .................. 8
   1.3 Paper outline ...................... 8

2 Background .......................... 9
   2.1 Real-time systems .................. 9
   2.2 Litmus\textsuperscript{RT} ............... 9
   2.3 SysRT ........................... 9
   2.4 Schedulers ........................ 10
   2.5 Scheduling algorithms ............... 11
   2.6 Dahl’s effect ...................... 12
   2.7 Criticality and priority .......... 13
   2.8 Criticality levels ................ 13
   2.9 Periodic task model .............. 14

3 Related work ........................ 15
   3.1 Single-core mixed criticality scheduling .............. 15
   3.2 Multi-core real-time scheduling ................. 15
   3.3 Multi-core mixed criticality scheduling .......... 16

4 Implementation ..................... 17
   4.1 Litmus\textsuperscript{RT} ................. 17
      4.1.1 Litmus\textsuperscript{RT} native schedulers ......... 17
      4.1.2 Litmus\textsuperscript{RT} non-native schedulers ....... 19
   4.2 SysRT ........................... 19

5 Experiments .......................... 21
   5.1 Measurements ....................... 21
      5.1.1 Scheduler evaluation .............. 21
      5.1.2 SysRT validation ................. 21
   5.2 Test platform ....................... 22
   5.3 Scheduler evaluation .......... 22
      5.3.1 Hard real-time only taskset ....... 22
      5.3.2 Hard and Soft real-time only taskset .... 24
      5.3.3 Hard and Soft real-time with sporadic tasks ...... 27
   5.4 Validation .......................... 34
      5.4.1 Litmus\textsuperscript{RT} ................. 35
      5.4.2 SysRT .......................... 37
6 Discussion

6.1 Scheduler evaluation in LitmusRT

6.2 Validation
   6.2.1 LitmusRT
   6.2.2 SysRT

6.3 Comparison

6.4 Conclusions

7 Future work
CHAPTER 1

Introduction

1.1 Introduction

The design of modern embedded systems (e.g. smartphones, cars, airplanes) is becoming increasingly complex. On one hand these systems are designed with complex architectures with an increasing number of processing units, sometimes even on a single chip. On the other hand the (sometimes conflicting) requirements for these systems with regard to reliability, cost, weight, occupied space, efficiency, energy consumption and heat generation are becoming more strict.

Many of these embedded systems are real-time systems and as such the tasks they execute have strict deadlines. Consider for example the flight controls and instruments in an airplane, or the braking system in a car. However, it may be the case that not all tasks within one embedded system are of the same level of criticality, an airplane might also have a multimedia system for the passengers and an air conditioning system, which have a lower level of criticality than the flight controls.

One relatively simple way to implement such a system is to separate all the components with different levels of criticality and have them run on different pieces of hardware. However, this approach would score relatively low on all the aforementioned requirements except reliability, as well as having very poor scalability. A better option would be to implement multiple components with possibly different levels of criticality on a single piece of multi-core hardware. The challenge is then to effectively schedule these tasks making sure on one hand that safety- and mission-critical (high criticality) tasks such as the flight controls always meet their deadlines while on the other hand the non-critical (low criticality) tasks get the best possible performance.

Before implementing any such embedded system, it should first be thoroughly tested to determine whether or not the system and its scheduler are reliable and produce the desired results. These tests can be conducted in a number of ways: Jun Xiao proposes a real-time operating system simulator written in SystemC called SysRT. SysRT can simulate scheduling algorithms and mixed criticality workloads on different hardware configurations. A second way to perform tests is by using a real-time operating system. One suitable option for a real-time operating system is the Litmus\textsuperscript{RT} kernel patch for the Linux kernel \cite{5}. This patch is used to create a real-time Linux operating system, which we will from now on refer to simply as Litmus\textsuperscript{RT}. The Litmus\textsuperscript{RT} project also provides a userspace library, \texttt{liblitmus}, which can be used to create real-time tasks in Litmus\textsuperscript{RT}.
1.2 Research question

In this paper we will evaluate the effectiveness of several different scheduling algorithms when dealing with a mixed criticality workload on a multi-core system. This way we aim to create insight into how different schedulers perform on such a workload and what their advantages and disadvantages are. Our aim is not to find or propose a better way to schedule mixed criticality workloads.

Furthermore, we wish to compare the results of the aforementioned real-time operating system simulator, SysRT, to those of a Litmus$^{RT}$ patched real-time operating system. By doing so we want to determine how well SysRT represents an actual real-time system and thus how useful it is for testing real-time systems.

1.3 Paper outline

We start off by explaining some necessary background information in Chapter 2. Real-time systems, Litmus$^{RT}$ SysRT, schedulers and mixed criticality are explained in this chapter. We then explain the scientific context in which our research is placed in Chapter 3. In Chapter 4 we cover our implementation in detail for both Litmus$^{RT}$ and SysRT. Our experiments and their results are covered in Chapter 5 after which we give our discussion and conclusions in Chapter 6. Finally, we finish with suggestions for future work in chapter 7.
CHAPTER 2

Background

2.1 Real-time systems

Central to this paper are real-time systems; systems that can correctly execute real-time tasks or processes. A real-time task is a task where not only the correct completion of the task matters, but also the time in which the task is completed [4, 6]. These tasks have so called deadlines at which the task has to be completed - if it is not finished before the deadline, the execution of the task has failed. Consider for instance the airbag controller in a car or a video stream. If the airbag does not deploy within a certain time, the deployment has failed as the driver will have already hit their head on the steering wheel. Similarly, if frames in a video stream are rendered after their deadline, the video has already passed the moment at which the frame should have been displayed, thus displaying it no longer has any use.

A real-time system’s main focus is therefore meeting its deadlines rather than maximising throughput. This is the main difference to non-real-time systems, where in many cases the main focus is indeed maximising throughput.

2.2 LitmusRT

LitmusRT is a kernel patch that can be applied to the Linux kernel, creating a Linux kernel with real-time capabilities [5]. In addition to applying the patch, the kernel configuration also requires some specific changes before the LitmusRT patched kernel is able to boot correctly. LitmusRT provides a sample configuration file where most of the necessary configurations are already applied, only a few things such as enabling in-kernel preemption and disabling hyper-threading and central processing unit (CPU) frequency scaling still have to be configured.

In addition to the kernel patch, LitmusRT also provides a user space library, liblitmus and a set of tracing tools, feather trace. Liblitmus provides an API through which the programmer can communicate with the LitmusRT kernel; it allows the programmer to define a real-time process and communicate its existence and parameters to the kernel. The feather trace tools are used to trace how processes are scheduled and what overheads occur. It is designed to cause very little to no performance impact in order to not influence any results by tracing them.

2.3 SysRT

SysRT is a real-time system simulator that can simulate periodic tasksets on various architectures. SysRT consists of three parts: an application layer, a kernel layer and an architecture layer as shown in figure 2.1.
The application layer provides the user with a way to create a set of tasks that they want to run a simulation for. A process can either be a single instance (ST) or a periodic task (PT). A PT is simply a task that periodically executes an ST job. In addition to STs and PTs, SysRT can also model Directed Acyclic Graphs (DAGs).

The kernel layer is responsible for process management, resource management, interrupt handling and scheduling decisions. It communicates directly with the application layer to request information about task states and send out scheduling decisions. It also communicates with the architecture layer to handle interrupts and computation time to tasks.

The architecture layer simulates the actual hardware such as processors and memory. It keeps track of what each processor is executing and communicates this info to the kernel layer. It also simulates the migration or context switch overhead when a task is preempted.

