# Cache Related Pre-emption Delays in Embedded Real-Time Systems

**William Richard Elgon Lunniss**

EngD

University of York

Computer Science

September 2014

# Abstract

*Real-time* systems are subject to stringent deadlines which make their *temporal* behaviour just as important as their *functional* behaviour. In *multi-tasking* real-time systems, the execution time of each task must be determined, and then combined together with information about the scheduling policy to ensure that there are enough resources to schedule all of the tasks. This is usually achieved by performing *timing analysis* on the individual tasks, and then *schedulability analysis* on the system as a whole.

In systems with cache, multiple tasks can share this common resource which can lead to *cache-related pre-emption delays* (CRPD) being introduced. CRPD is the additional cost incurred from resuming a pre-empted task that no longer has the instructions or data it was using in cache, because the pre-empting task(s) evicted them from cache. It is therefore important to be able to account for CRPD when performing schedulability analysis.

This thesis focuses on the effects of CRPD on a single processor system, further expanding our understanding of CRPD and ability to analyse and optimise for it. We present new CRPD analysis for *Earliest Deadline First* (EDF) scheduling that significantly outperforms existing analysis, and then perform the first comparison between *Fixed Priority* (FP) and EDF accounting for CRPD. In this comparison, we explore the effects of CRPD across a wide range of system and taskset parameters. We introduce a new *task layout optimisation* technique that maximises system schedulability via reduced CRPD. Finally, we extend CRPD analysis to *hierarchical* systems, allowing the effects of cache when scheduling multiple independent applications on a single processor to be analysed.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgements

This thesis would not have been possible without the support of a number of people. Firstly and foremost I would like to thank my supervisor Rob Davis for his help and guidance with my research over the course of my EngD. Parts of this thesis originate from collaborative research work and I would especially like to thank Sebastian Altmeyer for his part. We spent many long evenings refining papers for conferences over Skype.

Thirdly I would like to thank my industrial supervisor, Antoine Colin, and everyone at Rapita Systems for providing such a supportive and friendly work environment. Being able to work on alternative problems in-between long periods of research proved very enjoyable.

Finally I would like to thank my fiancée Amelia for her love and support over the course of my EngD.

# Declaration

Except where stated otherwise all of the work contained within this thesis represents the original contribution of the author, and has not been previously submitted for examination. This thesis contains work that has been published in peer-reviewed, conferences, workshops and journals.

The initial response time analysis software framework supporting analysis for FP scheduling was provided by Sebastian Altmeyer. Through the course of this thesis, the author extended this software framework to add support for task layout optimisation, EDF, and hierarchical scheduling, based on the analysis presented in this thesis. Sebastian Altmeyer also provided the case study data used as part of the evaluation. The author wrote the main body of the papers and journal articles, with the additional authors providing help with proof reading draft copies.

Chapter 4 is based on the following publication:
W. Lunniss, S. Altmeyer, C. Maiza, R.I. Davis, "**Integrating Cache Related Pre-emption Delay Analysis into EDF Scheduling**". In proceedings of the 19th IEEE Conference on Real-Time and Embedded Technology and Applications (RTAS), April 9-11th, 2013, pp 75-84.
*Awarded K.M. Scott Prize for Best Student Paper 2013 - Computer Science Department, University of York*

Chapter 5 is based on the following publication:
W. Lunniss, S. Altmeyer, R.I. Davis "**Optimising Task Layout to Increase Schedulability via Reduced Cache Related Pre-emption Delays**". In proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS), Nov 8-9th, 2012, pp 161-170.

Chapter 6 is based on the following publication:
W. Lunniss, S. Altmeyer, R.I. Davis, "**A Comparison between Fixed Priority and EDF Scheduling accounting for Cache Related Pre-emption Delays**". Leibniz Transactions on Embedded Systems, Volume 1, Number 1, April 2014, pp 1-24. DOI: http://dx.doi.org/10.4230/LITES-v001-i001-a001.

Chapter 7 is based on the following publications:
W. Lunniss, S. Altmeyer, G. Lipari, R.I. Davis, "**Accounting for Cache Related Pre-emption Delays in Hierarchical Scheduling**". In proceedings of the 22nd International Conference on Real-Time and Network Systems (RTNS), Oct 8-10th, 2014
*Awarded Outstanding Paper*

W. Lunniss, S. Altmeyer, R.I. Davis, "**Accounting for Cache Related Pre-emption Delays in Hierarchical Scheduling with Local EDF Scheduler**". In proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC), Oct 8-10th, 2014

W. Lunniss, S. Altmeyer, G. Lipari, R.I. Davis, "**Cache Related Pre-emption Delays in Hierarchical Scheduling**". Under submission to Real-Time Systems, Special issue for RTNS 2014

# CHAPTER 1. INTRODUCTION

We are surrounded by e*mbedded* systems contained within larger devices, from medical pacemakers to the engine and control systems in large commercial aircraft. Many of these embedded systems are also *real-time* systems that have specific deadlines that they must meet, and are often required to interact with an outside environment. It is therefore important that these real-time systems meet their temporal requirements, as well as being functionally correct. Real-time systems can be categorised as *soft* and *hard* real-time. A soft real-time system can tolerate a moderate number of deadline misses, at the expense of reduced quality of service, such as in a live video streaming system. In contrast, a deadline miss in a hard real-time system would constitute a failure of the system. Some hard real-time systems are also safety critical systems such that a deadline miss, and thus a system failure, could cause someone physical harm. Most real-time systems are *multi-tasking* systems built up of a number of individual tasks. To verify the temporal behaviour of a multi-tasking system, the execution time of each task must be determined, and then combined together with information about the scheduling policy to ensure that there are enough resources to run all of the tasks that make up the system. This is usually achieved by performing *timing analysis* on the individual tasks, and then *schedulability analysis* on the system as a whole.

### Timing Analysis

Timing analysis is used to determine the execution time of a task in isolation, specifically excluding any effects due to scheduling. In most cases, a task's execution time will vary depending on factors such as the input data, but also on the state of hardware features such as processor caches. At a high level the analysis must calculate how long each block of code takes to execute, and then combine the blocks together so as to maximise the execution time. *Static*

analysis does this by determining the execution time using a detailed model of the hardware without executing the software. M*easurement-based* techniques measure the execution time of the software running on the target hardware. In systems with cache the analysis must also consider the potential variation in access times to fetch instructions and data depending on the state of the cache.

Real-time systems have to respond to inputs from outside of the system and have specific deadlines that they must meet. Therefore, one of the most important aspects of a task's execution time is what is known as the *worst case execution time* (WCET). The WCET of a task describes the amount of time that a task will spend executing under the worst case scenario, such as the worst case data input, and is obtained using WCET analysis. The goal of WCET analysis is to calculate a *sound*, greater than or equal to the actual WCET, and *tight*, close to the actual WCET, WCET estimate [99].

### Schedulability Analysis

In real applications a system is usually built up of a number of tasks, collectively called a *taskset*. In addition to calculating the WCET of every task in isolation it is just as important to ensure that all the tasks, when running on the same platform and sharing resources, will meet their deadlines. A scheduling policy is used to determine which task in the taskset should run at any given point in time. Schedulability analysis uses the scheduling policy along with information about the tasks and their WCET, obtained through timing analysis, to determine whether or not the system as a whole is schedulable given the hardware resources available. Tasks can either be scheduled *pre-emptively* or *non-pre-emptively* in a multi-tasking system. In a pre-emptive multi-tasking system, tasks can be pre-empted so that a higher priority task can run, which must also be taken into account when performing schedulability analysis. Schedulability analysis can also take into account access to any shared resources that introduce blocking when a task is unable to execute because another task has a lock on a resource which it needs.

## Cache Related Pre-emption Delays

In a pre-emptive multi-tasking system with cache, when a task is pre-empted, *cache-related pre-emption delays* (CRPD) can be introduced. CRPD is the additional cost incurred from resuming a pre-empted task that no longer has the instructions or data it was using in cache, because the pre-empting task(s) evicted them from cache. CRPD will be incurred as the task uses data and invokes instructions during the remainder of its execution that were evicted by

the pre-empting task(s). CRPD is not a fixed cost per pre-emption, as is usually the case for traditional *context switch costs*, so simply subsuming an upper bound on the CRPD into the execution time of the pre-empting task could be very pessimistic. It is therefore important to accurately account for CRPD when performing schedulability analysis on a real-time system. There are techniques that can be used to reduce or completely eliminate CRPD, usually at the expense of increased task WCETs. For example, the cache can be partitioning so that each task has its own space in cache. However, Altmeyer *et al.* [8] recently noted that the increased predictability of a partitioned cache, in terms of eliminating CRPD, does not compensate for the performance degradation in the WCETs due to the smaller cache space per task.

## 1.1 Contribution

The main hypothesis of this thesis is:

> Accurate analysis of *cache related pre-emption delays* (CRPD) is essential for resource efficient scheduling of complex embedded real-time systems.

This thesis focuses on the effects of CRPD on a single processor system and further expands our understanding of CRPD and puts its impact into context through the following:

### CRPD Analysis for EDF

Up until now, research into CRPD analysis has mostly focused on *Fixed Priority* (FP) scheduling [37] [77] [115] [6] [7], and while there exists some analysis for *Earliest Deadline First* (EDF) scheduling [71], we have identified the potential for significant pessimism in the analysis. We therefore present a number of new methods for analysing CRPD under EDF scheduling that significantly outperform the existing analysis.

### Task Layout Optimisation

CRPD is dependent on how tasks are positioned in cache, which is controlled by their layout in memory. We present a technique for optimising task layout in memory so as to increase system schedulability via reduced CRPD.

### Detailed Comparison between FP and EDF

We perform a detailed comparison between FP and EDF scheduling when accounting for CRPD. We explore the relative impact of CRPD on these two

popular scheduling algorithms across a large range of taskset and system parameters in order to gain a better understanding of how CRPD affects system schedulability.

### CRPD Analysis for Hierarchical scheduling

Hierarchical scheduling [56] [60] provides a means of running multiple applications or components on a single processor as found in a partitioned architecture. It is motivated by the need to run multiple components independently of each other without allowing them to impact the functional or temporal behaviour of each other. However, as caches are shared there is the potential for component CRPD to significantly impact schedulability. We present new analysis for bounding CRPD in hierarchical systems.

## 1.2 Structure

This thesis is structured as follows. Chapter 2 covers key background material on caches, timing analysis, and schedulability analysis. Chapter 3 discusses CRPD and reviews existing analysis techniques for calculating an upper bound on CRPD when performing schedulability analysis. Chapter 3 also discusses techniques that can be used to reduce or eliminate CRPD through reduced pre-emptions and greater cache predictability. The new research contributions of this thesis are contained in Chapters 4 to 7. Chapter 4 introduces our new CRPD analysis for bounding CRPD under EDF scheduling. Chapter 5 details how the task layout can be optimised in order to increase system schedulability via reduced CRPD. Chapter 6 presents a detailed comparison between FP and EDF scheduling accounting for CRPD in order to better put the effects of CRPD into context. Chapter 7 extends CRPD analysis to systems using hierarchical scheduling. Finally, Chapter 8 concludes and outlines future work.

# CHAPTER 2. BACKGROUND

In this chapter, we review key background research that forms the basis of the work presented later in this thesis. Section 2.1 covers the basics of real-time scheduling and schedulability analysis. Section 2.2 introduces core terminology relating to caches. Finally, Section 2.3 reviews timing analysis techniques for calculating a bound on the execution time of individual tasks.

## 2.1 Real-Time Scheduling

In real applications a system is usually built up of a number of tasks, collectively called a *taskset*. In addition to calculating the WCET of every task in isolation it is just as important to ensure that all the tasks, when running on the same platform and sharing resources, will meet their deadlines.

A scheduling policy is used to determine which task in the taskset should run at any given point in time. Scheduling policies can be classified as either *offline* or *online*. Offline scheduling, often referred to as static cyclic scheduling, uses a pre-computed schedule with very low runtime overhead. Online scheduling does not generate a schedule in advance, and instead determines which task should run at runtime. Under offline scheduling, the pre-determined schedule ensures that the schedulability of the system is known in advance. Sporadic jobs are more difficult to accommodate, but can be served using spare capacity. The *Slot Shifting* method by Fohler [65] makes use of available capacity after determining a valid schedule for periodic jobs to schedule sporadic jobs online. However, despite the benefits of offline scheduling, it lacks flexibility and may lead to an underutilisation of the processor compared to an online scheduling policy. Due to these limitations, many systems use an online scheduling policy, which is the focus of this thesis. Some classical online scheduling policies include:

- *Fixed Priority* (FP) [80] [85] - Fixed priority policy where tasks are allocated priorities offline and then scheduled according to those priorities at runtime
- *Earliest Deadline First* (EDF) [85] - Dynamic priority policy where jobs with earlier absolute deadlines are given higher priorities. As the priorities are based on absolute deadlines of the individual jobs, task priorities change dynamically over the course of the schedule.

Tasks can either be scheduled *pre-emptively* or *non-pre-emptively* in a multi-tasking embedded system. In a non-pre-emptive system, tasks cannot interrupt each other and run one after the other. Non pre-emptive scheduling is more predictable than using pre-emption because tasks will be allowed to run to completion. However, it is only possible to schedule some types of tasks pre-emptively. In a pre-emptive multi-tasking system, pre-emption is the act of temporarily interrupting a task in order to share CPU time between all the tasks running on the system. This switching from one task to another is known as a *context switch* and can introduce *context switch costs* due to the overhead involved with saving and restoring task state. A task may be pre-empted because a task with a higher priority needs to run, because the task is waiting on access to a locked resource, or because the task has used up its allotted time, otherwise known as a *time slice*.

There are both non-pre-emptive and pre-emptive variants of FP and EDF scheduling. In this thesis we focus on the pre-emptive variants as the non-pre-emptive variants can perform very poorly for tasksets containing tasks with a range of task periods and execution times [47].

## Schedulability Tests

Schedulability tests are used to determine if a taskset is schedulable, such that all the tasks will meet their deadlines given the worst-case pattern of arrivals and execution. For a given taskset and scheduling algorithm, the response time for each task can be calculated and compared against the tasks' deadline. A taskset is *schedulable* if all valid sequences of jobs that may be generated by the taskset can be scheduled without deadline misses. A taskset is *feasible* if there exists a scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the taskset without any deadline misses. A scheduling algorithm is *optimal* with respect to a task model if it can schedule all feasible tasksets that comply with the task model.

For a given schedulability test, it can be categorised as one of the following:

- *Sufficient* - every taskset deemed to be schedulable by the test is actually schedulable.
- *Necessary* - every taskset deemed to be unschedulable by the test is actually unschedulable.
- *Exact* - if a test is sufficient and necessary.

When comparing two schedulability tests, test A and test B the following terms are used:

- *Dominates* - test A dominates test B if all the tasksets deemed schedulable by test B are also deemed schedulable by test A, and test A deems additional tasksets schedulable.
- *Incomparable* - tests A and B are *incomparable* if they each deem a different set of tasksets schedulable.