SysRT is implemented using SystemC, a system level design language useful for simulation of real hardware.

2.4 Schedulers

Schedulers are an essential part of any system, as they allocate CPU time to the available waiting processes. There are many different scheduler implementations, each of which has its own characteristics, strengths and weaknesses; we will talk more about scheduling algorithms in the next section.

In a multi-core system, the scheduler can be either a global or a partitioned one. A global scheduler has a single global ready queue and a single scheduler instance that schedules processes on any of the available processor cores in the system. Which core a process is assigned to depends on which core happens to be idle when that process is at the front of the queue. This also means that processes can migrate from one core to another; for example if it was first scheduled on core 0, before being preempted and later scheduled on core 1.
A partitioned scheduler works by letting the operating system, or in some cases the programmer assign a processor core affinity to a process. The process will then only be scheduled on that specific core. To achieve this, processes are first put into a global ready queue after which they are moved to a local ready queue depending on their assigned affinity. A local scheduler then schedules processes from the local ready queue on a single processor core, much like the scheduler in a single core system. In most cases, and indeed also in the experiments presented in this study, the same local scheduler is used for all cores in a system when scheduling in a partitioned manner, but this is not mandatory.

2.5 Scheduling algorithms

In this paper we only consider seven different well known scheduling algorithms, as there are too many slightly varying algorithms to cover every single one of them. We consider two general purpose schedulers (FIFO and RR) and five real-time schedulers (P-FP, G-FP, P-EDF, G-EDF and Pfair).

First in first out (FIFO) is a relatively simple scheduling algorithm; the first process to arrive and enter the queue is also the first to be executed. This results in little overhead, as there are only the bare minimum of context switches. Response times may be high as a process with a long execution time will stall the queue for its full execution time. Process priority depends solely
on the arrival time of the process, as such deadlines are not evaluated which leads to deadline
misses being likely.

The Round Robin (RR) scheduling algorithm works by introducing the concept of time slices.
Each process can only execute consecutively for at most the full duration of a time slice. If a
process isn’t done by the end of a time slice, the process gets preempted and moved to the
back of the queue. The next process in the queue is then allowed to execute until the end of
its time slice and so on. RR introduces a lot of extra overhead compared to FIFO, due to the
more frequent context switches, especially with a small time slice. Deadlines might be frequently
missed when using RR, as the scheduler simply cycles over the waiting processes and does not
evaluate deadlines to determine priority.

The Fixed Priority (FP) scheduling algorithm assigns each process a priority once and then
executes them in a fixed order depending on that priority. The amount of priority levels is
bounded and if multiple processes with the same priority are in the queue, they are executed
as FIFO queues. If priorities are set correctly by the programmer (i.e. real-time processes have
higher priorities than non-real-time processes) the FP scheduler can be effective at meeting
deadlines. The FP scheduler can be implemented with or without preemption. A scheduler that
is implemented with preemption allows a currently executing process to be interrupted if a new
process enters the queue with a higher priority. By default we mean preemptive FP when using
the term FP, we use NP-FP to indicate the non-preemptive version.

The Earliest Deadline First (EDF) algorithm schedules processes according to their deadline.
As its name implies it schedules the process with the least remaining time until their deadline first.
Non-real-time processes without a deadline might take a long time to execute when scheduled
by EDF, as they only execute when there are no real-time processes in the queue, which is some
cases might lead to starvation of non-real-time tasks. EDF can also be implemented with or
without support for preemption. By default we mean preemptive EDF when using the term
EDF, we use NP-EDF to indicate the non-preemptive version.

A Proportionate fair (Pfair) scheduling algorithm divides the total CPU time into small time
slots. Tasks are then also divided into smaller subtasks that fit into the time slots. These small
subtasks are then assigned to slots in such a way that all tasks progress in a proportionately
fair way, i.e. they progress at a similar rate that is proportionate to their total execution cost.
This is achieved by putting introducing windows that contain multiple slots - each task is then
allowed to only execute for one slot per window [2]. Pfair introduces a lot of overhead due to
the many context switches. Depending on context switch cost, Pfair may produce sub-optimal
results due to large overheads.

2.6 Dhall’s effect

In multi-core global scheduling certain tasksets are not feasible even though their utilisation
is well below 1 per processor. If we look at G-EDF for example, and we have a system with
n > 1 cores and n + 1 tasks of which n tasks have a deadline of Da and an execution time of
Ca < 1/2Da, and one task, Tb has a deadline of Da < Db < Da + Ca and an execution cost of
Db − Ca < Cb < Db then in theory this taskset is schedulable as the load is below 1 per processor.
However, when scheduled with G-EDF, task Tb will miss its deadline. If we take n = 2, Da = 8,
Ca = 3, Db = 10 and Cb = 9, we get the following schedule when scheduling with G-EDF.

Figure 2.4: G-EDF schedule for the taskset, displaying Dhall’s effect
Task 3 clearly misses its deadline, but from the schedule diagram below it is clear that all three tasks are able to meet their deadlines if another schedule is used.

Figure 2.5: Example non-G-EDF schedule for the taskset

This shows that using G-EDF to schedule on more than one core could indeed have very poor performance for certain tasksets, and proves that G-EDF is not optimal in multi-core systems. This problem, known as Dhall's effect, is not unique to the G-EDF scheduler, but might occur with any global scheduling algorithm. However, with other schedulers for which a schedule is precomputed offline, the issue can be prevented by changing the schedule. Due to EDF being a dynamic scheduler, its schedule is usually not precomputed and Dhall's effect may occur unexpectedly.

2.7 Criticality and priority

In a mixed criticality (embedded) system, each task has a certain level of criticality. Criticality however is not the same as the priority we mentioned when talking about scheduling algorithms. The criticality refers to how critical the successful execution (in case of real-time tasks, the meeting of deadlines) for that task is, e.g. if both a high(er) criticality task and a low(er) criticality task are in danger of failing execution due to missing their deadlines, the high(er) criticality task should be executed first [6].

In contrast, priority is dynamically assigned by the scheduler (with the exception of static fixed priority scheduling algorithms) and only dictates at what position in the scheduling queue a task is put and thus how soon it is to be executed. In the example above, where two tasks are in danger of missing their deadlines, the scheduler should ideally give the high(er) criticality task a higher priority, so that it is put somewhere in the front of the queue and can execute first in an attempt to still meet the deadline.

It might sound like criticality and priority are equivalent, as a high criticality task generally also get a high priority. But in some cases a lower criticality task might get assigned a higher priority in order to meet a deadline, but only if it does not affect the successful execution of any higher criticality tasks.

2.8 Criticality levels

In this paper we consider three levels of criticality, hard real-time, soft real-time and non-real-time or best effort. A hard real-time task is in most cases a safety or mission-critical task; this means that if the task does not meet its deadlines the consequences can be severe. An airbag controller or flight instruments are good examples of a hard real-time safety-critical task. Due to their critical nature, these tasks are subject to very strict requirements when it comes to program code [3] and thus are not allowed to have unexpected variation in their execution time.

A soft real-time task is a non-safety-critical task, but still a real-time task meaning that it has a deadline. However, if deadlines are not met the consequences are only minor. A good example is a video stream, if deadlines are not met the video stutters, which is undesirable but tolerable to a certain degree. Soft real-time tasks are typically not subject to strict requirements on program code and may have a greatly varying execution time.
Non-real-time tasks are neither real-time nor safety- or mission-critical. The task has no deadlines but instead the total execution time (response time) should be minimized where possible. A good example is the air conditioning system, if the user inputs a new, they would like to feel the result as quickly as possible.

### 2.9 Periodic task model

In the experiments performed for this study we use a periodic task model for the creation of our taskset. A periodic task model consists of real-time tasks $\tau_i$ that have a period $T_i$, relative deadline $D_i$, and execution cost, $C_i$ where $C_i < D_i$ for task $i$ and $\tau_i = (C_i, D_i, T_i)$ for tasks $i = 1, \ldots, n$. Starting from $t_0$, for each task $\tau_i$ a job with execution cost $C_i$ is released at the start of each period $t_n$ (where $n$ is the $n^{th}$ period of $\tau_i$) and must complete by $t_n + D_i$. In our experiments for every task $\tau_i$ the relative deadline is equal to the period $D_i = T_i$. If the periods of all tasks are equal, the periodic task model is called harmonic [7].
CHAPTER 3

Related work

3.1 Single-core mixed criticality scheduling

Previous research has been done on scheduling mixed criticality real-time workloads, mainly for single-core systems. Vestal established the foundation for many other works focusing on single-core mixed criticality scheduling. In his paper Vestal uses period transformation and applies Audsley’s priority assignment algorithm to mixed criticality workloads to improve schedulability of tasksets. Period transformation splits a higher criticality task into subtasks with a smaller period, increasing its priority. Audsley’s algorithm attempts to find a priority assignment under which the given taskset is schedulable [10]. Another approach is to use a dynamic scheduler to schedule mixed criticality workloads on a single-core system, as done by Park and Kim [8] with their dual criticality CBEDF scheduler. Their approach is to let high criticality jobs produce slack that can be used by low criticality jobs. Slack is the processing time that is not used by high criticality jobs, either because not 100 percent of the available time was filled with high criticality jobs (empty slack), or because high criticality jobs executed for less than their allocated time budget (remaining slack) [8]. The budget for high criticality tasks is usually a generously taken worst case execution time (WCET) [7, 10], therefore the amount of slack generated is often ample for the execution of low criticality tasks. CBEDF works by making a schedule offline (not at runtime) to determine where empty slack occurs, and tries to optimize the locations of empty slack. During runtime it then checks if the current time is allocated time or slack, if it is slack a low criticality job is executed, if not then the high criticality job that was allocated to this time is executed [8].

3.2 Multi-core real-time scheduling

Research has also been done on scheduling (hard) real-time workloads on multi-core systems, which adds a lot of difficulty to the scheduling problem in comparison to single-core systems. Two main approaches have been researched; partitioned and global scheduling. Due to Dhall’s effect [6] partitioned scheduling was initially viewed as more efficient and favoured over global scheduling. Aside from Dhall’s effect, partitioned scheduling had the major advantage that after the tasks have been assigned to processors, scheduling them is no different from scheduling (multiple) single-core systems. Another advantage is that if a task overruns its budget, only tasks on the same processor are affected. A partitioned scheduling approach also does not have any processor migrations and uses multiple single-processor run-queues, both of which reduce the overall scheduler overhead. However, the major problem with partitioned scheduling is finding the optimal task allocation, which is an NP-Hard problem [6].

Global scheduling has the advantage that no such task allocation has to be made. A global scheduling approach is also better at utilising slack, as any task can utilise slack on any processor. Preemptions occur less frequently than in partitioned scheduling as well, only when no processors are idle will a task have to be preempted. A major downside of global scheduling approaches
is that it was a completely new field, and much of the knowledge acquired in single-processor scheduling techniques was not applicable.

3.3 Multi-core mixed criticality scheduling

Only very little research has been conducted on combining these two research subjects, scheduling mixed criticality workloads on multi-core systems. This issue was first explored by Anderson et al. [1], they propose a slack based multi-core scheduling architecture that is suitable for mixed criticality tasksets and they use LitmusRT to test their implementation. Their implementation uses the five criticality levels as specified in RTCA DO-178B [1]. For each level a container is created which handles intra container scheduling for tasks of the corresponding criticality level. Different containers may also have different scheduling algorithms; they choose cyclic executive for the highest two criticality containers and use EDF for lower criticality ones. Each container is allocated a certain execution budget and an allocation window, because they only consider a harmonic taskset, the allocation window is simply equal to the task periods [1]. Each task is then given multiple execution requirements (WCETs), one for each criticality level ($L_c$) equal to or lower than the $L_c$ of the task (i.e. in a system with criticality levels $L_c = A, B, C, D, E$, a task with $L_c = C$ will have a WCET for levels $L_c = C, D, E$). When scheduling a task of $L_c = C$ the (smaller) WCET values for $L_c = C$ are used and the cumulative utilisation of all tasks with $L_c \geq C$ cannot exceed 1.0. High criticality tasks are usually given very generous WCETs, which means that they will not consume their full budget most of the time. If the execution time of a high criticality task ($L_c = A$) stays under the WCET that it was given for $L_c = B$, then tasks with $L_c = B$ are also able to successfully execute, using the slack generated by tasks of $L_c = A$. This means that lower criticality tasks cannot affect the execution of higher criticality tasks. However, if a higher criticality task exceeds its $L_c = D$ WCET, tasks of $L_c > D$ might miss their deadlines, which means that the lower the level of criticality, the less assurance there is that the task will meet its deadlines.

This approach is extended by Mollison et al. [7], by adding slack redistribution rules that were left as future work by Anderson et al. In their extension they add slack shifting to reduce the overall tardiness of the system [7]. Each time a higher criticality job finishes execution, the remaining slack is allocated to a lower criticality task using the WCET of that task for its own criticality level. If this task then executes for less than its WCET, it is left in the system as a ghost job, and when the scheduler for that criticality level is invoked it suspends until a job of the same criticality level is available.

In our paper we do not use WCET estimates, as in our implementation we know exactly what the execution cost of a task is. For that reason the slack based approach used by Anderson et al and Mollison et al is not applicable to our experiments.
4.1 Litmus\textsuperscript{RT}

Our Litmus\textsuperscript{RT} taskset implementation consists of two different parts; one of which implements a taskset for the schedulers that are implemented in Litmus\textsuperscript{RT} natively, and another for three schedulers that are not natively implemented. For the Litmus\textsuperscript{RT} implemented schedulers we have the ability to employ \textit{liblitmus}, for the others we use POSIX Thread (Pthread) attributes instead.

4.1.1 Litmus\textsuperscript{RT} native schedulers

As briefly stated before, we use and heavily rely on \textit{liblitmus} for our implementation of the Litmus\textsuperscript{RT} native schedulers, Pfair, P-EDF, G-EDF and P-FP. \textit{Liblitmus} allows us to put a process in real-time mode and give it (among other things) a period, deadline and class (soft or hard real-time).

To create our mixed workload we use a set of periodic tasks which can be configured on an individual level, each task requires at least a relative deadline, execution cost and real-time class. The period is always equal to the relative deadline in our implementation, hence it does not need to be given separately. Optionally, up to 3 probabilities and execution times can be given for soft real-time tasks, which will use a probability mass function to determine the execution cost for each job simulating less predictable execution times. In addition to the periodic tasks, parameters can also be given for individual sporadic tasks.