Schedulability tests are interested in the schedulability of a taskset under the worst-case system load, for which they can use the *synchronous busy period.* From [107] [112], a synchronous busy period is a processor busy period in which all tasks are released simultaneously at the beginning of the processor busy period, and then, at their maximum rate, and ended by the first processor idle period (the length of such a period can be zero). Note that once pre-emption costs are considered the synchronous busy period may not represent the worst-case.

### 2.1.1 System Model

For a complete list of notation used throughout, see the "List of Notations" on page 183.

Our system model comprises a single core processor running a taskset $\Gamma$ made up of a fixed number of tasks ($\tau_1..\tau_n$) where $n$ is a positive integer. We assume a discrete time model. The taskset is scheduled using either pre-emptive FP or pre-emptive EDF. In the case of FP scheduling, each task has a unique fixed priority and the priority of task $\tau_i$, is $i$, where a priority of 1 is the highest and n is the lowest. In the case of EDF, each task has a unique task index ordered by relative deadline from smallest to largest. In the case of a tie when assigning the unique task indices, an arbitrary choice is made.

Each task $\tau_i$ has the following properties:

- $C_i$ - worst case execution time (determined for non-pre-emptive execution)
- $T_i$ - minimum inter-arrival time or period
- $D_i$ - relative deadline
- $J_i$ - release jitter
- $U_i$ - utilisation ($U_i = C_i / T_i$)
- $R_i$ - response time

Each task, $\tau_i$ may produce a potentially infinite stream of jobs that are separated by a minimum inter-arrival time or period $T_i$. Each job of a task has an absolute deadline $d_i$ which is $D_i$ after it is released. We define $T_{max}$ as the largest period of any task in the taskset, and similarly $D_{max}$ as the largest relative deadline of any task in the taskset.

In this thesis we consider tasks with either *constrained* deadlines, $D_i \leq T_i$ or *implicit* deadlines, $D_i = T_i$.

The system model could also contain an additional term, $B_i$, used to represent blocking due to access to shared resources other than the processor that require mutual exclusion. Blocking can be accounted for via approaches such as the *Stack Resource Policy* (SRP) introduced by Baker [16] which we note introduces no additional context switches. However, this thesis uses a simpler system model that does not contain $B_i$.

## 2.1.2 Schedulability Analysis

We now briefly cover existing schedulability analysis for FP and EDF scheduling assuming context switch costs are constant and subsumed into the tasks' execution times.

### FP Scheduling

FP scheduling assigns each task a fixed priority which is then used as the priorities of the tasks' jobs. Under FP scheduling the sets of tasks that can pre-empt each other are based on the statically assigned fixed task priorities. Using the fixed priorities, we can define the following sets of tasks for determining which tasks can pre-empt each other: hp($i$) and lp($i$) are the sets of tasks with higher and lower priorities than task $\tau_i$, and hep($i$) and lep($i$) are the sets containing tasks with higher or equal and lower or equal priorities to task $\tau_i$.

The exact schedulability test for FP scheduling assuming constrained deadlines calculates the worst case response time for each task and then compares it to its deadline. The equation used to calculate $R_i$ is [15] [70]:

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil C_j \tag{2.1}$$

Equation (2.1) can be solved using fixed point iteration. Iteration starts with the minimum possible response time, $R_i^0 = C_i$, and continues until either $R_i^{\alpha+1} > D_i$ in which case the task is unschedulable, or until $R_i^{\alpha+1} = R_i^{\alpha}$ in which case the task is schedulable and has a worst-case response time of $R_i^{\alpha}$. Note the convergence of equation (2.1) may be speeded up using the techniques described in [55].

Under FP there are a number of techniques that can be used to assign the fixed priorities. *Deadline Monotonic* [80] assigns higher priorities to tasks with shorter deadlines. *Rate Monotonic* [85] assigns higher priorities to tasks with shorter periods. Audsley's *Optimal Priority Assignment* (OPA) algorithm [14] takes a different approach. Using a greedy algorithm it evaluates the schedulability of each task, from lowest to highest priority, to devise an optimal priority for each task. It can be applied assuming the schedulability of a task meets certain conditions, such as not being dependent on the relative priority ordering of higher priority tasks. A drawback of OPA is that it selects the first schedulable priority assignment that it finds, which may result in a taskset that is only just schedulable. The *Robust Priority Assignment* (RPA) algorithm [52] improves on OPA by avoiding this drawback.

Assuming negligible pre-emption costs, Leung and Whitehead [80] showed that Deadline Monotonic priority ordering is an optimal priority ordering for constrained deadline tasks which can have synchronous releases. Rate Monotonic is an optimal assignment for tasks with implicit deadlines [85], and OPA can generate an optimal assignment for tasks with arbitrary deadlines and periodic tasksets with offset release times [14].

## EDF Scheduling

In 1973, Liu and Layland [85] gave an exact schedulability test that indicates whether a taskset is schedulable under EDF *if and only if* (iff) $U \leq 1$, under the assumption that all tasks have *implicit* deadlines ($D_i = T_i$). In the case where $D_i \neq T_i$ this test is still necessary, but is no longer sufficient.

Assuming negligible pre-emption costs, in 1974 Dertouzos [57] proved EDF to be optimal among all scheduling algorithms on a uniprocessor. In 1980, Leung and Merrill [79] showed that a set of periodic tasks is schedulable under EDF iff all absolute deadlines in the period [0,max{$s_i$}+ 2H] are met, where $s_i$ is the start time of task $\tau_i$, min{$s_i$}=0, and H is the hyperperiod (least common multiple) of all tasks periods.

In 1990 Baruah *et al.* [19], [20] extended Leung and Merrill's work [79] to sporadic tasksets. They introduced $h(t)$, the *processor demand function*, which denotes the maximum execution time requirement of all tasks' jobs which have both their arrival times and their deadlines in a contiguous interval of length $t$. Using this they showed that a taskset is schedulable iff $\forall t > 0, h(t) \leq t$ where $h(t)$ is defined as:

$$h(t) = \sum_{i=1} \max\left\{0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right\} C_i \tag{2.2}$$

Examining equation (2.2), it can be seen that $h(t)$ can only change when $t$ is equal to an absolute deadline, which restricts the number of values of $t$ that need to be checked. In order to place an upper bound on $t$, and therefore the number of calculations of $h(t)$, the minimum interval in which it can be guaranteed that an unschedulable taskset will be shown to be unschedulable must be found. For a general taskset with arbitrary deadlines $t$ can be bounded by $L_a$ [67]:

$$L_a = \max\left\{D_1, ..., D_n, \frac{\sum_{i=n}^{n} (T_i - D_i) U_i}{1 - U}\right\} \tag{2.3}$$

Spuri [112] and Ripoll *et al.* [107] showed that an alternative bound $L_b$, given by the length of the synchronous busy period can be used. $L_b$ is computed by solving the following equation using fixed point iteration:

$$w^{\alpha+1} = \sum_{i=1}^{n} \left\lceil \frac{w^{\alpha}}{T_i} \right\rceil C_i \tag{2.4}$$

There is no direct relationship between $L_a$ and $L_b$ which enables $t$ to be bounded by $L = \min(L_a, L_b)$. Combined with the knowledge that $h(t)$ can only change at an absolute deadline, a taskset is therefore schedulable under EDF iff $U \leq 1$ and:

$$\forall t \in Q, h(t) \leq t \tag{2.5}$$

Where $Q$ is defined as:

$$Q = \{d_k \mid d_k = kT_i + D_i \wedge d_k < \min(L_a, L_b), k \in N\} \tag{2.6}$$

In 2009, Zhang and Burns [129] presented their Q*uick convergence Processor-demand Analysis* (QPA) algorithm which exploits the monotonicity of $h(t)$ to determine schedulability by checking a significantly smaller number of values of $t$. Let $d_i$ be any absolute deadline of a job from task $\tau_i$, $d_i = kT_i + D_i, k \in N$ and define $d_{\min} = \min\{D_i\}$. When a system is unschedulable, they define $d^{\Delta}$ as:

$$d^{\Delta} = \max\{d_i \mid 0 < d_i < L \wedge h(d_i) > d_i\} \tag{2.7}$$

QPA starts with a value of $t$ that is close to $L$ and then iterates back towards 0. For a schedulable system this sequence converges to 0, but can be stopped once $h(t) \leq d_{\min}$. For an unschedulable system it converges to $d^{\Delta}$. On each iteration $t$ is set to the output of $h(t)$ and $h(t)$ is re-evaluated with the new value of $t$. If $h(t) = t$, then $t$ is set to the largest absolute deadline that is less than $h(t)$. Figure 2.1 shows an illustration of how the QPA algorithm works.

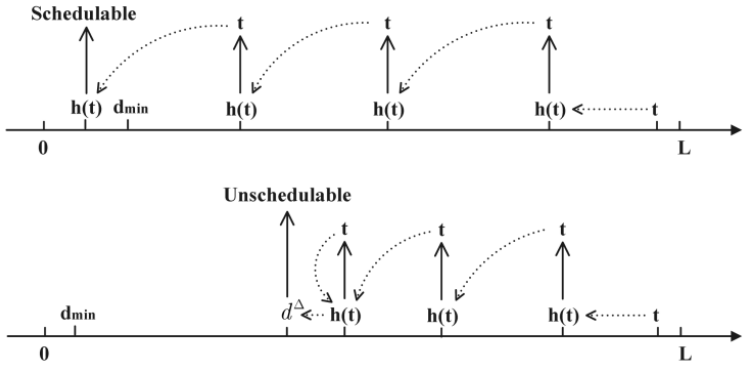



*Figure 2.1 - Illustration of how the QPA algorithm works from [129]*

## 2.2 Real-Time Systems and Cache

There are a number of features in modern processors that improve the average case performance, but make analysis of systems difficult due to the uncertainty that they introduce. These performance enhancing features include *caches*, *pipelines*, *branch predication* and *out-of-order execution*. When performing timing analysis they must be accounted for as they can affect the execution time of the basic blocks of code depending on what has been executed previously. Furthermore, in a pre-emptive multi-tasking system a pre-empting task can

affect the execution time of a pre-empted task by altering the state of these hardware features, for example by evicting the contents of the cache. In this thesis we focus on analysing the effects caused by caches in real-time systems using pre-emptive multi-tasking, which we discuss in detail in Chapter 3. First we give a brief summary about caches, and then review the techniques that can be used to analyse them when performing timing analysis, in Section 2.3.

Caches are small fast memories which are used to speed up access to frequently used blocks that reside in main memory, either RAM or permanent storage such as EPROM. CPU caches are either split into *instruction* and *data* caches, or combined into a *unified* cache. Figure 2.2 shows a simplified representation of a CPU, 4KB of cache and 4MB of EPROM that could be found in an embedded system. Only a small percentage of the data or instructions from memory can be stored in the cache at any point in time, but accesses to the cache require significantly fewer cycles. If the instruction or data resides in cache, then accessing it will result in a *cache hit*, if not, it will result in a *cache miss* and the instruction or data must be fetched from memory first.

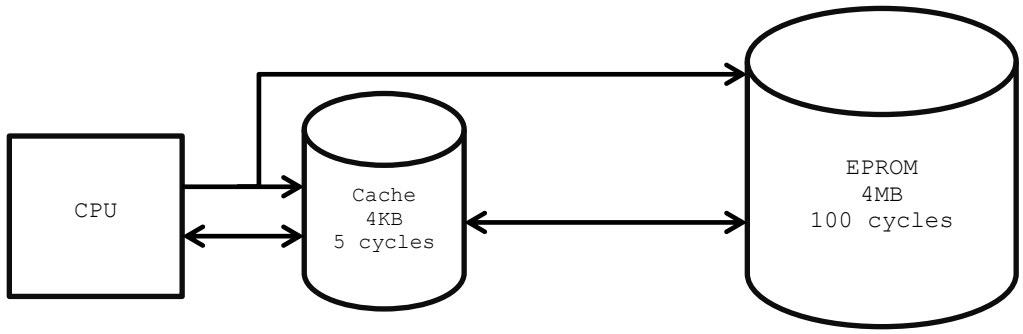



*Figure 2.2 - Layout of the CPU, Cache and EPROM Memory showing relative size and access times*

In this thesis we focus on instruction only caches. In the case of data caches, the existing analysis in Chapter 3 and the analysis that forms the contribution of this thesis (Chapters 4-7) would either require a write-through cache or further extension in order to be applied to write-back caches.

Caches provide a predictable, but almost chaotic performance boost. Provided the current state of the cache is known, whether the next access will result in a hit or a miss can be calculated. However, it can be very difficult to keep track of the contents of the cache. Accessing data which is in the cache will always be faster than accessing data from memory. However, under some scenarios the time taken to execute a set of instructions that are in cache can even be slower

than when the instructions are not in cache. This is referred to as a *timing anomaly* and is caused when other hardware features interact and result in additional blocks having to be loaded from the cache. This makes the ability to classify if a fetch will result in a hit or a miss even more important [88]. One solution is to simply disable the cache. However, as the demands of embedded systems increase it becomes increasingly cost ineffective to keep caches disabled as they can provide such a significant performance increase [44]. It is therefore important to be able to analyse systems with cache in order to verify todays' embedded systems.

Many aerospace systems partition different software systems so that they cannot interfere with each other. As caches are shared amongst everything running on a processor this is a cause for concern. CAST-20 [43] investigated caches in aerospace systems. In particular, it noted that "cache memory should receive special scrutiny in a partitioned system because the cache mechanism is not aware of the address partitioning architecture" [43]. This is a concern as the partitions are supposed to ensure that tasks in one partition do not affect another. However, as caches are not aware of the partitioning tasks in one partition can evict instructions and data belonging to a task in a different partition. This in turn can then affect the execution time of the other task, despite them being separated.

Another problem with cache and predicting its behaviour is that an empty cache is not always the worst case. For example, when the write back policy is being used on a data cache, blocks have to be written back to memory before they can be evicted.

An additional case where an empty cache is not the worst case is the *domino effect* [24]. The domino effect describes a situation where a repeating pattern of instructions cause the cache to transition through a number of states without converging. This could occur when a loop repeatedly calls a number of functions/instructions that are laid out in memory in a specific way. Due to the initial state and replacement policy, the cache does not end up in a consistent state, which means a different number of cache misses can occur on each loop iteration. Due to this effect, it must be assumed that the worst case number of cache misses occur on every iteration of the loop.

These factors combine together to make our ability to accurately analyse caches very important when verifying the temporal behaviour of real-time systems.

### 2.2.1 Cache Structure

In order to maximise the useful contents in the limited cache space, caches work on the principles of locality. At any given time, a task is likely to access instructions or data that it has accessed recently, which exploits temporal locality. A task is also likely to access instructions or data that are located close to those that it has recently accessed, exploiting spatial locality.

Caches are partitioned into a number of *cache sets*, $S$, such that each memory block $m$ maps to a single cache set. Each block can contain $L$ lines, and by loading a memory block with multiple lines caches are able to exploit spatial locality. For example, a memory block may hold 4 lines each containing an instruction which can be loaded into the cache in one go.

Each cache set may contain up to $K$ memory blocks, where $K$ is equal to the *associativity* of the cache, and in the general case, a cache is called a *set-associative* cache with $K$ associativity. A *direct mapped* cache is a special case where $K$=1, resulting in each memory block being able to reside in a single cache set. Conversely, a *fully associative* cache is the other special case where $K$=$S$, resulting in each memory block being able to reside in any cache set.

### 2.2.2 Replacement Policies

Except for direct mapped caches, cache sets can store multiple memory blocks and once they become full they must choose what to evict. This is achieved through a cache *replacement policy*, where the goal is to replace the least useful memory block which can be done by exploiting the concepts of locality. Some of the commonly used replacement policies are listed below [104].

#### Least-Recently-Used (LRU)

LRU replaces the element in cache that was used least recently. It effectively maintains a queue of length equal to the length of the associativity of the set. Every time an element is accessed from cache it is moved to the front of the queue, whether it was in the cache or not. When a cache miss occurs the element at the back of the queue is evicted. LRU does a good job at keeping useful elements in cache.

### First-In First-Out (FIFO or Round-Robin)

FIFIO, which is also known as Round-Robin, uses a FIFO queue to choose what is evicted from cache. It simply replaces the element which has been in cache for the longest time. Unlike LRU, if an element is accessed while it is in cache, it is not moved to the front of the queue. It is however, much simpler to implement than LRU. A downside is that it causes domino effects.

### Most Recently Used (MRU)

MRU keeps track of elements that have been used recently and when a cache miss occurs, replaces an element that has not been used recently. MRU uses a *status bit* for each cache line. On each access, this status bit is set to 1 and once the last status bit is set to 1, all other status bits are reset to 0. Once a cache miss occurs, one of the elements with a status bit that is equal to 0 is replaced.

### Pseudo-LRU (PLRU)

LRU can become prohibitively expensive to implement in caches with large associativity, such as 4-way or greater. Pseudo-LRU is an alternative that almost always discards the least recently used element by using a tree-based approximation of LRU. Each node in the tree records which leaf is older/newer. Each time an element is accessed, the nodes are updated. When a cache miss occurs, the tree is followed to find the element to be evicted. Pseudo-LRU caches can also cause the domino effect.

### Random/Pseudo-Random

Random or Pseudo-Random replacement polices make no attempt to keep important elements in cache; instead they replace elements at random. It does not require storing any information to decide what to evict and is simple to implement as it only requires a random or pseudo-random number generator. A benefit of random replacement policies is that *probabilistic* analysis [54] [5] can be performed on caches that use it. Additionally, random/pseudo-random replacement policies reduce the possibility of performance anomalies due to access history [102].

## 2.3 Timing Analysis

In order to determine if a taskset is schedulable when running on a multi-tasking system, it is essential to know how long each of the tasks could take to execute. This is achieved by performing timing analysis on the tasks. Timing

analysis methods can be classified into three types of analysis; *static*, *measurement-based*, and a combination of the two *hybrid measurement-based* analysis. Static analysis calculates the execution time for blocks using a model of the hardware. Measurement-based analysis executes the software on the target hardware and records execution time measurements. Hybrid measurement-based analysis combines the two. It determines the execution times by measuring small sections of code, and then calculates a bound on execution time based on the program structure obtained using static analysis and the collected measurements. While this thesis does not focus on timing analysis, we present a brief review of the literature as it forms the basis for later work on the cache analysis required by CRPD analysis.

### 2.3.1 Static Analysis

Static WCET analysis aims to calculate an upper bound on the WCET by statically calculating what the execution time for each block of code will be, and then combining them together to find the *worst-case path* (WC path) through the code.

#### Initial Work

Early work on static WCET analysis was driven by the seminal paper by Puschner and Koza in 1989 [100]. In [100], Puschner and Koza used source code to try to calculate an upper bound on the maximum execution time of tasks. Calculating an estimate for the WCET of an arbitrary program reduces to the *Halting problem* [74]. It was therefore apparent from the onset that a number of restrictions would have to be placed on the code in order to facilitate estimation by bounding the execution time. Some of those restrictions such as not using GOTOs and not having unbounded loops and recursive procedures are still present in today's techniques. In order to add additional information to the source code a number of high level path description constructs were defined. These were based on C like syntax and include things such as the ability to specify the maximum number of iterations for loops using bounds, and markers for dealing with multiple paths through loops. They proposed a set of formula, or timing schema, that could be used to combine together execution times for simple language constructions, assuming the execution time for them could be obtained. For example the execution time for a sequence of statements is the sum of the execution times for each statement. A downside of this approach is that it requires modifying the source code in ways such as replacing standard loops with their modified bounded versions.

Later in 1991, Park and Shaw [95] took an alternative approach of using external annotations which has the benefit of not requiring a new programming language or language subset. Additionally, they focused more on the mapping between source code and the resulting object code. They used two levels of granularity in their analysis, *small atomic blocks*, and *large atomic blocks*. A small atomic block is as small as possible and could be an assignment, or an addition, for example, $a = b + c$ contains two atomic blocks. However, this is complicated by simple compiler optimisations.

An example from [95] is that the sequence $a = b + c; d = d + a$ can be compiled as follows:

```
a = b + c; ==> mov @b, d0
               add @c, d0
               mov d0, @a
d = d + a; ==> add d0, @d
```

*Figure 2.3 - Optimised assembly code generated from two simple statements [95]*

In this example, `@a` is the memory address of variable *a*, and `d0` is data register 0. As variable *a* was already in a register after the first statement, the second statement can be achieved in one machine instruction. This then makes it difficult to predict the execution time of a source code statement when considering it in isolation. Compiler optimisations can also cause multiple atomic blocks to be merged into one machine instruction. In the example, = and + are achieved using one add for the second statement. Most problems like this can be eliminated by using their second level of granularity, large atomic blocks, which are as large as possible and represent an entire basic block. Where a basic block is a sequence of instructions without any decisions or branches so that the control flow enters at the beginning and leaves at the end. Regardless of which level of granularity used, Park and Shaw combined together the atomic blocks using a simple timing schema in the same way used by Puschner and Koza in [100]. In their work they also considered system interferences in their calculations due clock interrupts and dynamic RAM refreshes. However, they did not consider the effect of advanced hardware features such as pipelines or caches. In order to examine the effectiveness of their tool, Park and Shaw collected measurements of the code and compared it against the predicted bounds. For simple procedures, they were able to successfully calculate tight bounds. For complex procedures such as those with nested loops, where the number of iterations for the inner loop is dependent on the iteration number of the outer loop, such as sorting algorithms resulted in much looser bounds. This

was refined by introducing more user annotations that enabled infeasible paths to be eliminated which helped to produce tighter bounds.

In 1993 Park [94] started work on defining and refining which user annotations are needed for calculating a tight bound on the WCET. These user annotations provide execution information about the program which has since been known as *flow facts*. These flow facts describe information such as loop bounds, dependencies on conditions or statements and frequency relationships for sub paths through loops. Using this information Park performed dynamic path analysis to eliminate infeasible paths which leads to reduced pessimism while keeping the estimate sound. Park concluded that at a minimum loop bounds must be provided with additional information helping to make further improvements. In some cases complete information is not necessary as partial information can often be sufficient. Therefore, it is worth providing the broad and general information first, then refining it with more specific localised information.

Early static analysis found the WC path by using a tree based approach backed by a timing schema. Provided the execution times of each basic block are known, they can be added to the tree which can then be traversed from the bottom up to find the WC path. This only works when the execution times of procedures and blocks are independent, which is not the case in modern processors with caches and other hardware features. In 1997, Puschner and Schedl [101] proposed using a graph based approach for finding the WC path, otherwise known as path based approaches. The approach used timing graphs which are similar to flow graphs to represent the structure and timing behaviour of the code. Flow facts are used to constrain the graph and the problem is then solved by finding the path through the graph with the maximum cost using *integer linear programming* (ILP).

The initial path based static analysis techniques used *explicit* path enumeration to find the WC path. After all infeasible paths had been evaluated every possible path was explicitly examined. The following example from [81] illustrates the problem.

```
for (i=0; i<100; i++) {
  if (rand() > 0.5)
    j++;
  else
    k++;
}
```

*Figure 2.4 - Example of exponential blowup of paths if every path is explicitly enumerated from [81]*

The loop in Figure 2.4 above has $2^{100}$ different paths and yet if incrementing *j* and *k* have the same cost then all of the $2^{100}$ paths are WC paths. Li and Malik in 1997 [81] proposed that by *implicitly* considering each path in the solution, the computational effort can be significantly reduced. This is known as *implicit path enumeration technique* (IPET), and forms the basis of the modern static analysis process.

## WCET Analysis Processes

Modern static WCET analysis uses IPET to express the analysis problem as an ILP that is solved by maximising an objective function to find the path with maximal length. The execution times of basic blocks are determined using very detailed and accurate hardware. There are different approaches that can be used to find and combine all the required information, but it is usually broken down into the following phases [61] [126]. Reconstruction of the *call graph* (CG) and *control flow graph* (CFG), *architecture modelling* broken down into *pipeline analysis* and *cache analysis*, and *value analysis*. Finally, *path analysis,* which is the process of generating and solving an ILP, to compute the path through the program that maximises the execution time.

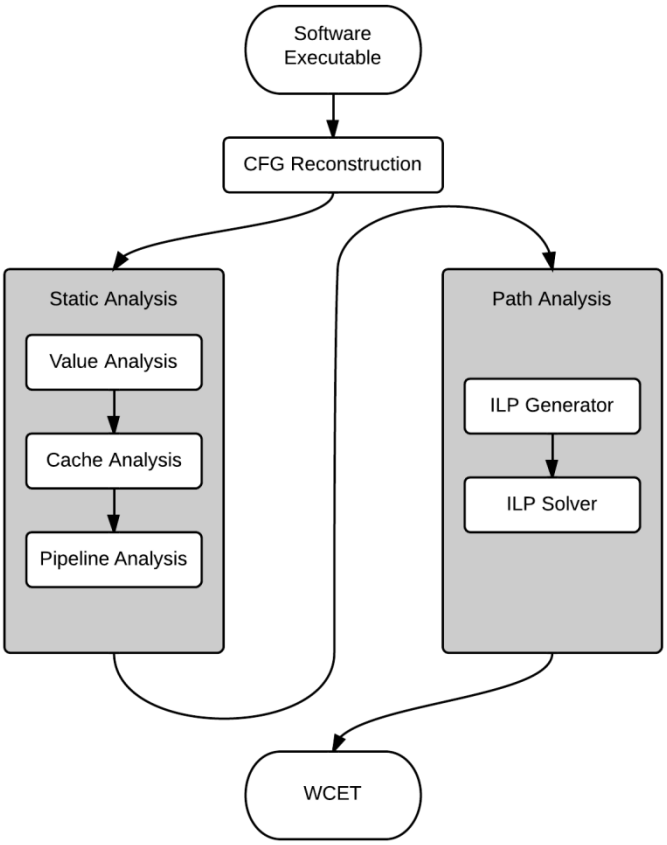



*Figure 2.5 - WCET analysis process for a typical static analysis tool*

While these are different phases, most techniques solved all phases together in order to calculate as tight an estimate as possible. This is because the outcomes affect each other: the value of inputs affects which paths are taken which affects the execution time of blocks due of hardware features. This then affects which blocks are on the WC path. The result of this combined analysis is a potentially very large ILP problem that must be solved using ILP solvers. Today value analysis can be used to determine a large number of flow facts automatically.

### Architecture Modelling

Regardless of how the blocks are combined the execution times for the basic blocks need to be found. This analysis needs to determine how long a basic block will take to execute which is dependent on the type of instructions in the block, the input data, and any hardware features that effect the execution time. Architecture modelling accounts for hardware features such as pipelines and caches and along with value analysis is usually solved using *abstract interpretation*. Using abstract interpretation to perform cache analysis is discussed in Section 2.3.2. Abstract interpretation is semantics based, meaning it computes approximate properties of the semantics of programs. The key concept is it hides some of the details, while still remaining correct, so that a

simplified representation can be used. This increases the feasibility of the analysis by making it easier to obtain a result in a finite time. This enables the problem to be solved as an ILP system. In 1977, Cousot [45] applied abstract interpretation to static analysis of programs, forming the basis for much of the research that has been conducted since.

### Value Analysis

Ferdinand and Wilhelm [62] explain that abstract interpretation is used to perform a program's computation using *value descriptions* or *abstract values* in place of *concrete values*. This allows one to work with a set of inputs, ideally all inputs, rather than just one input. This also helps to ensure the computation completes in finite time. The results obtained from abstract analysis while often less precise, can still be proved to be larger than the real WCET; they never underestimate it. An example given from [62] is that if a boolean variable is sometimes true, then its value is correctly described by "I don't know", but not by "false". To guide the results, an objective function is defined and constraints are placed on it. In static WCET analysis, the objective would be to maximise the execution time. The constraints placed aim to prevent the WCET estimate from becoming too pessimistic by, for example, excluding infeasible paths.

### Path Analysis

The last part of the problem is the path analysis which comes down to solving a potentially very large ILP problem. Once the overall structure of the software has been obtained from the object code the path analysis must identify the WC path. There will often be a number of possible WC paths that depend, directly or indirectly, on the input data. The path analysis aims to eliminate as many of the infeasible paths as possible. This helps to increase the accuracy of the overall WCET estimate, as the estimated execution time for those paths do not need to be included. Using flow facts, either provided by the user or found using value analysis, infeasible paths can be eliminated. This is achieved by bounding loops and specifying dependencies between blocks of code, especially inside conditional statements. When this is combined together along with the architectural modelling, an ILP problem representing the system with a number of constraints must then be solved.

## Limitations

Increased complexity of modern processors has made analysis more difficult and computationally more intensive due to the higher number of factors that

must be taken into account. There are some techniques which can help to make the cache easier to analyse, but they do not cover all cases. This has led to an alternative approach where the architectural model is separated and then used as direct input to the ILP, rather than forming part of an overall larger ILP. Examples include the separation of the cache analysis [116] by Theiling *et al*., the idea being that the problem can be broken down into smaller less complex problems which are then composed together. However, as previously noted this results in a more pessimistic WCET estimate because of the lack of feedback between the different parts of the analysis.

The described analysis is achieved by analysing the program without executing it. However, additional information is almost always required in the form of annotations provided by the developers to better describe the system. These annotations help to fill in the missing information from the analysis. For example, Section 4 of AbsInt's white paper on their static analysis tool aiT [1], details the required annotations that are needed in order to obtain a WCET estimate. At a minimum, aiT requires the maximum number of iterations for loops and the targets of computed calls and branches. If recursion is used, then upper bounds on the recursion depth must also be specified. Any function pointers will also require annotations. Information about memory mapping is also required if accesses to different memory locations have different access times.