After setting all task parameters a new thread is started with the highest priority. This thread is in charge of generating a job for each task with an execution time equal to the user defined execution time. The job generation works by executing a loop and checking how much time has passed in each iteration. If that time is equal to the desired execution cost, the job generation will stop and save the amount of iterations needed for a job of that specific execution cost. This iteration count, \(n\), is used when executing the actual jobs during the experiment.

\textbf{Pseudocode 1} Job generation

\begin{verbatim}
1: function job_gen(desired)
2:     start ← gettime()
3:     n ← 0
4:     while True do
5:         current ← gettime()
6:         elapsed ← current − start
7:         n ← n + 1
8:         if elapsed ≥ desired then
9:             return n
\end{verbatim}
After the job generations are complete, threads are started for job generation and for each periodic task. These threads then proceed to initialise and pass their parameters to the LitmusRT kernel, after which they wait for synchronous task release, signalled by the user through liblitmus. Upon release, the threads start by getting the current time that is to be used for checking if deadlines are missed. The first thread to execute locks a mutex, flips a boolean value and writes the current time to a global variable before unlocking again. Each subsequent thread then sees the flipped boolean and only reads the time and puts it into a local variable.

**Pseudocode 2** Period start synchronisation

1: lock mutex
2: if set then
   3: start ← globalstart
4: else
   5: start ← gettime()
   6: globalstart ← start
   7: set ← True
8: unlock mutex

After the start time has been saved, each thread starts executing its job; this job is a function equivalent to the job generation function and runs for the amount of iterations found during job generation. The job function differs slightly from the generation function but these changes have been carefully made so they do not affect the execution time. The most important change is that the job checks for the occurrence of deadline misses using the start time we found earlier and an offset based on how many periods have passed since the start - if the deadline is missed the job ends and returns failure. The thread then either records the miss and continues with the execution of the next job, or exits altogether depending on whether it was a hard or soft deadline miss. When the job doesn’t miss its deadline, success is returned and the thread simply waits for the start of the next period.

**Pseudocode 3** Job execution

1: function job(n, deadline, start, offset)
2:   i ← 0
3:   while i < n do
4:     current ← gettime()
5:     elapsed ← current − (start + offset)
6:     i ← i + 1
7:     if elapsed ≥ deadline then
8:        return failure
9:     return success

In addition to the periodic real-time tasks, a thread is created for the generation of sporadic tasks. This is also a (soft) real-time thread, but its execution cost is so small that it does not have any impact on the execution of the actual task set. Instead of executing a job function at the start of each period, this function generates jobs with a variable period (in a user defined range) using random number generation. During each period, the generator checks if a new job should be launched using a counter. If no new job should be launched, the counter is incremented, otherwise the counter is reset and a new counter is generated before creating a new thread for the sporadic task. This sporadic task thread then executes a single job instance and records the time it takes to complete that job (the response time) using a linked list.

Each periodic thread runs a certain amount of periods depending on the runtime and its period length (runtime/period = num_periods) before exiting and joining to the main thread. The sporadic task generation thread gets an exit signal after all the periodic threads have ended. The main thread then gathers the results such as total deadline misses and the response times of sporadic tasks and prints them before terminating the program.
Pseudocode 4 Sporadic task generation

1: function GEN_SPORADIC(min_period, max_period, execution_cost)
2:   current ← 0
3:   counter ← random(range(min_period, max_period))
4: while run do
5:   if current ≥ counter then
6:     current ← 0
7:     counter ← random(range(min_period, max_period))
8:     sporadic_task(execution_cost)
9:   else
10:      current ← current + 1
11:     wait_next_period()
12: return success

Pseudocode 5 Sporadic task execution

1: function SPORADIC_TASK(execution_cost)
2:   start ← gettime()
3:   job(execution_cost)
4:   end ← gettime()
5:   response ← end – start
6:   linkedlist_add(response)
7: return success

4.1.2 LitmusRT non-native schedulers

To implement our taskset for the schedulers that are not implemented by LitmusRT, G-FP, FIFO and RR, we make more extensive use of the pthread library as well as some three functions written by Dr. Luca Abeni [9]. This implementation remains much the same as the liblitmus implementation, however we had to substitute all functions we used from liblitmus with something else.

Setting the scheduler and priorities of threads was accomplished by setting pthread attributes for each thread. We set PTHREAD_EXPLICIT_SCHED and set the scheduler to FIFO or RR while giving all threads the same priority to schedule them with FIFO and RR. For G-FP we set different priorities for all the threads instead.

To facilitate synchronous release of periodic tasks we used a pthread barrier which will allow all threads to continue as soon as the last thread reaches the barrier. This means we now had to know beforehand how many threads are going to be released, but this was not a problem.

For the implementation of periodic tasks and waiting for the next period we used three functions written by Dr. Abeni [9]; the first function start_periodic_timer creates a periodic task struct with the provided offset and period in microseconds. We can then use this struct to wait for the next period using the wait_next_activation function, which in turn calls the timespec_add_us function which waits for exactly the remaining time until the start of the next period.

4.2 SysRT

Implementing our workload in SysRT is relatively simple compared to LitmusRT; we first need to set the smallest time unit for the simulation, and then define either only a global scheduler or also multiple local schedulers for the kernel, depending on the user’s needs.

Pseudocode 6 Defining global scheduler

1: global_sched ← EDF
2: p_kernel ← kernel(global_sched, num_cores, label)
Pseudocode 7 Defining partitioned schedulers

1: \( \text{global sched} \leftarrow \text{FP} \)
2: \( \text{scheduler1} \leftarrow \text{EDF} \)
3: \( \text{scheduler2} \leftarrow \text{PFair} \)
4: \( p\_\text{kernel} \leftarrow \text{kernel(\text{global sched, num cores, label})} \)
5: \( p\_\text{kernel}.\text{set scheduler}(\text{core, scheduler1}) \)
6: \( p\_\text{kernel}.\text{set scheduler}(\text{core, scheduler2}) \)

Next a taskset has to be created, this is done by first creating a task period, relative deadline, offset and priority after which we add an execution cost by creating pseudo instructions for the task. We then define what should happen upon a deadline miss for this task (and give it a processor affinity if using a partitioned scheduler) before finally passing the task to the kernel.

Pseudocode 8 Defining a taskset

1: \( t1 \leftarrow \text{task(\text{period, deadline, offset, priority})} \)
2: \( t1.\text{code}(\text{execute(20), lock(L1), execute(10), unlock(L1)}) \) \( \triangleright \) Create a job that executes for 20, requests resource L1, executes for 10 when L1 is granted and finally frees L1 again
3: \( t1.\text{miss action}(\text{terminate}) \)
4: \( t1.\text{affinity}(0) \)
5: \( \text{kernel.add(t1)} \)
6: \( t2 \leftarrow \text{task(\text{period, deadline, offset, priority})} \)
7: \( t2.\text{code}(\text{execute(50), lock(L1), execute(20), unlock(L1)}) \)
8: \( t2.\text{miss action}(\text{kill}) \)
9: \( t2.\text{affinity}(1) \)
10: \( \text{kernel.add(t2)} \)

To define a variable period or execution cost, instead of passing a single, static value, we pass a \textit{UniformVar} with the desired range for the value. Once all tasks are passed to the kernel, the simulation start can be called with the desired runtime. During the simulation, SysRT will write data such as deadline misses, response times, context switches and migrations to a file.
CHAPTER 5

Experiments

5.1 Measurements

5.1.1 Scheduler evaluation

The performance of a scheduler cannot be measured in the same way for every scenario, as different scenarios have different requirements. A scheduler that performs excellent in one scenario can perform poorly when presented with a different scenario. For example, the focus of Linux’s Completely Fair Scheduler lies mostly with maximising throughput. Whereas for a real-time system this focus lies predominantly with making sure that tasks meet their deadlines rather than maximising throughput.