Once the required annotations are provided they must be maintained along with any changes to the system which can be a non-trivial challenge. Moreover, if the developers' understanding of the system or their model of the inputs is incorrect, the WCET estimate will be inaccurate. Applying the static WCET analysis tool aiT to automotive communication software is discussed by Byhlin *et al*. [40]. The authors noted that detailed system and code knowledge is often required and a number of annotations must be supplied. They also had to use relative addressing in their annotations and the analysis often required them to make changes and then recompile the software, which altered the code layouts.

As static WCET analysis tools rely on an accurate and complete model of the hardware, a new model must be developed for every new configuration of hardware. However, these models are inherently costly to develop because of the complexity of modern hardware which limits the availability of them to the most commonly used hardware.

### 2.3.2 Static Analysis for Systems with Cache

Static analysis techniques can produce very pessimistic WCET estimates when cache is used because of the difficulty of knowing what will be in cache at any point in time. Being able to accurately model the state of the cache is therefore essential in calculating a tight WCET estimate.

Cache analysis in WCET analysis was originally proposed in 1994 in Mueller's PhD thesis [90] via *static cache simulation.* Static cache simulation simulates the state of the cache at each program point using dataflow analysis. From this *abstract cache states* which describe the possible states of the cache can be found. These abstract cache states describe what may be in cache and take a sound but often pessimistic view of the cache. Using the abstract cache states, Mueller proposed four ways to categorise each instruction using instruction categorisations; *always-hit*, *always-miss*, *first-miss* and *conflict*. Always-hit is for instructions that are always in cache when fetched while always-miss is for instructions which are never in cache when fetched. First-miss is common for instructions that form part of a loop. On the first iteration they are not in cache, but in subsequent iterations they hit as they have now been loaded. Finally conflict is for any remaining instructions which were not categorised using the first three options. In more recent literature instruction categorisations are known as cache categorisations, conflict is often referred to as *unknown* and an additional *first-hit* has been introduced. Mueller's approach was only applied to direct mapped caches and used a simple union to merge abstract cache states at control flow merges. In the case where the abstract caches states were different at a control flow merge, any non-matching entries are marked as conflict.

#### Set-Associative Caches

In 2000 Mueller [91] extended his approach to work with set associative caches using the LRU replacement policy. Set associative caches introduce additional challenges into the analysis because multiple blocks can be in the same cache set simultaneously. As new ones are added the blocks age and depending on the replacement policy, the oldest block is evicted. The analysis must therefore track which blocks are in cache and how old they are as that then determines when they will be evicted. Additional pessimism can be introduced at control flow merges when the abstract cache states are combined using a union because of the extra potential for uncertainty. In order to limit this, additional data flow analysis was introduced. This included *linear cache states* to determine whether a block will be in cache before the first iteration of a loop, the difference between

always-hit and first-miss. Secondly the *dominator cache states* were used for determining what must be cached at a specific program point, used for determining which blocks will be always-hit. Finally *post-dominator sets* were introduced to determine what will be cached at a specific program point in the future, regardless of the path taken to reach that point. Despite this additional analysis, pessimism is still introduced if the abstract cache states are very different.

### Cache State Merging

Alt *et al.* in 1996 [2] introduced *must* and *may* analysis, described below to deal with merging abstract cache states at control flow merges for set associative caches. This is an alternative method which builds on the concept of cache categorisations introduced by Mueller [90]. As the analysis is dependent on the replacement policy, the following explanation of must and may analysis is just for the LRU replacement policy, and is for a fully associative cache.

Must analysis determines what must be in cache and aims to find as many blocks that are definitely in the cache as possible. This uses the maximum age of each block to determine if it must be in the cache. Using Figure 2.6 as an example, *d* is known to definitely be in the same place in cache in both paths, so it is kept in the same place. Block *a* has two different ages, as does *c*, so the maximum age is taken. Blocks *e* and *f* are not presented in both abstract cache state so it cannot be determined if they are still in the cache after the merge.

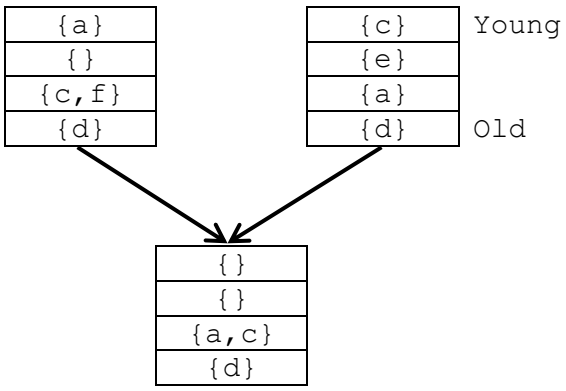



*Figure 2.6 - Merging cache states using must analysis example from [61]*

May analysis aims to eliminate blocks that definitely are not in the cache anymore. Any blocks which cannot be determined to not be in cache may be left in cache. In order to achieve this, the minimum age of a block is used. In the case where the block is present in one abstract cache state and not the other, it

must still be considered. Figure 2.7 shows the same example as Figure 2.6 but with may analysis. As none of the blocks are evicted the resulting abstract cache state contains all the original blocks in their youngest place in the cache.

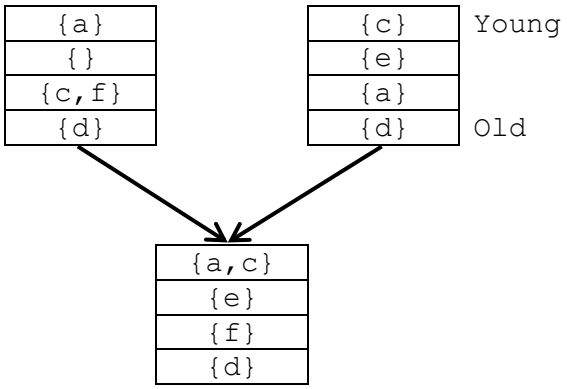



*Figure 2.7 - Merging cache states using may analysis example from [61]*

In addition to must and may analysis, *virtual inlining virtual unrolling* (VIVU) is used to determine which blocks will miss when first accessed, but hit on a subsequence access. These blocks are the same as first-miss using Muller's cache categorisations. This is important for analysing loops and recursive procedures. This virtually[1] inlines non-recursive procedures and virtually unrolls the first iteration of all recursive procedures and loops. The benefit of this approach is that it accounts for blocks being loaded in cache and reused during loops and small recursive procedures. Otherwise they would have to be categorised as always miss or unknown using earlier analysis techniques. Further work such as in 2000 by Theiling *et al.* [116] defined a separate *persistence* analysis which uses VIVU combined with a slightly modified abstract cache state update function.

The must, may, and persistence analysis is solved by starting with empty abstract cache states at each program point and then iteratively updates them until all abstract cache states become stable. Once the must, may and persistence analysis has been performed, a cache categorisation can then be assigned to every block. Any block found to be in the abstract cache state after must analysis is categorised as always-hit. Any block not found in the abstract cache state after the may analysis is categorised as always-miss. In Mueller's work [91], he effectively just had may analysis and inferred the results of the must analysis from the may analysis and the additional data flow information.

[1] Virtually in this context means that the source or object code is not modified. Instead the representation that is used for analysis is.

As with Mueller's work, this approach still suffers from the same problem of introducing pessimism at control flow merges.

The aim of the must, may and persistence analysis is to determine which blocks are in cache at any given program point. However, during the analysis uncertainty about the state of the cache is often introduced due to the abstractions. This is especially so when the cache analysis is performed separately. This leads to the analysis being unable to determine the state of a block. In this case the sound approach is to calculate the execution time for both, and use the worst of the two. Additional uncertainty is also introduced when function pointers are used unless the user annotates them sufficiently.

It is also important to note that the analysis is specific to the replacement policy and the configuration of the cache, for example, its associativity. Replacement policies and configurations are applied using additional constraints and are specific for each instance. LRU is the easiest to model because it is the most predictable [105]. However, many modern processors have the more cost effective to implement policies that are less predictable, such as the Pseudo-LRU policy. Reineke *et al.* [105] presented an analysis of the predictability of different replacements policies for the purpose of static WCET analysis. In the case of Pseudo-LRU, the must analysis will find fewer blocks that must be in cache and the may analysis will find more. This is because Pseudo-LRU will not always evict the least recently used block and because of this, extra pessimism is introduced in the result. FIFO is similar to LRU in that it maintains a queue based on the age of blocks. The difference is that blocks are not moved to the front of the queue when accessed. This makes analysis of FIFO in the case of a miss the same as for LRU but in the case of a hit, it can only be guaranteed that the block is in the cache, and could be evicted on the next access. MRU is even more problematic because it tracks accesses by setting a status bit to 1, however, once every cache line's status bit is set to 1, it resets them all to 0. This leads to the cache never being in a state where its entire contents can be determined.

In the case of random replacement policies, it is not possible to deterministically analyse the contents of the cache. Instead, probabilistic analysis, which uses the probability of an access resulting in a cache hit to generate a probabilistic distribution of the execution time, can be utilised. Examples of this include probabilistic analysis developed under the PROARTIS project [42]. Altmeyer *et al.* provided a review of static probabilistic timing analysis in [5]. In [54] Davis *et al.* extended the analysis to deal with the effects of cache in multi-tasking

systems. However, the focus of this thesis is on analysis for deterministic replacement policies.

## Integration with WCET Analysis

The cache analysis is of little use on its own and must be combined with WCET analysis. While it can be applied to an overall ILP problem such as in Alt *et al.* [2], Theiling *et al.* in 2000 [116] presented a method that performs the cache analysis separately. The results can then be used as constraints for the overall ILP problem and allow the ILP problem to be simpler. This makes the overall computation effort smaller, but it does increase the pessimism in the final estimate as valuable information is not fed into the cache analysis about which paths have been taken.

## Data and Unified Caches

The methods described so far all focused on analysing instruction caches, which is the focus of this thesis. There has been work towards analysing data and unified caches [63]. Cache analysis can be used if the addresses of referenced data can be statically computed. This means that global variables are usually easy to determine. As local variables and parameters are placed on the stack and are addressed relatively based on the stack pointer, if recursion is not in use then data flow analysis [72] can be used along with stack analysis. However, some addresses cannot be statically determined, such as those referenced by pointers or arrays. In this case the analysis must consider a set of possible memory locations, rather than a specific memory location, when performing must and may analysis. This inevitably results in increased pessimism.

Data caches introduce additional challenges because they can be written to as well as read from. Some of the write policies are easier to analyse than others. Write through caches are simpler because they write the contents back to memory straight away. Write back caches are more complicated because they only write a modified block back to memory when it is evicted. This is implemented in the hardware using a *dirty bit* to indicated blocks which have been modified. In order to analyse these caches, the analysis is extended in the same way to also include a dirty bit, and the must and may analysis are adjusted to account for it.

## Multi-level Caches

While the focus of this thesis is on single level instruction caches, it is becoming ever more common in embedded systems to see multiple levels of cache, either

two (L1 and L2) or even three levels (L1, L2 and L3) in multi-core systems. These extra caches sit between the top level cache, L1 cache, and main memory. They will be larger than the L1 cache, but with higher access times. They will still be faster than main memory.

The following example is based around fetching an instruction in a system with an L1 and L2 instruction and L1 and L2 data cache. When loading the instruction, the L1 instruction cache will be checked first. If the instruction is not there then the L2 instruction cache will be checked next. If the instruction was not in any of the cache levels, then the instruction will be fetched from main memory. For each level of cache that must be searched there is an additional, ever increasing, delay. Therefore, the analysis needs to track which cache level each block is in in order to calculate an accurate WCET estimate.

In 2011, Hardy and Puaut [69] extended the cache analysis developed by Theiling *et al.* [116] to work with multi-level caches by introducing the concept of *cache access classification* (CAC). For every memory reference *r*, and cache level *l*, a CAC is determined that captures whether *r* will result in an access to cache level *l*. A CAC can be one of the following; *always*, *never*, *uncertain-never* or *uncertain*. Where uncertain-never describes an access that could or could not occur the first time, the next access will never occur at cache level *l*. The CAC combined with the *cache hit/miss configuration* is then used for analysing the next cache level. They described their analysis for *non-inclusive*, *inclusive* and *exclusive* cache hierarchies using the LRU replacement policy. They have also adapted it for non LRU replacement policies. They noted that a current challenge is that pessimism in the cache analysis of the previous cache level effects the results of the next level. Extending this to three levels of cache as found in some multi-core systems and the need for increased precision in cache analysis becomes even more important.

### 2.3.3 Measurement-based Analysis

Measurement-based WCET analysis is an alternative to static analysis. It is also sometimes known as *dynamic* analysis and is commonly used in industry. Rather than analysing the executable the software is run on the target system. The simplest form works by recording the time at the start of a system or tasks' execution and at the end of it as it executes on the target. This could be achieved for example, by setting an external pin high at the beginning and setting it low at the end of a task. A probe could be attached to the pin and it would record the length of time that the pin was high.

One problem with measurement-based analysis is that measuring the end to end timing of a system will not reveal the WCET unless the WC path is exercised by the test case. Due to system complexity, and dependence on input data, it can be very difficult to find a test that exercises the WC path. It is possible to design systems so that a WCET estimate that is close to the real WCET. This can be achieved by making the code very simple or having a single path through the code. However, this may not be feasible for complex software, making it practically impossible to execute every possible path. Furthermore, good functional test cases may be very poor at exerting the worst case temporal behaviour, further increasing the number of tests need. This limited testing then introduces the problem of working out which tests to run and when sufficient testing has been performed. Because of this problem, simple measurement-based analysis are unsuitable for determining WCET estimate that is guaranteed to be at least as high as the real WCET, unless the software has a very small number of paths.

One solution to the problem was proposed in 1997 when Mueller and Wegener [92] used a genetic algorithm to try to find good test cases. They start with an initial population of test cases which they evaluated. Test cases that resulted in a high WCET were regarded as strong individuals and were brought forward through the generations. The end result was test cases that gave the highest WCET estimate. This allows good test cases to be found that are better than randomly trying different ones. However, the discovered test cases resulted in a lower WCET estimate that the actual WCET. This highlights the fact that good test cases, especially end-to-end ones, are difficult to find.

A benefit of measurement-based analysis is that while the computational cost for static WCET analysis increases with the complexity of the system, measurement-based WCET analysis scales linearly with the number and size of tests. Additionally, it does not require an often expensive and complex to develop hardware model. It is therefore easy to adapt techniques when moving to newly released hardware as there is no need for a new hardware model to be developed.