In our scheduler evaluation experiments we measure the maximum feasible load per processor for our specific taskset in Litmus$^\text{RT}$. We find this maximum by running the experiment each time with a slightly higher load until a deadline miss occurs. Once we determine this maximum we measure the total time spent on context switches and migrations, and the average response time of non-real-time tasks. Context switch and migration cost is measured in clock cycles using the feather trace tools (ft tools) provided by the Litmus$^\text{RT}$ project, specifically using ftcat and ft2csv. To convert clock cycles to milliseconds, we use the clockrate of the CPU in megahertz (2200Mhz) in the following formula: $ms = \frac{cycles}{clockrate}/1000$. The ft tools can only be used for schedulers natively implemented by Litmus$^\text{RT}$, therefore we did not measure context switch and migration cost for the G-FP, FIFO and RR schedulers. The average response time is measured in the sporadic task thread itself.

For these experiments we were not able to measure the soft deadline miss ratio as the schedulers in Litmus$^\text{RT}$ do not prioritise hard real-time tasks over soft real-time ones. This means that as we increase the workload past the scheduler’s maximum load, hard deadline misses will occur as well as soft ones. Only with a carefully designed taskset is it possible to guarantee hard real-time deadlines are met while soft real-time deadlines are being missed.

We measure the maximum load and context switch cost with our taskset first for a hard real-time only workload, then for a hard and soft real-time combined workload and finally for a hard real-time, soft real-time and non-real-time workload. For the latter workload we also measure the average response time of sporadic tasks allowing us to see the difference in performance that a mixed criticality workload causes.

5.1.2 SysRT validation

To validate the SysRT simulations we use a carefully created taskset (discussed in more detail later) and run experiments with it in both SysRT and Litmus$^\text{RT}$. From these experiments we gather the soft deadline miss ratio, migration and context switch cost and response times. This carefully designed taskset allowed us to reliably measure the deadline miss ratio of only soft real-time tasks.
5.2 Test platform

The first Litmus\textsuperscript{RT} experiments were executed on a Lenovo G50 notebook with i5-5200U 2.2Ghz duo core processor and 4 GiB of RAM. Hyper threading and CPU frequency scaling were disabled for all tests to prevent any unexpected influences. Ubuntu 14.04 with Linux kernel version 4.1.3 is used as the base operating system on which we installed Litmus\textsuperscript{RT} and SysRT. Both the SysRT and Litmus\textsuperscript{RT} part of the SysRT validation were performed on a quad-core iMac running the same version of Litmus\textsuperscript{RT}.

5.3 Scheduler evaluation

5.3.1 Hard real-time only taskset

Our hard real-time only taskset consists of a total of 5 hard real-time tasks per processing core, three of which we left unchanged for all experiments and two of which we varied to find the highest feasible load with this taskset. In the tables below we present the exact configuration of the taskset that we ran the experiments with.

![Figure 5.1: Execution cost in ms ($C_i$), deadline in ms ($D_i$), period in ms ($T_i$) and utilisation ($U_i$) of the static component of the taskset per core, totaling 0.425 utilisation per core](image)

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>125</td>
<td>1000</td>
<td>1000</td>
<td>0.125</td>
</tr>
<tr>
<td>Task 2</td>
<td>105</td>
<td>1050</td>
<td>1050</td>
<td>0.100</td>
</tr>
<tr>
<td>Task 3</td>
<td>350</td>
<td>1750</td>
<td>1750</td>
<td>0.200</td>
</tr>
</tbody>
</table>

![Figure 5.2: Execution cost in ms ($C_i$), deadline in ms ($D_i$), period in ms ($T_i$) and utilisation ($U_i$) of the variable part of the taskset per core on 1 and 2 cores for Pfair (a), P-EDF (b), G-EDF (c), P-FP (d), G-FP (e), FIFO (f) and RR (g)](image)

(a) Pfair

1 Core

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>210</td>
<td>1200</td>
<td>1200</td>
<td>0.175</td>
</tr>
<tr>
<td>Task 5</td>
<td>393</td>
<td>1000</td>
<td>1000</td>
<td>0.393</td>
</tr>
</tbody>
</table>

2 Cores

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>210</td>
<td>1200</td>
<td>1200</td>
<td>0.175</td>
</tr>
<tr>
<td>Task 5</td>
<td>386</td>
<td>1000</td>
<td>1000</td>
<td>0.386</td>
</tr>
</tbody>
</table>

(b) P-EDF

1 Core

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>210</td>
<td>1200</td>
<td>1200</td>
<td>0.175</td>
</tr>
<tr>
<td>Task 5</td>
<td>399</td>
<td>1000</td>
<td>1000</td>
<td>0.399</td>
</tr>
</tbody>
</table>

2 Cores

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>210</td>
<td>1200</td>
<td>1200</td>
<td>0.175</td>
</tr>
<tr>
<td>Task 5</td>
<td>399</td>
<td>1000</td>
<td>1000</td>
<td>0.399</td>
</tr>
</tbody>
</table>
The above taskset gives us the maximum utilisation values presented in the graph below.
The combined context switch and migration cost for this taskset is presented in the graph below.

**Figure 5.4:** Context switch and migration combined cost per real-time scheduler for the taskset on 1 and 2 cores with a 10 minute runtime, using a logarithmic scale

### 5.3.2 Hard and Soft real-time only taskset

For the following experiments we introduce soft real-time tasks into the taskset. The soft real-time tasks use a probability mass function to determine the execution cost; we consider the worst case execution time (WCET), average case execution time (ACET) and best case execution time (BCET) of these tasks. In this set of experiments we only vary the soft real-time tasks’ utilisation and leave the hard real-time task utilisation at 0.5 per core for all experiments. The
Figure 5.5: Execution cost in ms ($C_i$), deadline in ms ($D_i$), period in ms ($T_i$) and utilisation ($U_i$) of the static hard real-time component of the taskset per core, totaling 0.5 utilisation per core

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>125</td>
<td>1000</td>
<td>1000</td>
<td>0.125</td>
</tr>
<tr>
<td>Task 2</td>
<td>350</td>
<td>1750</td>
<td>1750</td>
<td>0.200</td>
</tr>
<tr>
<td>Task 3</td>
<td>210</td>
<td>1200</td>
<td>1200</td>
<td>0.175</td>
</tr>
</tbody>
</table>

Figure 5.6: Execution cost in ms ($C_i$), deadline in ms ($D_i$), period in ms ($T_i$) and utilisation ($U_i$) for the ACET ($AC_i$), BCET ($BC_i$) and WCET ($WC_i$) of the soft real-time component of the taskset per core on 1 and 2 cores for Pfair (a), P-EDF (b), G-EDF (c), P-FP (d), FIFO (e) and RR (f)

(a) Pfair

<table>
<thead>
<tr>
<th></th>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>195</td>
<td>140</td>
<td>240</td>
<td>1000</td>
<td>1000</td>
<td>0.195</td>
<td>0.140</td>
<td>0.240</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>195</td>
<td>140</td>
<td>240</td>
<td>1000</td>
<td>1000</td>
<td>0.195</td>
<td>0.140</td>
<td>0.240</td>
</tr>
</tbody>
</table>