### 2.3.4 Hybrid Measurement-based Analysis

*Hybrid measurement-based* WCET analysis combines statically obtained control flow information with measurements collected from the software running on the target. These measurements often replace the value analysis and architectural modelling that are used in static analysis, although flow facts can

still be gained using static analysis. As with pure measurement-based analysis, the quality of the results are dependent on the coverage of the test cases. However, because blocks can be combined together from a number of runs, there is less of a need to find a test cases that exercises the complete WC path through the code. Hybrid measurement-based analysis requires fewer annotations for use in determining flow facts than static analysis, but may require annotations to control/optimise to computed WCET bound. These points combine together to make hybrid measurement-based analysis a very attractive alternative for industry. A potential for optimism in the computed WCET bound is that blocks are combined to build the WC path under the assumption that they are independent and that the architecture is timing compositional. In practice, performance enhancing features such as caches can cause the execution time of a block of code to be dependent on what has previously been executed. While this could be solved by testing all possible paths and obtaining full path coverage, this is often unfeasible.

An example implementation of the hybrid measurement-based approach is pWCET [25] where the background was first described in [26]. pWCET uses probabilistic WCET analysis to calculate *execution time profiles* (ETPs) for each basic block of code. Note that this is not the same as applying extreme value statistics to the measurements to account for missing tests. Instead measurements are recorded for every run of each block and combined together to create the ETPs.

Non-probabilistic analysis would only records the maximum, and in some cases the minimum values. Either form of analysis must combines these blocks together, usually using some form of timing schema. The following is an example simple schema based on a syntax tree representation that allows timing information to be combined.

- W(**X**) = integer, when **X** is a basic block
- W(**X**; **Y**) = W(**X**) + W(**Y**) - combines together two blocks, **X** and **Y**
- W(if **Z** then **X** else **Y**) = W(**Z**) + *max*{W(**X**), W(**Y**)}
- W(for **Z** loop **X** ) = ($n$ + 1)W(**Z**) + $n$W(**X**) - where $n$ is the maximum number of iterations

In the case of pWCET, rather than using integers, ETPs are used when combining the values for each block. In order to function correctly with the ETPs the additions must be replaced with join operators. In the case where the ETPs are independent the join is simply a convolution. However, in the case of dependent ETPs there are effects that are (possibly highly) correlated that are

not accounted for. When precise information about dependencies is known, alternative operators can be used. In this case the two ETPs can be joined to give an ETPs equal to P(A = t ∧ B = *s* ). In other words, the join gives the probability that block A runs for *t* time units *and* block B runs for *s* time units.

pWCET has since been turned into a commercial tool, Rapita Systems' RapiTime [103]. While obtaining a probabilistic WCET estimate is useful for some real-time systems such as communications which need to achieve a certain QoS, many hard real-time systems need absolutes. In that case the highest values from the ETP can be taken which is the approach used by RapiTime, although it can display the full ETP for use in appropriate scenarios. An example for presenting ETPs is when attempting to optimise the code. Having a distribution of the measured execution times enables insight into the variation in execution times necessary to make improvements.

The full approach used in RapiTime is as follows. First, the structural analysis is applied to the source code and then pre-processed. During the analysis, the CG and CFG are obtained so that the measurements can be matched and combined with the correct blocks of code. The pre-processed code is then turned into instrumented code by inserting *instrumentation points* (Ipoints). These Ipoints are usually macros or small procedures that output an ID and timestamp which can then be recorded. The instrumented code can then be compiled in the normal way to produce an instrumented executable. When executed on the target, the Ipoints that were inserted into the code write data to memory or to an output port. The IDs in the data are then used to match the timing information from the timestamps with the CFG obtained during the initial analysis. Execution times for each procedure can then be composed together using a tree based approach from the bottom up to calculate the WCET estimate. However, as noted before, this does not account for dependencies between the execution times of procedures caused by caches and other hardware features unless full path coverage has been obtained.

As with pure measurement-based approaches, blocks must be tested in order to collect timing information for them. Ernst and Ye [58] proposed an approach where they reverted to standard static analysis for blocks without timing information that were not successfully tested. This enables relatively fast calculation for the ideally small number of blocks. However an accurate hardware model is then required which is one of the major points that hybrid measurement-based analysis is supposed to address.

The above mentioned techniques used tree based approach to calculate the WC path based on the source code. Betts and Bernat [30] proposed a method to transform graphs based on object code into a tree so that a timing schema can be used as if it was based on the source code. This is an interesting take on the problem, although this has not been fully implemented in a tool. The benefit of starting with object code is that it eliminates any uncertainty introduced by compiler optimisations.

It can be difficult to collect measurements from some systems for a number of reasons, including:

- If there are no free I/O ports to connect a logic analyser to the target
- If there are free I/O ports, but using them significantly limits performance and under-utilises the CPU due to the slow speed of the I/O ports
- If the above two scenarios hold, the only way to extract data may be to store it in memory, and then download it later. However, if the on board memory is limited in capacity, this could result in only being able to test a small portion of the system.

Due to this, the method of extracting data must be tailored to the specific hardware. Hybrid measurement-based analysis will often require a large amount of data to be extracted from the target system. Depending on the detail required, the source code can have different levels of instrumentation, which will generate more or less data to be extracted. It is therefore important in large systems with limited I/O port bandwidth or available memory to pick suitable levels of instrumentation. This could range from recording timing information for each procedure, down to instrumenting each basic block of code. Initially, everything could be instrumented at procedure level. On a second run, the procedures that contribute to the WCET could have more instrumentation added in order to obtain extra information about which parts of the code are contributing the most to the execution time. Even greater detail can be obtained if necessary. This may be useful when evaluating the performance of one statement over another where, for example, one uses specific hardware features of the processor. However, it would generally not be used in the final code as the overheads may become prohibitive if every other statement is an Ipoint that requires a memory or I/O port access.

Measurement-based techniques also suffer from the *probe effect* due to modifying the source code in order to generate the data required to measure the execution time. In doing so the behaviour of the code is altered which can affect

the execution time. In systems with cache, the additional code can alter the layout of code in memory, potentially affecting the number of cache hits/misses across the whole software.

An alternative is to use industry standard hardware debuggers such as ARM's ETM [12] or Nexus [93] which are built into some chips. These allow a consistent way to extract execution information from targets in the form of branch traces. These traces record every branch that is taken [29] and therefore branches that are not taken must be inferred. They however produce less accurate measurements as records are grouped together and time stamped, rather than individually time stamped. Additionally, information could be missed if there is a high number of branch instruction grouped closely together due to bandwidth limitations of the JTAG port that is used for communication.

### 2.3.5 Measurement-based Analysis for Systems with Cache

The execution time of a basic block in a system with cache is history dependent: execution time of a block can vary depending on the path that was taken to get to it. Therefore, measuring all of the individual paths and combining them is only valid for the specific path through the program. To produce a sound WCET estimation when cache is used, full path coverage is technically needed. Figure 2.8 shows an example demonstrating why full path coverage is required. The WC path after executing the two solid line tests was calculated to result in a WCET of 150, as shown by the dashed line path. If in this example loading B into cache evicts F from cache, when it would otherwise have been in cache after executing C and D, then the WCET would increase to 190.

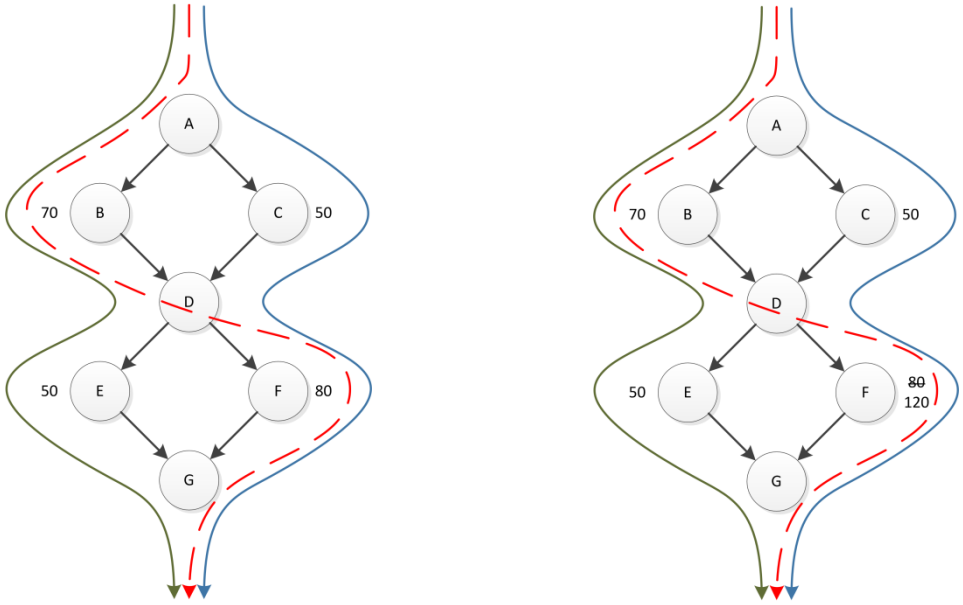



*Figure 2.8 – Example showing why full path coverage may be needed in a system with cache. Executing the solid line tests results in a calculated WC path (dashed line) of A->B->D->F->G with an estimated WCET of 150 (left). If B evicts F, which would otherwise still be in cache when it is called after D, the WC path would remain the same, but the WCET would increase to 190 (right)*

There are some potential solutions to this problem. In 2000 Petters [96] flushed the cache before each measurement block in order to obtain a WCET estimate that was not affected by the cache. However, this clearly introduces a large amount of pessimism and will remove most of the benefits of using the cache. In 2003, Colin and Petters [44] investigated how much of an effect different hardware features had on the WCET. They found that for the SimpleScaler simulator, the level of overestimation was much smaller than the performance loss due to disabling the cache. This emphasises the importance of cache and our ability to account for it when performing WCET analysis.

In 2005 Kirner *et al.* [75] took a similar approach to Petters [96] by partitioning the CFG into program segments. While they did use basic blocks, they also used larger multi-path program segments which had a number of paths through them. This allowed for a less pessimism via considering larger blocks in isolation, at the expense of requiring a higher number of measurements. In order to ensure that all the paths were tested, the authors used a model checker to generate suitable test cases, rather than relying on manually defined test cases. An extension to this could be to consider procedures in isolation and try to obtain full path coverage for each procedure.

A slightly different way to tackle the problem was taken by Betts *et al.* [31] in 2006 with their concept of WCET coverage. Although this was designed for pipelines, the ideas are still relevant for caches. It is based on the fact that traditional functional coverage metrics, such as branch coverage or MC/DC coverage, will often result in poor temporal coverage when advanced processor features such as caches are used. Because of this, and despite the benefits of hybrid measurement-based analysis, there is no way to prove that sufficient testing has been performed. They therefore setup a number of WCET coverage metrics which reflect different levels of temporal coverage when pipelines are used. A basic form of WCET coverage for caches can be achieved by applying the technique used in Petters [96] to flush the cache at the start of each basic block. Kirner *et al.* [75] presented an approach where the CFG is split into multi-path program segments, in which case WCET coverage would be obtained by ensuring that every path through each program segment had been tested.

## 2.4 Summary

This chapter has introduced the key background research that forms the grounding for the work presented in this thesis. When analysing tasks in isolation a sound WCET for each task can be calculated and can be done in such a way that the effects of caches are also accounted for. Schedulability analysis can then be used to determine if all of the tasks when running on the system will meet their deadlines. However, the schedulability analysis assumes that the tasks' WCET obtained in isolation will not be affected when scheduling multiple tasks pre-emptively. While this assumption is valid for simple architectures, it is not for more complex ones that contain performance enhancing features such as cache. In the next chapter, we look at existing work that uses information from static analysis and scheduling information to determine the schedulability of a system, accounting for the effects of cache when scheduling multiple tasks pre-emptively.

# CHAPTER 3. CACHE RELATED PRE-EMPTION DELAYS

In this chapter, we describe *cache related pre-emption delays* (CRPD), and review the current state-of-the art analysis for accounting for CRPD when performing schedulability analysis. We also discuss a number of techniques that can be used to minimise these delays either by reducing the number of pre-emptions, or reducing/eliminating intra-task cache conflicts. From this point we assume that an accurate model of the processor being used is available, and that the static analysis techniques discussed in Chapter 2 can be applied to our system. We can then assume that we are able to obtain the following properties:

- A sound WCET estimate for each task in isolation
- Correct information about what 'must' and 'may' be in cache at each program point

## 3.1 Cache Related Pre-emption Delays

When a pre-emption occurs there is a mandatory delay introduced by the need to save the state of the current task, decide which task to switch to, and then setup the new task. This delay is known as the *context switch cost* (CSC). As this is a fairly constant cost, it can usually be upper bounded and then *subsumed* into the execution time of the pre-empting task. In other words, in order to perform schedulability analysis on a taskset, the execution time of each task in the system is inflated by a bound on the time taken by the scheduler/operating system to switch to and then back from a task.

In a system with cache after a pre-emption occurs there can be additional costs due to interferences on the cache which affect the pre-empted task(s). This is known as *cache-related pre-emption delay* (CRPD) and it cannot simply be subsumed into the execution time of the pre-empting task without potentially

introducing significant pessimism. This is because CRPD is dependent on the pre-empting and pre-empted tasks and the point of pre-emption. Specifically, it is incurred when a pre-empted task resumes and no longer has the instructions or data that the task was using in cache, because the pre-empting task(s) evicted them from cache. It is therefore difficult to determine the exact CRPD because the delay will not be incurred at once. Instead, CRPD will be incurred as the task uses data and invokes instructions that were evicted by the pre-empting task(s) during the remainder of its execution. In addition to being highly variable, CRPD can be significantly larger than CSC. In a study of a large multicore platform, Bastoni *et al.* [22] found the CSC to be around 5-10μs in the worst case, with variation being down to the number of tasks and scheduling policy which would not be changed at runtime. In comparison, they found the worst-case pre-emption costs to be much greater and more varied than the CSC, specifically they varied between 1-10000μs depending on the cache usage and system load. Figure 3.1 shows an example pre-emption with a small amount of CSC occurring when switching tasks and a large amount of CRPD spread out during the execution of a task after being pre-empted.

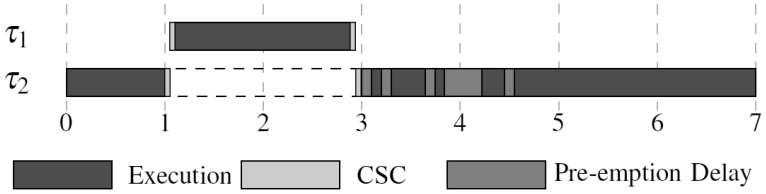



*Figure 3.1 - Illustration of the effects of a pre-emption. CSC are incurred when switching tasks, and pre-emption delays are incurred during the remainder of a tasks execution after pre-emption as it accesses blocks that were evicted from cache during the pre-emption*

As noted, the CSC is fairly constant and can be upper bounded and is therefore usually subsumed into the execution time of the pre-empting task. Figure 3.2 shows a revised version of Figure 3.1 with the CSC replaced by an increase to the execution time of task $\tau_1$.