(b) P-EDF

<table>
<thead>
<tr>
<th></th>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>235</td>
<td>180</td>
<td>280</td>
<td>1000</td>
<td>1000</td>
<td>0.235</td>
<td>0.180</td>
<td>0.280</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>230</td>
<td>175</td>
<td>275</td>
<td>1000</td>
<td>1000</td>
<td>0.230</td>
<td>0.175</td>
<td>0.275</td>
</tr>
</tbody>
</table>

(c) G-EDF

<table>
<thead>
<tr>
<th></th>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>235</td>
<td>180</td>
<td>280</td>
<td>1000</td>
<td>1000</td>
<td>0.235</td>
<td>0.180</td>
<td>0.280</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>235</td>
<td>180</td>
<td>280</td>
<td>1000</td>
<td>1000</td>
<td>0.235</td>
<td>0.180</td>
<td>0.280</td>
</tr>
</tbody>
</table>
The values for G-FP are missing in the above tables as we were unable to produce a feasible taskset for the G-FP scheduler by only varying the soft real-time tasks. The above taskset leads to the soft real-time utilisation values presented in the graph below; using the BCET, ACET and WCET.
Figure 5.7: Maximum feasible soft real-time utilisation per processor for the
taskset, based on the ACET (a), BCET (b) and WCET (c) of the soft real-time
tasks on 1 and 2 cores with a 10 minute runtime

The combined context switch and migration cost for this taskset is presented in the graph below.

Figure 5.8: Context switch and migration combined cost per real-time scheduler
for the taskset on 1 and 2 cores with a 10 minute runtime, using a logarithmic
scale

5.3.3 Hard and Soft real-time with sporadic tasks
For our last experiments we introduced three different sporadic non-real-time tasks into the
workload which will be discussed in more detail below. We again use a static hard real-time
utilisation of 0.5 per processor and vary only the soft real-time component of the taskset. The
tables below present the exact configuration of the taskset.
Figure 5.9: Execution cost in ms ($C_i$), deadline in ms ($D_i$), period in ms ($T_i$) and utilisation ($U_i$) of the static hard real-time component of the taskset per core, totaling 0.5 utilisation per core

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>125</td>
<td>1000</td>
<td>1000</td>
<td>0.125</td>
</tr>
<tr>
<td>Task 2</td>
<td>350</td>
<td>1750</td>
<td>1750</td>
<td>0.200</td>
</tr>
<tr>
<td>Task 3</td>
<td>210</td>
<td>1200</td>
<td>1200</td>
<td>0.175</td>
</tr>
</tbody>
</table>

Figure 5.10: Execution cost in ms ($C_i$), deadline in ms ($D_i$), period in ms ($T_i$) and utilisation ($U_i$) for the ACET ($^{AC}_i$), BCET ($^{BC}_i$) and WCET ($^{WC}_i$) of the soft real-time component of the taskset per core on 1 and 2 cores for Pfair (a), P-EDF (b), G-EDF (c), P-FP (d), FIFO (e) and RR (f)

(a) Pfair

1 Core

<table>
<thead>
<tr>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>165</td>
<td>110</td>
<td>210</td>
<td>1000</td>
<td>0.165</td>
<td>0.110</td>
<td>0.210</td>
</tr>
</tbody>
</table>

2 Cores

<table>
<thead>
<tr>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>175</td>
<td>120</td>
<td>220</td>
<td>1000</td>
<td>0.175</td>
<td>0.120</td>
<td>0.220</td>
</tr>
</tbody>
</table>

(b) P-EDF

1 Core

<table>
<thead>
<tr>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>235</td>
<td>180</td>
<td>280</td>
<td>1000</td>
<td>0.235</td>
<td>0.180</td>
<td>0.280</td>
</tr>
</tbody>
</table>

2 Cores

<table>
<thead>
<tr>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>205</td>
<td>150</td>
<td>250</td>
<td>1000</td>
<td>0.205</td>
<td>0.150</td>
<td>0.250</td>
</tr>
</tbody>
</table>

(c) G-EDF

1 Core

<table>
<thead>
<tr>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>235</td>
<td>180</td>
<td>280</td>
<td>1000</td>
<td>0.235</td>
<td>0.180</td>
<td>0.280</td>
</tr>
</tbody>
</table>

2 Cores

<table>
<thead>
<tr>
<th>$C_i^{AC}$</th>
<th>$C_i^{BC}$</th>
<th>$C_i^{WC}$</th>
<th>$D_i$</th>
<th>$T_i$</th>
<th>$U_i^{AC}$</th>
<th>$U_i^{BC}$</th>
<th>$U_i^{WC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4</td>
<td>228</td>
<td>200</td>
<td>240</td>
<td>800</td>
<td>0.286</td>
<td>0.250</td>
<td>0.300</td>
</tr>
<tr>
<td>Task 5</td>
<td>225</td>
<td>170</td>
<td>270</td>
<td>1000</td>
<td>0.225</td>
<td>0.170</td>
<td>0.270</td>
</tr>
</tbody>
</table>
The three sporadic tasks we introduced in this set of experiments are presented below. Their execution cost is fixed while the period is within a range.

Figure 5.11: Execution cost in ms ($C_i$), period range in ms ($T_i$) and utilisation range ($U_i$) of the sporadic non-real-time component of the taskset

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$T_i$</th>
<th>$U_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>100</td>
<td>[1000, 5000]</td>
<td>[0.020, 0.100]</td>
</tr>
<tr>
<td>Task 2</td>
<td>300</td>
<td>[6000, 9000]</td>
<td>[0.033, 0.050]</td>
</tr>
<tr>
<td>Task 3</td>
<td>750</td>
<td>[5000, 15000]</td>
<td>[0.050, 0.150]</td>
</tr>
</tbody>
</table>

The above taskset leads to the soft real-time utilisation values presented in the graph below, using the BCET, ACET and WCET.
Figure 5.12: Maximum feasible soft real-time utilisation per processor for the taskset based on the ACET (a), BCET (b) and WCET (c) of the soft real-time tasks on 1 and 2 cores with a 10 minute runtime.

In the figures below the schedule diagrams, generated using the ft tools, are shown for the LitmusRT native schedulers.

Figure 5.13: Schedule diagram of the first 50ms with Pfair on 2 cores.
Figure 5.14: Schedule diagram of the first 50ms with P-EDF on 2 cores

Figure 5.15: Schedule diagram of the first 50ms with G-EDF on 2 cores

Figure 5.16: Schedule diagram of the first 50ms with P-FP on 2 cores

For these experiments we also measured the average response time of the sporadic tasks, shown in the graph below for sporadic tasks 1, 2 and 3.
Figure 5.17: Average response time of sporadic tasks t1, t2 and t3 for Pfair, P-EDF, G-EDF, P-FP, G-FP, FIFO and RR on 1 and 2 cores with a 10 minute runtime

The combined context switch and migration cost for this taskset is presented in the graph below.

Figure 5.18: Context switch and migration combined cost per real-time scheduler for the taskset on 1 and 2 cores with a 10 minute runtime, using a logarithmic scale

In the graphs below display the cost per context switch or migration for the first 500 occurrences for the Litmus\textsuperscript{RT} native schedulers.
Figure 5.19: Cost per context switch or migration for the first 500 context switches using Pfair on 2 cores

Figure 5.20: Cost per context switch or migration for the first 500 context switches using P-EDF on 2 cores
5.4 Validation

The taskset used to validate SysRT is defined in the table below.
Figure 5.23: Task type, execution cost in ms \((C_i)\), period in ms \((T_i)\) and utilisation \((U_i)\) of the taskset used with SysRT

<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>(C_i)</th>
<th>(T_i)</th>
<th>(U_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Hard-RT</td>
<td>20</td>
<td>50</td>
<td>0.400</td>
</tr>
<tr>
<td>Task 2</td>
<td>Hard-RT</td>
<td>30</td>
<td>90</td>
<td>0.333</td>
</tr>
<tr>
<td>Task 3</td>
<td>Hard-RT</td>
<td>50</td>
<td>140</td>
<td>0.357</td>
</tr>
<tr>
<td>Task 4</td>
<td>Soft-RT</td>
<td>30</td>
<td>190</td>
<td>0.157</td>
</tr>
<tr>
<td>Task 5</td>
<td>Soft-RT</td>
<td>80</td>
<td>350</td>
<td>0.228</td>
</tr>
<tr>
<td>Task 6</td>
<td>Soft-RT</td>
<td>170</td>
<td>500</td>
<td>0.340</td>
</tr>
<tr>
<td>Task 7</td>
<td>Soft-RT</td>
<td>[200, 700]</td>
<td>1000</td>
<td>[0.200, 0.700]</td>
</tr>
<tr>
<td>Task 8</td>
<td>Soft-RT</td>
<td>[500, 900]</td>
<td>1300</td>
<td>[0.385, 0.692]</td>
</tr>
<tr>
<td>Task 9</td>
<td>Non-RT</td>
<td>200</td>
<td>[1000, 5000]</td>
<td>[0.040, 0.200]</td>
</tr>
<tr>
<td>Task 10</td>
<td>Non-RT</td>
<td>500</td>
<td>[3000, 9000]</td>
<td>[0.056, 0.167]</td>
</tr>
<tr>
<td>Task 11</td>
<td>Non-RT</td>
<td>1500</td>
<td>[5000, 15000]</td>
<td>[0.100, 0.300]</td>
</tr>
</tbody>
</table>

5.4.1 Litmus\(^{RT}\)

For the Litmus\(^{RT}\) part we ran the previously discussed taskset on the Pfair, P-EDF, G-EDF and P-FP schedulers. The table below displays the distribution of tasks when using partitioned schedulers.

Figure 5.24: Task distribution for partitioned schedulers in Litmus\(^{RT}\) for 3 and 4 cores

<table>
<thead>
<tr>
<th># Cores</th>
<th>Core #</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Cores</td>
<td>Core 1</td>
<td>({\tau_{1}, \tau_{2}, \tau_{9}, \tau_{11}})</td>
</tr>
<tr>
<td></td>
<td>Core 2</td>
<td>({\tau_{3}, \tau_{4}, \tau_{6}})</td>
</tr>
<tr>
<td></td>
<td>Core 3</td>
<td>({\tau_{5}, \tau_{7}, \tau_{8}, \tau_{10}})</td>
</tr>
<tr>
<td>4 Cores</td>
<td>Core 1</td>
<td>({\tau_{1}, \tau_{2}})</td>
</tr>
<tr>
<td></td>
<td>Core 2</td>
<td>({\tau_{3}, \tau_{4}, \tau_{6}})</td>
</tr>
<tr>
<td></td>
<td>Core 3</td>
<td>({\tau_{5}, \tau_{7}, \tau_{8}})</td>
</tr>
<tr>
<td></td>
<td>Core 4</td>
<td>({\tau_{9}, \tau_{10}, \tau_{11}})</td>
</tr>
</tbody>
</table>

The graph below displays the deadline miss ratios for soft real-time tasks for the aforementioned schedulers.
The combined context switch and migration cost in clock cycles is presented in the graph below.

Figure 5.26: Total context switch and migration combined cost for Pfair, P-EDF, G-EDF and P-FP scheduling on 3 and 4 cores

The average response times of the non-real-time tasks, $\tau_9$, $\tau_{10}$ and $\tau_{11}$ are plotted in the graph below.
Figure 5.27: Average response time of sporadic tasks t9, t10 and t11 for Pfair, P-EDF, G-EDF and P-FP scheduling on 3 and 4 cores

5.4.2 SysRT

For the SysRT part we ran the taskset on global EDF, global FP and a heterogeneous Partitioned scheduler. The local schedulers used in the partitioned scheduler are displayed in the table below. The partitioned scheduler was only evaluated for 3, 4 and 5 cores.

Figure 5.28: Configuration of local schedulers for the partitioned scheduler experiments for 3, 4 and 5 cores

<table>
<thead>
<tr>
<th># Cores</th>
<th>Core #</th>
<th>Local Scheduler</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Cores</td>
<td>Core 1</td>
<td>FP</td>
<td>{τ₁, τ₂, τ₉, τ₁₁}</td>
</tr>
<tr>
<td></td>
<td>Core 2</td>
<td>EDF</td>
<td>{τ₃, τ₄, τ₆}</td>
</tr>
<tr>
<td></td>
<td>Core 3</td>
<td>RR</td>
<td>{τ₅, τ₇, τ₈, τ₁₀}</td>
</tr>
<tr>
<td>4 Cores</td>
<td>Core 1</td>
<td>Pfair</td>
<td>{τ₁, τ₂}</td>
</tr>
<tr>
<td></td>
<td>Core 2</td>
<td>FP</td>
<td>{τ₃, τ₄, τ₆}</td>
</tr>
<tr>
<td></td>
<td>Core 3</td>
<td>EDF</td>
<td>{τ₅, τ₇, τ₈}</td>
</tr>
<tr>
<td></td>
<td>Core 4</td>
<td>RR</td>
<td>{τ₉, τ₁₀, τ₁₁}</td>
</tr>
<tr>
<td>5 Cores</td>
<td>Core 1</td>
<td>Pfair</td>
<td>{τ₁, τ₂, τ₅}</td>
</tr>
<tr>
<td></td>
<td>Core 2</td>
<td>FP</td>
<td>{τ₃}</td>
</tr>
<tr>
<td></td>
<td>Core 3</td>
<td>NP-EDF</td>
<td>{τ₄, τ₇}</td>
</tr>
<tr>
<td></td>
<td>Core 4</td>
<td>EDF</td>
<td>{τ₆, τ₈}</td>
</tr>
<tr>
<td></td>
<td>Core 5</td>
<td>RR</td>
<td>{τ₉, τ₁₀, τ₁₁}</td>
</tr>
</tbody>
</table>

The deadline miss ratio for this taskset is presented in the graph below, 6, 7 and 8 cores have been left out as their results were the same as for 5 cores.
The graph below presents the amount of context switches and migrations happening during the simulation.

Figure 5.30: Total number of context switches (CS) and migrations (MG) for Partitioned (PRT), FP and EDF scheduling

The average response times of the non-real-time tasks, $\tau_9$, $\tau_{10}$ and $\tau_{11}$ are plotted in the graph below, values for less than four cores for FP and EDF have been left out due to their large nature.
Figure 5.31: Average response time of sporadic tasks t9, t10 and t11 for Partitioned (PRT), FP and EDF scheduling on 3 to 8 cores
6.1 Scheduler evaluation in Litmus\textsuperscript{RT}

From our experiments we gathered that in all three scenarios Pfair and the two variations of EDF have the best performance, having the highest feasible maximum load for our taskset. The P-FP and G-FP schedulers both performed poorly, the cause being the structure of the taskset. What makes a big different for Fixed Priority schedulers is whether or not the taskset is harmonic. In our case it was not, which can lead to the next iteration of a higher priority task being executed rather than the previous one for a lower priority task. This leads to deadline misses when the lower priority task is very near its deadline. In a harmonic taskset, this issue does not occur.