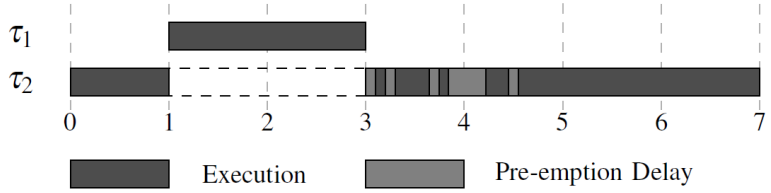



*Figure 3.2 - Illustration of how the CSC can be subsumed into the execution time of the pre-empting task when compared to Figure 3.1*

CRPD depends on the point at which a task is pre-empted. For example, pre-empting a task when it has not loaded anything into cache, or when it no longer requires anything it has in cache will have minimal effects. Figure 3.3 is taken from [28] and is based on Matlab automotive code that models an automatic transmission controller. Pre-emption points were placed at fixed points and a high priority task which evicts all cache lines was used. The plotted CRPD at each point in the figure below was calculated by taking the difference in the execution time with and without pre-emption.

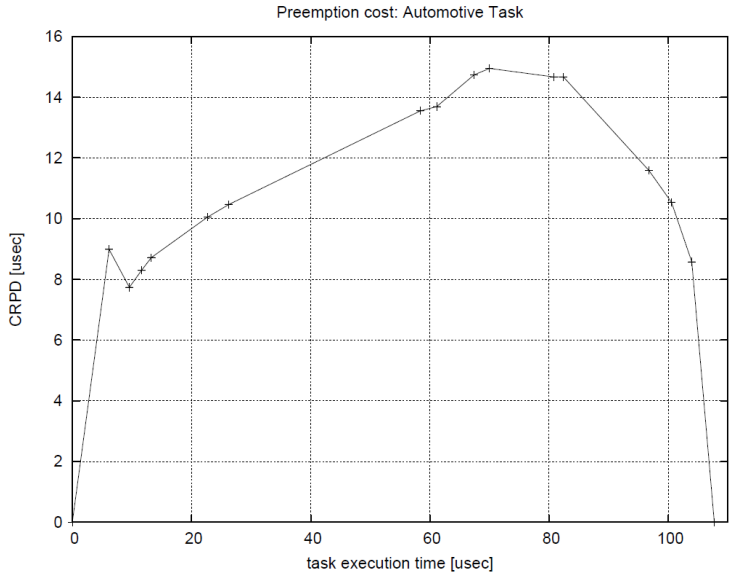



*Figure 3.3 - Example showing CRPD can vary throughout the execution of a task as the maximum amount of CRPD is related to the amount of useful information that has to be re-loaded back into cache. Example taken from [28]*

Furthermore, if a task is pre-empted shortly after resuming from a pre-emption, it may not have yet re-loaded all of the evicted blocks and will therefore not be able to incure the maximum CRPD from the first pre-emption. However, without knowing the exact point at which a task is pre-empted, we must make the pessimistic assumption that the pre-emption will result in the maximum CRPD being incured directly after the pre-emption. This results in a simplified

representation of the CRPD whereby it is combined into a single cost pre-emption as shown in Figure 3.4.

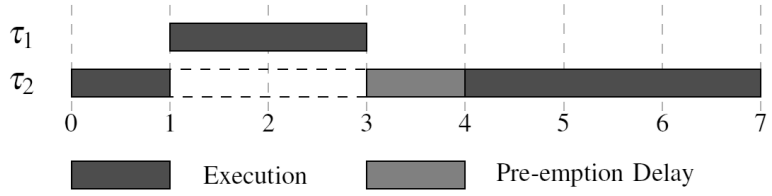



*Figure 3.4 - Simplified and potentially pessimistic representation of CRPD, assuming it is incurred at once after a task resumes*

The analysis presented in this thesis does not consider blocking due to shared resources. However, we note that the effect of CRPD when using shared resources via SRP [16] can be accounted for as shown in [6] [7].

In order to determine an upper bound on the CRPD, we must calculate how many blocks may be evict from cache that then need to be reloaded, and then multiply that by the additional time incurred when reloading a block from memory.

### 3.1.1 Block Reload Time

The additional time taken to reload a block from memory into cache after a pre-emption is dependent on the hardware and is known as the *block reload time* (BRT). There are three possible cases of processor architecture [127]. For processors than employ a simple architecture that does not suffer from timing anomalies such as the *ARM7*, this is simply the difference in the number of cycles to load a block from cache verses from memory. If timing anomalies are possible but not domino effects, for example *TriCore*, then the BRT can be increased to include any additional time that may be incurred as a result of a cache miss. If timing anomalies and domino effects are possible in the architecture, for example *PPC 755*, then the effects of a cache miss cannot be constant bounded. Therefore the effects of CRPD cannot be calculated separately [109]. In this work we assume that the BRT can be determined and that there are no domino effects.

### 3.1.2 UCBs and ECBs

To calculate the number of blocks that must be reloaded, CRPD analysis uses the concept of *useful cache blocks* (UCBs) and *evicting cache blocks* (ECBs) based on the work by Lee *et al.* [77]. Any memory block that is accessed by a task while executing is classified as an ECB, as accessing that block may evict a cache block of a pre-empted task. Out of the set of ECBs, some of them may also be UCBs. A memory block $m$ is classified as a UCB at program point $\rho$, if (i) $m$ may be cached at $\rho$ and (ii) $m$ may be reused at program point $q$ that may be reached from $\rho$ without eviction of $m$ on this path. In the case of a pre-emption at program point $\rho$, only the memory blocks that are (i) in cache and (ii) will be reused, may cause additional reloads. The maximum possible pre-emption cost for a task is determined by the program point with the highest number of UCBs. For each subsequent pre-emption, the program point with the next smallest number of UCBs could be considered. In this thesis, we assume that the set of UCBs and ECBs can be obtained via static analysis.

We represent the set of UCBs and ECBs as a set of integers with the following meanings:

$$s \in \mathrm{UCB}_i \Leftrightarrow \tau_i \text{ has a useful cache block in cache set } s$$

$$s \in \mathrm{ECB}_j \Leftrightarrow \tau_j \text{ may evict a cache block in cache set } s$$

Depending on the approach used, CRPD analysis combines the UCBs belonging to the pre-empted task(s) with the ECBs of the pre-empting task(s). Using this information the total number of blocks that are evicted, which must then be reloaded after the pre-emption, can be calculated and combined with the cost of reloading a block, the BRT, to then give the CRPD. We could therefore calculate an upper bound on the cost of task $\tau_j$ directly pre-empting $\tau_i$ as $\mathrm{BRT} \bullet \left|\mathrm{UCB}_i \cap \mathrm{ECB}_j\right|$. However, note that it could be optimistic in the case of nested pre-emptions and thus cannot be used directly.

As an example, let $\mathrm{UCB}_2 = \{2,3,4,5\}$, $\mathrm{ECB}_1 = \{3,4,5,6,7,8,9\}$ and BRT=1. An upper bound on the CRPD due to a job of task $\tau_1$ directly pre-empting a job of task $\tau_2$ once is then given by:

$$
\begin{aligned}
&= \mathrm{BRT} \bullet \left| \mathrm{UCB}_2 \cap \mathrm{ECB}_1 \right| \\
&= \mathrm{BRT} \bullet \left| \{2,3,4,5\} \cap \{3,4,5,6,7,8,9\} \right| \\
&= 1 \bullet \left| \{3,4,5\} \right| \\
&= 1 \bullet 3 \\
&= 3
\end{aligned}
$$

We use the term *cache utilisation* to describe the ratio of the total size of the tasks to the size of the cache. A cache utilisation of 1 means that the tasks fit exactly in the cache, whereas a cache utilisation of 5 means the total size of the tasks is 5 times the size of the cache.

We focus on instruction only caches. In the case of data caches, the analysis would either require a write-through cache or further extension in order to be applied to write-back caches. We also assume that tasks do not share any code.

### Set-Associative Caches

In the case of set-associative LRU[1] caches, a single cache set may contain several UCBs. For example, $\mathrm{UCB}_1 = \{2,2,4\}$ means that task $\tau_1$ has two UCBs in cache set 2 and one UCB in cache set 4. As one ECB suffices to evict all UCBs of the same cache set, multiple accesses to the same set by the pre-empting task do not appear in the set of ECBs. A bound on the CRPD in the case of LRU caches due to task $\tau_j$ directly pre-empting $\tau_i$ is thus given by substituting the intersection between a set of UCBs and ECBs, $\mathrm{UCB}_i \cap \mathrm{ECB}_j$, with a modified version, $\mathrm{UCB}_i \cap' \mathrm{ECB}_j$. Where $\mathrm{UCB}_i \cap' \mathrm{ECB}_j = \left\{ m \mid m \in \mathrm{UCB}_i : m \in \mathrm{ECB}_j \right\}$ and the result is a multiset that contains each element from $\mathrm{UCB}_i$ if it is also in $\mathrm{ECB}_j$. A precise computation of CRPD in the case of LRU caches is given in Altmeyer *et al.* [9]. The equations provided in this thesis can be applied to set-associative LRU caches with the above adaptation to the set-intersection.

### Definitely-Cached UCBs

During timing analysis, a memory blocks may not be classified as a cache hit or a cache miss and is contained within the set of cache blocks derived through may analysis. In this case the block could be categorised as a UCB, but would also be counted as a cache miss by the timing analysis for the purpose of

[1] The concept of UCBs and ECBs cannot be applied to the FIFO or PLRU replacement policies as shown by Burguière *et al.* [35]

calculating the task's WCET. This could lead to additional pessimism when performing CRPD analysis. Altmeyer *et al.* [3] introduced the concept of *definitely-cached UCBs*, or DC-UCBs, to solve this problem. They extend the original UCBs definition with a third requirement to give:

A memory block $m$ is classified as a DC-UCB at program point $\rho$, if (i) $m$ may be cached at $\rho$ and (ii) $m$ may be reused at program point $q$ that may be reached from $\rho$ without eviction of $m$ on this path, and (iii) $m$ is considered a hit at program point $q$ by the timing analysis.

By restricting the set of UCBs to just those considered as a hit by the timing analysis, the number of UCBs can be reduced which leads to a tighter bound on the CRPD. In practice using DC-UCBs could lead to an under estimation in the CRPD analysis however it would always be accompanied by an equal or greater overestimation in the WCET estimate from the timing analysis. This occurs when a memory block that could not be categorised by the static analysis is actually a UCB and would actually result in a cache hit without pre-emption. However, the static analysis will assume the worst case, a cache miss, in the event that it cannot categorise an access to a block. Therefore, any unaccounted CRPD that may be introduced by a pre-emption would have already been accounted for, as an assumed cache miss, during the WCET analysis.

In this thesis, we use the more precise DC-UCB definition when referring to UCBs. Additionally, the UCB data presented in later chapters for comparing approaches was collected using DC-UCB analysis.

## 3.2 CRPD Analysis for FP Scheduling

In this section, we review existing approaches for calculating CRPD when performing schedulability analysis for FP scheduling. To account for the CRPD when determining the schedulability of a taskset, a component $\gamma_{i,j}$ is introduced into the response time analysis equation for FP, equation (2.1) , where $\gamma_{i,j}$ represents the cost of a single pre-emption of task $\tau_i$ by task $\tau_j$. This gives a revised equation for $R_i$ as:

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil (C_j + \gamma_{i,j}) \tag{3.1}$$

Note that the analysis effectively determines the response time via a busy period calculated based on a synchronous release of tasks. However, it also

assumes that the maximum number of pre-emptions could occur. This is not possible with a synchronous release of tasks and is thus a slightly pessimistic assumption.

Note that once we include CRPD in the schedulability analysis, the effectiveness of priority assignments used under FP are changed. For example Audsley's OPA algorithm has a number of conditions [14] such as requiring the schedulability of a task to not be dependent on the relative priority ordering of higher priority tasks. When considering CRPD, this condition no longer holds. Deadline Monotonic and Rate Monotonic are optimal assignments assuming negligible pre-emption costs under constrained and implicit deadline tasks respectively. However, once CRPD is considered, they are no longer optimal in the general case [53], as shown in Figure 3.5.


*Figure 3.5 - Example schedule demonstrating that Deadline Monotonic is not optimal when CRPD is considered. a) Shows three tasks scheduled under FP with priorities assigned using Deadline Monotonic priority order. Due to the pre-emption and resulting pre-emption delay, task $\tau_3$ misses its deadline. b) Shows the same tasks with the priorities of task $\tau_1$ and $\tau_2$ swapped. In this case the pre-emptions that resulted in pre-emption delays are avoided, and all tasks meet their deadlines*

We define aff($i$,$j$) = hep($i$) ∩ lp($j$) (based on the notation presented in Section 2.1.2) to mean all tasks that can have CRPD caused by task $\tau_j$ pre-empting them, which affects the response time of task $\tau_i$. In other words, it is the set of tasks that may be pre-empted by task $\tau_j$ and have at least the priority of task $\tau_i$.

There are then a number of approaches that have been developed in order to calculate $\gamma_{i,j}$ which we will now briefly summarise.

### ECB-Only

Busquets *et al.* [37] in 1996 presented their ECB-Only approach which considers just the pre-empting task. It captures the worst case effect of task $\tau_j$ pre-empting any task regardless of that task's UCBs, by assuming that every block evicted by task $\tau_j$ will have to be reloaded.

$$\gamma_{i,j}^{ecb} = \text{BRT} \bullet \left| \text{ECB}_j \right| \tag{3.2}$$

### UCB-Only

In 1998, Lee *et al.* [77] presented the UCB-Only approach, which considers just the pre-empted task(s). The UCB-Only approach accounts for nested pre-emptions by calculating the maximum number of UCBs that may need to be reloaded by *any* task that may be directly pre-empted by task $\tau_j$.

$$\gamma_{i,j}^{ucb} = \text{BRT} \bullet \max_{\forall k \in \text{aff}(i,j)} \left\{ \left| \text{UCB}_k \right| \right\} \tag{3.3}$$

The disadvantage of the ECB-Only and UCB-Only approaches is that they only consider either the pre-empting tasks or the pre-empted tasks. The following approaches aim to solve this problem by combining UCBs and ECBs from the pre-empted and pre-empting tasks. However, as previously noted we cannot simply take the intersection of the pre-empting task's ECBs with the pre-empted task's UCBs as this would be optimistic in the case of nested pre-emptions.

### UCB-Union

In 2007 Tan and Mooney [115] considered both the pre-empted and pre-empting task(s) in their UCB-Union approach. UCB-Union accounts for the effects of nested pre-emptions by assuming that the UCBs of any tasks that could be pre-empted, including nested pre-emptions, by task $\tau_j$ are evicted by the ECBs of task $\tau_j$.

$$\gamma_{i,j}^{ucb-u} = \text{BRT} \bullet \left| \left( \bigcup_{\forall k \in \text{aff}(i,j)} \text{UCB}_k \right) \cap \text{ECB}_j \right| \tag{3.4}$$

### ECB-Union

Altmeyer *et al.* [6] presented their ECB-Union approach in 2011 which compliments Tan and Mooney's UCB-Union approach. It accounts for nested pre-emptions by computing the union of all ECBs that may affect a pre-empted

task. The reasoning behind the approach being that a direct pre-emption by task $\tau_j$ is represented by the pessimistic assumption that task $\tau_j$ has itself already been pre-empted by all of the tasks with a higher priority. Hence, a pre-emption by task $\tau_j$ may result in the eviction of $\bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h$. The maximum number of blocks that may be evicted as a result of an already nested pre-emption by task $\tau_j$ is then obtained by considering the maximum number of UCBs that may need to be reloaded by *any* task that may be directly pre-empted by task $\tau_j$, as in the UCB-Only case.

$$\gamma_{i,j}^{ucb-u} = \text{BRT} \bullet \max_{\forall k \in \text{aff}(i,j)} \left\{ \left| \text{UCB}_k \cap \left( \bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \right\} \tag{3.5}$$

### 3.2.1 Multiset Approaches

The approaches presented thus far all calculate the CRPD due to a single pre-emption of task $\tau_i$ by task $\tau_j$. However, calculating the pre-emption costs this way can introduce additional pessimism when there are nested pre-emptions. The approaches effectively assume that task $\tau_j$ can pre-empt each intermediate task $\tau_k$ the same number of times that it pre-empts task $\tau_i$. While this is potentially true if $D_k = D_i$, it can be a pessimistic assumption when $D_k < D_i$ and particularly when $D_k << D_i$.

The remainder of the approaches take a different approach by calculating the CRPD due to all jobs of task $\tau_j$ executing within the response time of task $\tau_i$. They do so by using *multisets* which are unordered collections of elements which can contain the same element multiple times. For example, a multiset can be used to represent the costs of all possible pre-emptions. The total CRPD could then be bounded by calculating how many pre-emptions could occur as $q$, and then taking the sum of the $q$ largest values from the multiset.

#### Staschulat

Staschulat *et al.* [113] in 2005 took a different approach towards combining pre-empted and pre-empting task(s). The analysis accounts for the fact that each additional pre-emption of task $\tau_i$ may result in a smaller pre-emption cost than the last. In order to integrate their approach into the response time analysis we use $\gamma'_{i,j}$ to represent the total cost of all pre-emptions due to jobs of task $\tau_j$ executing within the response time of task $\tau_i$. The approach is integrated into the response time analysis equation for FP, equation (2.1), to give:

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left( \left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil C_j + \gamma'_{i,j} \right) \tag{3.6}$$

In order to present Staschulat *et al.* approach, we define the maximum number of jobs of task $\tau_k$ that can execute during the response time of task $\tau_i$, $E_k(R_i)$ as:

$$E_k(R_i) = \left\lceil \frac{R_i}{T_k} \right\rceil \tag{3.7}$$

The first step of Staschulat *et al.* approach is to form a multiset, $M$, containing the cost of each possible pre-emption of task $\tau_j$ pre-empting jobs of any lower priority task $\tau_k \in \mathrm{aff}(i, j)$ that can execute during the response time of task $\tau_i$. $M$ is given by:

$$M = \bigcup_{\forall k \in \mathrm{aff}\ (i,j)} \left( \bigcup_{E_k(Ri)} \left\{ \left( \mathrm{UCB}_k \cap \mathrm{ECB}_j \right)^n \mid n \in \left[ 1; E_j(R_k) \right] \right\} \right) \tag{3.8}$$

where $\left( \mathrm{UCB}_k \cap \mathrm{ECB}_j \right)^n$ gives the $n$-th highest pre-emption cost for task $\tau_j$ pre-empting task $\tau_k$. As $M$ is a multiset, the union over $E_k(R_i)$ means that the set of values for task $\tau_k$ are repeated $E_k(R_i)$ times in $M$.

The next step is to calculate the maximum number of pre-emptions $q$, including nested-pre-emptions, from the set of tasks $\tau_k \in \mathrm{aff}(i, j)$ that can execute during the response time of task $\tau_i$:

$$q = \sum_{\forall k \in \mathrm{aff}\ (i,j)} E_k(R_i) \tag{3.9}$$

The total CRPD due to all pre-emptions due to jobs of task $\tau_j$ executing within the response time of task $\tau_i$ is then given by the sum of the $q$ largest pre-emptions.

$$\gamma'^{\,sta}_{i,j} = \mathrm{BRT} \bullet \sum_{l=1}^{q} \left| M^l \right| \tag{3.10}$$

where $M^l$ is the $l$-th largest element from the multiset $M$.

However, as shown in [7], this approach can significantly over-estimate the number of pre-emptions that can affect the response time of the pre-empted task, especially when there are a large number of tasks.

### UCB-Union Multiset

In 2012, Altmeyer *et al.* [7] presented their UCB-Union Multiset approach which combines the UCB-Union approach with Staschulat *et al.* [113] method of counting the maximum number of pre-emptions incurred by intermediate tasks. The first step is to form a multiset $M_{i,j}^{ucb}$ containing $E_j(R_k)E_k(R_i)$ copies of the $\text{UCB}_k$ of each task $\tau_k \in \text{aff}(i, j)$. This multiset reflects the fact that jobs of task $\tau_j$ cannot evict the UCBs of jobs of task $\tau_k$ more than $E_j(R_k)E_k(R_i)$ times during the response time $R_i$ of task $\tau_i$. Hence:

$$M_{i,j}^{ucb} = \bigcup_{\forall k \in \text{aff}(i,j)} \left( \bigcup_{E_j(R_k)E_k(R_i)} \text{UCB}_k \right) \tag{3.11}$$

To represent the pre-empting tasks, Altmeyer *et al.* form a multiset $M_{i,j}^{ecb}$ containing $E_j(R_i)$ copies of the $\text{ECB}_j$ of task $\tau_j$. This multiset reflects the fact that during the response time $R_i$ of task $\tau_i$, task $\tau_j$ can evict cache blocks in the set $\text{ECB}_j$ at most $E_j(R_k)E_k(R_i)$ times.

$$M_{i,j}^{ecb} = \bigcup_{E_j(R_i)} \left( \text{ECB}_j \right) \tag{3.12}$$

$\gamma'^{\,ucb-m}_{i,j}$ is then given by the size of the multiset intersection between $M_{i,j}^{ucb}$ and $M_{i,j}^{ecb}$:

$$\gamma'^{\,ucb-m}_{i,j} = \text{BRT} \bullet \left| M_{i,j}^{ucb} \cap M_{i,j}^{ecb} \right| \tag{3.13}$$

### ECB-Union Multiset

Altmeyer *et al.* [7] also presented the ECB-Union Multiset approach which builds upon the ECB-Union approach. It computes the union of all ECBs that may affect a pre-empted task during a pre-emption by task $\tau_j$. Specifically, it accounts for nested pre-emptions by assuming that task $\tau_j$ has already been pre-empted by all tasks of a higher priority.

The first step is to calculate the number of UCBs that task $\tau_j$ could evict when pre-empting an intermediate task, $\tau_k$. This is given by calculating the intersection of the UCBs of the pre-empted task, task $\tau_k$, with the set of ECBs belonging to the pre-empting tasks $\bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h$ to give:

$$\left| \text{UCB}_k \cap \left( \bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \tag{3.14}$$

Note $h \in \text{hp}(j) \cup \{j\}$ is used to account for the case when tasks can share priorities.

The ECB-Union multiset approach recognises that task $\tau_j$ cannot pre-empt each intermediate task $\tau_k$ more than $E_j(R_k)E_k(R_i)$ times during the response time of task $\tau_i$. Therefore, the next step is to form a multiset $M_{i,j}$ that contains the cost of task $\tau_j$ pre-empting task $\tau_k$ , equation (3.14), repeated $E_j(R_k)E_k(R_i)$ times, for each task $\tau_k \in \text{aff}(i, j)$, hence:

$$M_{i,j} = \bigcup_{\forall k \in \text{aff}(i,j)} \left( \bigcup_{E_j(R_k)E_k(R_i)} \left| \text{UCB}_k \cap \left( \bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \right) \tag{3.15}$$

As only $E_j(R_i)$ jobs of task $\tau_j$ can execute during the response time of task $\tau_i$, the maximum CRPD is obtained by summing the $E_j(R_i)$ largest pre-emptions, the $E_j(R_i)$ largest values in $M_{i,j}$ .

$$\gamma'^{\,ecb-m}_{i,j} = \text{BRT} \bullet \sum_{l=1}^{E_j(R_i)} \left| M^l_{i,j} \right| \tag{3.16}$$

**Combined Multiset**

Altmeyer *et al.* [7] presented a further improvement to their multiset approaches by recognising that the UCB-Union Multiset and ECB-Union Multiset approaches are incomparable. Because of this, they can be combined to deliver a more precise bound that by construction dominates the use of either approach alone. Note that some tasksets can be deemed schedulable by the combined approach that would not be deemed by either approach individually. This is because the response time for each task can be individually determined using either approach.

$$R_i = \min\left(R_i^{ucb-m}, R_i^{ecb-m}\right) \tag{3.17}$$

## 3.3 CRPD Analysis for EDF Scheduling

In this section, we review an existing approach for calculating CRPD when performing schedulability analysis for EDF scheduling. The EDF scheduling always schedules the job with the earliest absolute deadline first. Assuming negligible pre-emption costs, it is an optimal scheduling algorithm for a single processor. Any time a job arrives with an earlier absolute deadline than the current running job, it will pre-empt the current job. When a job completes its

execution, the EDF scheduler chooses the pending job with the earliest absolute deadline to execute next. In the case where two or more jobs have the same absolute deadline, we assume the scheduler always picks the job belonging to the task with the lowest unique task index, see Figure 3.6. This has the benefit of minimising the number of pre-emptions. In the case where two task jobs have the same absolute and relative deadlines, it ensures that they cannot pre-empt each other. Furthermore, it ensures that after a pre-emption, the task that was pre-empted last is resumed first.

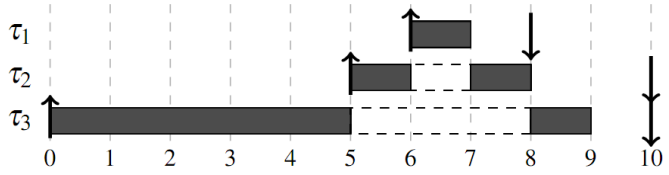



*Figure 3.6 - Example schedule showing how the scheduler chooses which task should execute. Task $\tau_3$ is released at t = 0. At t = 5, task $\tau_2$ is released, pre-empting $\tau_3$ as although it has the same absolute deadline, it has a lower task index. At t = 6, task $\tau_1$ is released, pre-empting task $\tau_2$. At t = 7, $\tau_1$ completes, the scheduler then chooses to resume task $\tau_2$ as although it has the same absolute deadline as task $\tau_3$, it has the lower task index*

We note that when CRPD is taken into account, EDF is no longer optimal in the general case. Consider the following example with two tasks shown in Figure 3.7. The first schedule a) shows three tasks scheduled under EDF. Due to the pre-emption and resulting pre-emption delay, task $\tau_3$ misses its deadline. The second schedule b) shows the same tasks scheduled under FP with priorities assigned $\tau_2$, $\tau_1$, $\tau_3$, highest to lowest. In this case the pre-emptions that resulted in pre-emption delays are avoided, and all tasks meet their deadlines.


*Figure 3.7 - Example schedule showing that EDF is not optimal when CRPD is considered. a) Shows three tasks scheduled under EDF. Due to the pre-emption and resulting pre-emption delay, task $\tau_3$ misses its deadline. b) Shows the same tasks scheduled under FP with priorities assigned $\tau_2, \tau_1, \tau_3$. In this case the pre-emptions that resulted in pre-emption delays are avoided, and all tasks meet their deadlines*

We assume that any task $\tau_j$ with a relative deadline $D_j < D_i$ can pre-empt task $\tau_i$. Therefore, we define the set of tasks that may have a higher priority, and can pre-empt task $\tau_i$, as:

$$hp(i) = \{\tau_j \in \Gamma \mid D_j < D_i\} \tag{3.18}$$

We use $P_j(D_i)$ to denote the maximum number of times that jobs of task $\tau_j$ can pre-empt a single job of task $\tau_i$ which we calculate as follows:

$$P_j(D_i) = \max\left(0, \left\lceil \frac{D_i - D_j}{T_j} \right\rceil\right) \tag{3.19}$$

We use $E_j(t)$ to denote the maximum number of jobs of task $\tau_j$ that can have both their release times and their deadlines in an interval of length $t$, which we calculate as follows:

$$E_j(t) = \max\left(0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor\right) \tag{3.20}$$

### JCR Approach

There has been little work towards integrating CRPD analysis into schedulability tests for EDF. To the best of our knowledge, the only existing work on integrating CRPD analysis with EDF schedulability tests was

developed by Ju *et al.* [71] in 2007. We refer to this approach as the JCR approach after the initials of the authors' names. The JCR approach calculates the number of blocks evicted due to task $\tau_j$ directly pre-empting task $\tau_i$ multiplied by the number of times that pre-emption could occur, $P_j(D_i)$. This is repeated for each task that could pre-empt task $\tau_i$ and summed up. Using our notation, this gives the CRPD associated with task $\tau_i$ being pre-empted as follows:

$$\gamma_i^{jcr} = \mathrm{BRT} \bullet \sum_{j \in \mathrm{hp}(i)} P_j(D_i) \left| \mathrm{UCB}_i \cap \mathrm{ECB}_j \right| \tag{3.21}$$

$\gamma_i$ can then be integrated into the processor demand bound function, equation (2.2), to give:

$$h(t) = \sum_{i=1}^{n} \max\left\{0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right\} \left(C_i + \gamma_i^{jcr}\right) \tag{3.22}$$

One source of pessimism in this approach is how it deals with nested, or indirect, pre-emptions. It always defines the CRPD between a pair of tasks and adds them together. For example, if during the pre-emption of task $\tau_i$ by task $\tau_j$, task $\tau_j$ was itself pre-empted by task $\tau_k$ the JCR approach calculates $\gamma_i$ to be the sum of the pre-emptions. However, unless $\mathrm{ECB}_j \cap \mathrm{ECB}_k = \emptyset$, the analysis could pessimistically calculate that some UCBs are evicted multiple times. In Chapter 4, we present a number of approaches for calculating CRPD under EDF scheduling and compare them to the JCR approach.

## 3.4 Limiting Pre-emptions

Recent work towards analysing CRPD has improved yet the fact that tasks can be pre-empted at any point in their execution leads to increased pessimism when considering the worst case pre-emptions. In this section, we briefly review a number of methods that aim to limit pre-emptions. In 2011 Bertogna [28] described an approach which extends previous work which he proposed in 2010 [27] with a goal to calculate the CRPD by ensuring that tasks can only be pre-empted at known points. This builds on work into *co-operative scheduling* from 1994 by Burns [36]. In Bertogna's work, he defined these known points as *fixed pre-emption points* (FPP) which allow for the pre-emption cost to be calculated while not significantly blocking the pre-empting task. It requires the programmer to define a set of potential pre-emption points during design time. The algorithm then selects pre-emption points to minimise the overall pre-

emption cost. A notable improvement of [28] over [27] is that the new approach can deal with the fact that the pre-emption cost varies at different points in the task.