The generic schedulers FIFO and RR performed well when presented with the hard real-time only taskset, however when we introduce soft real-time tasks with variable execution time the performance drops and when introducing sporadic tasks it drops even further. This can be explained by the fact these schedulers are not real-time schedulers and thus do not prioritise real-time tasks over non-real-time tasks. Therefore real-time tasks are hindered by non-real-time tasks and deadline misses become more likely.

Looking at migrations and context switches; as expected only Pfair has a significantly higher cost than all other real-time schedulers. However it did not significantly affect the performance. In our experiments the overhead caused by Pfair on a 2.2Ghz processor was less than 500ms for a 10 minute run.

In terms of response times Pfair performed slightly better than EDF, but considerably worse than FP, RR and FIFO. The high response times for both EDF and Pfair can be explained by noting that the Litmus\textsuperscript{RT} schedulers do not consider non-real-time tasks in their scheduling decisions. This means that non-real-time tasks are only executed when the processor would otherwise be idle. The lighter workload used for FP, RR and FIFO then explains why response times are lower, as the processor had more idle time.

For the real-time schedulers multi-core performance was usually similar to single-core performance. However it should be noted that our taskset was not affected by Dhall’s effect. If another taskset is chosen, multi-core performance might turn out much lower than single-core performance.

Litmus\textsuperscript{RT} proved to be ill suited as test platform to test mixed criticality workloads, the main reason being that the Litmus\textsuperscript{RT} schedulers do not prioritize hard real-time over soft real-time tasks and it has poor support for running non-real-time tasks alongside real-time tasks. This is problematic when the task hierarchy is of great importance, which is the case when testing mixed criticality systems.

In addition to these problems, we found that there were some severe memory leaks in the \textit{ft tools}. With each run of our experiments we would lose around 500MB of memory which also couldn’t be cleaned up by the operating system, presumably because the \textit{ft tools} run in kernel mode.
6.2 Validation

6.2.1 Litmus$^\text{RT}$

Due to the nature of the taskset used for these experiments, we were able to successfully measure the deadline miss ratio for soft real-time tasks without any hard real-time tasks missing their deadlines. When looking at the deadline miss ratio we see that G-EDF misses almost no deadlines which is what was expected for this taskset. P-EDF and P-FP miss a small amount of deadlines which is explained by inefficient load balancing of tasks across the available cores. Pfair misses the most by far which is likely due to the larger overhead that is inherent to the Pfair scheduling algorithm, as it performs a large amount of context switches by design. Interesting is also that Pfair has a larger deadline miss ratio when we increased the cores to 4, but this can be explained by the random element in our tests; such as the execution cost of soft real-time tasks and the variable period of non-real-time tasks.

Looking at the context switch and migration combined cost, as expected Pfair has the most overhead. For 3 cores both partitioned schedulers have the least overhead as they have no migrations and only switch when a task is completed or preempted for another task on the same core. For 4 cores we see an unexpectedly high overhead for P-FP, which is likely caused by the different task distribution that is used for 4 cores.

Looking at the response times, as expected G-EDF has the highest response times for non-real-time tasks as they are only executed when there are no real-time tasks available. Both partitioned schedulers have a core assigned to either only non-real-time tasks or non-real-time tasks and one other task, reducing the overall response times. Pfair has relatively low response times due to its switching nature which allows non-real-time tasks to execute more frequently.

6.2.2 SysRT

From the SysRT experiments’ deadline miss ratio we can see that EDF has the best performance, suffering no deadline misses even on 2 cores. The partitioned scheduler approach performs considerably worse, which can be explained by inefficient utilisation of all processing cores due to non-optimal load balancing across all cores. Fixed Priority performs the worst by far, the taskset used for SysRT was also a non-harmonic one, which explains the poor performance of the FP scheduler.

If we look at the context switches and migrations, the partitioned scheduler stands out with its high amount of context switches. Looking at the local schedulers used, this result is explained by the usage of Round Robin and Pfair schedulers as local schedulers. EDF and FP show a much lower amount of context switches, which is also what we expected as they both only switch either when a higher priority or earlier deadline task enters the queue (preemption), or when a job finishes execution. Migrations are consistent for all EDF and FP tests, and naturally not present for the partitioned scheduler.

Response times are similar across all three schedulers, the partitioned scheduler has slightly lower response times than EDF and FP, because tasks scheduled on different cores cannot affect each other’s execution in a partitioned system.

6.3 Comparison

Comparing the results for Litmus$^\text{RT}$ and SysRT we see similar trends in performance when comparing maximum utilisation and deadline miss ratios. G-EDF has the lowest deadline miss ratio while partitioned approaches perform slightly worse due to the less efficient utilisation of the available hardware.

Looking at context switches and migrations we see that both Litmus$^\text{RT}$ and SysRT EDF and FP have a similar amount of context switch overhead. Looking at the partitioned approach in SysRT we see an extremely large amount of switches happening due to the inclusion of Pfair as local scheduler, similar to the results for the Pfair scheduler found using Litmus$^\text{RT}$.

When looking at the response times, we see that in SysRT, FP and EDF show similar response times, whereas in Litmus$^\text{RT}$ the response times for EDF are considerably higher than those for
FP. This can be explained by partitioned nature of the FP scheduler in Litmus\textsuperscript{RT} causing more misses but allowing sporadic tasks to execute more quickly.

### 6.4 Conclusions

In our experiments, EDF and Pfair based schedulers performed the best by far. For a multi-core system EDF is however not optimal due to the possible occurrence of Dhall’s effect. Therefore Pfair is the better choice if it uncertain if the taskset used will be subject to Dhall’s effect. In addition, Pfair has the advantage of producing lower response times for non-real-time tasks. The advantage of EDF is the slightly smaller amount of deadline misses, but it comes at the cost of higher response times. The Fixed Priority and non-real-time schedulers perform worse in almost every aspect than Pfair or EDF, and are unsuitable for use with mixed criticality workloads.

When comparing SysRT and Litmus\textsuperscript{RT} we see similar trends in performance and scheduler overheads. Response times differ slightly which is explained by the different scheduler implementations used. We can conclude that the results gathered with SysRT are valid and thus that the SysRT simulator is also valid.
In this paper we only executed our experiments on a single platform and architecture. We would like to expand our work by adding experiments on multiple different platforms and architectures, such as a hexa- or octa-core platform and AMD or ARM architectures instead of only Intel. This way we can see if there are any significant differences in scheduling mixed criticality workloads for different architectures and if the SysRT simulator is also valid for them.

As we stated in the previous chapter, our results were highly dependent on the structure of the taskset; in our experiments we used only one taskset which we modified slightly to change the total load it generates. In future research we would like to perform the experiments for multiple completely differently structured tasksets so we can find in what way exactly the structure of the taskset influences our results.

In this work we could not always reliably measure the deadline miss ratio of soft real-time tasks. When not using a carefully designed taskset, hard deadline misses occur as well when soft ones do due to limitations of the LitmusRT schedulers. It would however be a valuable addition to see how the deadline miss ratio increases as the workload increases on different schedulers, as this gives insight into how well a scheduler recovers from a tolerable deadline miss.
Bibliography