Buttazzo *et al.* [39] in 2012 presented a survey of techniques that limit pre-emption. In addition to FPP from Bertogna [28], it also included *pre-emption thresholds scheduling* (PTS) [118], and *deferred pre-emptions scheduling* (DPS) [17] [48]. DPS allows a task to run for a period of time without being pre-empted up to a certain limit. Alternatively, PTS introduces an additional parameter to control the balance between fully pre-emptive scheduling, and non-pre-emptive scheduling. The pre-emption threshold allows a task to disable pre-emption by higher priority tasks, up to a certain priority. Out of these techniques, using FPP resulted in the most predictable system and seems most promising. However, as previously discussed, this approach requires determining and adding these points to the code. The problem becomes even less trivial when loops with large number of iterations, or branches with large variations in the number of instructions are involved. If the pre-emption points are not placed carefully, the time between possible pre-emptions could be either too long or too short depending on the path taken through the code. Recent work by Bo *et al*. [33] aimed to address these limitations and support branches, conditional statements and loops. They proposed a pseudo-polynomial-time algorithm for determining the optimal set of pre-emption points by operating on the CFG. However, the analysis became prohibitively expensive in terms of memory requirements and runtime, so the authors also proposed a near-optimal heuristic. Nevertheless, accurate CRPD analysis is still crucial as there will always be some pre-emptions.

## 3.5 Improving Cache Predictability

We now discuss a number of techniques that can be used to improve the predictability of cache, which in turn increases our ability to analyse it. The key challenge with improving cache predictability effectively is to maximise the useful information in cache. Some of the key techniques include *cache partitioning*, *cache locking*, *static code positioning*, or a very different approach of using a *scratchpad* instead of a traditional cache.

### Cache Partitioning

Cache partitioning [89] [98] [73] is a technique that can be used to reduce or eliminate intra-task interference by splitting the cache into a number of

partitions and allocating tasks to the partitions. For example, each task can be allocated its own partition in the cache so that it cannot interfere with the cache contents belonging to other tasks in the system. However, the reduced cache size per task can result in increased WCET through increased inter-task interference. Ideally, this is implemented using either a cache that can be locked on a way-by-way basis. However, if that is not possible then it can be achieved by using a compiler with specific support. Recent work by Altmeyer *et al.* [8] has investigated the performance of a partitioning architecture with no CRPD versus a traditional cache analysed using state-of-the-art CRPD analysis. They found that the increased predictability of a partitioned cache, in terms of eliminating CRPD, does not compensate for the performance degradation in the WCETs due to the smaller cache space per task. Cache partitioning can be implemented in hardware in some systems however, in most caches it requires specific compiler support in order to ensure each task is confined to its partition.

### Cache Locking

Cache locking is an alternative technique where a part, or the whole of the cache, is locked in order to fix the cache contents using specific hardware support in the cache. Accesses that result in a cache hit will be served as normal, while accesses that result in a cache miss will be served from memory but will not result in the cache being updated. Cache locking was first tackled in [41] by Campoy *et al.* in 2001. They used a genetic algorithm to find which blocks should be locked in cache. One of the key challenges with cache locking is that if a block is not on the WC path, then locking it into cache will not reduce the WCET. However, just selecting blocks that are on the WC path initially is not enough, because the WC path can change as the execution times of those blocks decreases. As with cache partitioning, cached locking also reduces or eliminates CRPD at the expense of a potentially increased WCET. In addition to the effort required to determine what should be locked into cache and when to do so, additional code must be added to the system in order to lock and unlock the cache.

### Static Code Positioning

Static code positioning uses a shared cache, but positions procedures/functions and/or tasks in memory such that the layout in cache results in reduced inter or intra-task interference, depending on the target of the optimisation. Unlike cache partitioning, static code positioning does not restrict the available cache

size that each task can make use of. Positioning tasks can usually be implemented by controlling how object files are combined at the linker/locator stage of compilation. However, positioning procedures or functions often requires specific compiler support unless each procedure or function can be compiled into a separate object file.

Out the above techniques, this thesis focuses on concepts behind static code positioning, which are discussed in detail in Section 3.5.1.

**Scratchpads**

Scratchpads are small fast memories like cache, but are directly addressable and occupy a distinct part of the memory address space. Scratchpads must be managed directly though the software, either by the programmer or by a compiler with specific support. The contents of the scratchpad are assigned prior to runtime and can remain constant as described by Suhendra *et al.* [114]. Alternatively, the contents can also be dynamically modified during runtime as in Wehmeyer and Marwedel [120]. Scratchpads are also suited to storing temporary results that do not reside in main memory.

Scratchpads are especially beneficial in multi-core systems as using them avoids contention for access to the slower main memory. Because there is no uncertainty over whether instructions or data will reside in the scratchpad, there is no uncertainty about the access time. This makes calculating a tight WCET estimate much easier. In 2009, Whitham first described a *scratchpad memory management unit* (SMMU) in [122], [123] and [121] that "combines the *address transparency* of a cache with the *time-predictability* property of a scratchpad" [122]. An OPEN operation can be issued to the SMMU which will cause it to map an area in the logical address space to the scratchpad. The SMMU will then copy the contents from external memory to the scratchpad. Any accesses to memory in that area will be transparently translated to use the scratchpad. A CLOSE operation can then be issued to reverse to process.

Recent work by Whitham *et al.* [125] [124] has introduced the concept of *explicitly reservation* whereby when a task is pre-empted, the state of the cache is saved, and is later restored when the task resumes.

### 3.5.1 Static Code Positioning

*Static code positioning*, also known as *code layout* techniques, can be used to reduce the task execution times by statically ensuring that the code is laid out in

its optimum configuration. This is achieved at the linker/locator stage of the code compilation and uses information about the cache, its associativity and the memory to position the code for optimum performance. An example from [86] is shown in Figure 3.8 that demonstrates the conflicts between procedures and the resultant evictions if they are not positioned optimally.

Static code positioning techniques were originally investigated to help decrease the *average-case execution time* (ACET). While ACET centric optimisations do not usually help improve the WCET, they formed the base for much of the WCET orientated code positioning work. In 2004/2005, Zhao *et al.* [131] [130] were first to apply code positioning techniques in order to try to reduce the WCET. However, their processor did not have a cache and they focused on reducing pipeline stalls. This work focused on reordering basic blocks in order to reduce branch penalties along the WC path. They used static WCET analysis to drive their optimisation. They also re-ran the WCET analysis after every modification to the block positions to account for the fact that the WC path can switch.

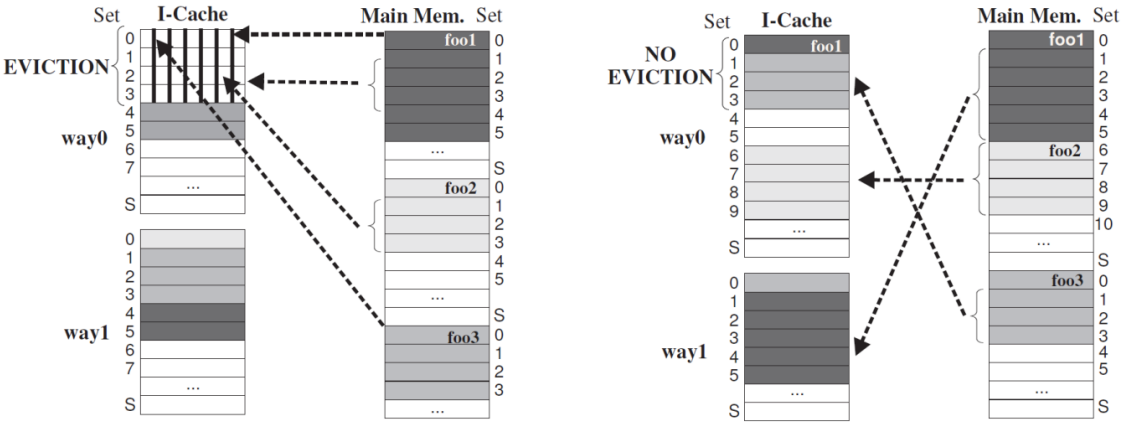



*Figure 3.8 – Illustration of how controlling procedure positions can reduce cache conflicts. Reproduced from [86]*

Lokuciejewski *et al.* in 2008 [86] were the first to try to reduce the WCET with respect to cache. They presented two different approaches that perform procedure positioning, a greedy algorithm and a fast heuristic. Both approaches use a call graph where the edges contain the call frequencies between procedures derived from static WCET analysis. The principles are similar to those of Pettis and Hanson [97] as described above. The two nodes that have the heaviest edge connecting them are selected. These nodes are then merged and their edges are coalesced. Again if a node is merged into an already merged node, the original call graph is used to determine the new ordering. Upon making a change, the WCET analysis is performed on the new graph, if the

change results in an increase in the WCET then it is rejected, otherwise it is accepted. The full WCET analysis is performed, to ensure that any changes to the WC path due to it switching are taken into account for the next round of optimisation. This process terminates when only disjointed nodes remain. The authors note that their greedy approach may become stuck in a local minimum as this is a common problem with greedy algorithms. However, this was not the case for their selected benchmarks. They reported up to a 22% reduction in WCET for their benchmarks.

Their fast heuristic is very similar to the greedy algorithm, however the WCET analysis is not re-run after every modification. While this is faster, it can lead to an overall worse WCET as they found in one of their benchmarks, a GSM encoder.

They also presented an approach based on *procedure cloning* which duplicates procedures in memory. This is based on their earlier work in [87]. However, this is more beneficial when static WCET analysis is being performed and it might not have the same benefits if driven by hybrid measurement-based WCET analysis. This is because it enables the overestimation incurred during static WCET analysis, due to being unable to annotate procedures with context dependent information, to be reduced. Examples include loops in procedures that are only iterated 10 times in one context and 100 times in another context. Static analysis has to assume that the loop is always iterated 100 times. Their procedure cloning approach was very successfully, with up to a 65% reduction. However that could be largely down to the less pessimistic WCET analysis. An interesting comparison would be to perform hybrid measurement-based analysis with full path coverage obtained using cache flushing to determine what the actual effect would be. An optimising compiler could take advantage of the procedure cloning to remove unused code from the call context specific procedures, which would then result in improved cache pre-fetching, and less pipeline stalls from branch miss-predictions.

In 2011, Falk *et al.* [59] took into account the cache configuration with the aim of reducing the WCET by minimising cache conflicts. While previous works such as Lokuciejewski *et al* [86] positioned procedures in order to improve cache performance, they did not consider the cache configuration. Factors such as the caches associativity and size were not taken into account. Falk *et al.* used a conflict graph with edges based on cache misses. The information was obtained using static WCET analysis. The aim is to place them contiguously in memory to reduce conflicts. As with previous work, a greedy algorithm was used to

select the nodes that were connected with the heaviest edge. These nodes were then merged and the change was evaluated and only accepted if it resulted in a reduction in the WCET. Additionally, they also rebuilt the conflict graph to ensure they were always optimising the current WC path. First they applied the processes to the basic blocks. Once the process terminates, they applied it to the procedures. Again, their greedy algorithm could be susceptible to becoming stuck in a local minimum, but this did not occur during their tests. One restriction of their work was that it focused on caches with a LRU replacement policy. This was due to the fact that static WCET analysis performs best when analysing LRU caches [105] compared to other less predictable policies, rather than a limitation of their approach.

Gebhard and Altmeyer [66] took an alternative approach in 2007 by using schedulability analysis to evaluate different layouts. They performed their analysis on a pre-emptive multi-tasking system with a goal to prevent pre-empting tasks from evicting the pre-empted tasks blocks from cache by positioning whole tasks contiguously in memory. First they collect performance influencing metrics such as tasks periods, sizes, interdependencies and timing constraints. The layouts are evaluated using a cost function that estimates the number of conflicts caused by a pre-emption. This uses information about the tasks' position in memory and the cache configuration to determine where the tasks are placed in the cache. The cost is proportional to the number of blocks belonging to the pre-empted task that reside in the same location as the pre-empting tasks' blocks. It also takes into account the lifespan of blocks due to the replacement policy. They then found an improved layout using both ILP and a *simulated annealing* (SA). While the ILP found an optimum solution, it suffered from increased complexity. They added an additional constraint that prevented any gaps in the memory in order to reduce the search space. They used a SA to find a non-optimal solution, but in reduced time. The new layouts resulted in up to a 50% decrease in the number of cache misses. However, the number of cache misses did not correlate directly with the values return by the cost function. This was because no consideration was taken for the actual code inside the tasks. If blocks containing loops were positioned so that they were safe from eviction, the overall number of misses is reduced significantly more than for straight line code which is not reused.

## 3.6 Summary

In this chapter we have discussed CRPD and the importance of being able to correctly account for it when determining the schedulability of a system. Specifically, we note that CRPD is dependent on the pre-empting and pre-empted task(s) and cannot simply be subsumed into the execution time of the pre-empting task as is done for traditional context switch costs. Therefore in order to ensure that a system can be scheduled, without simply overprovisioning the hardware, schedulability analysis must account for CRPD. We reviewed the current state-of-the art techniques for calculating CRPD under FP and EDF scheduling. These techniques work by bounding the maximum number of useful blocks that could be evicted from cache during a pre-emption that may need to be reloaded afterwards. We identified a potential source of pessimism in the existing analysis for calculating CRPD under EDF scheduling. In Chapter 4, we present new analysis for calculating CRPD under EDF and compare it to the existing approach.

We also reviewed a number of techniques that can be used to either minimise the number of pre-emptions, or to increase the predictability of the cache. We note that even if the number of pre-emptions is reduced, accurate CRPD analysis is still required. The predictability of caches can be increased by either locking content into cache, or positioning content to minimise interference. However, many of the techniques either require specific hardware or compiler support, which may make them less suitable for industry. Statically positioning tasks can be achieved by controlling the linker which could be applied with relative ease to existing systems. However, it has not yet been used to try to minimise CRPD. In Chapter 5, we present a new technique for positioning tasks so as to increase system schedulability via reduced CRPD.

In the existing work, the focus has been on comparing CRPD analysis under the same scheduling algorithm which makes it difficult to put the effects of CRPD into context. In Chapter 6, we perform a detailed comparison of FP vs EDF when accounting and optimising for CRPD.

Finally, we note that the existing CRPD analysis is designed for systems that have a single FP or EDF scheduler, and are not applicable to systems that use hierarchical scheduling, such as those that employ a partitioned architecture. In Chapter 7 we present new analysis for calculating CRPD when using hierarchical scheduling.

# CHAPTER 4. CRPD ANLAYSIS FOR EDF SCHEDULING

In this chapter we present new CRPD analysis for EDF and compare it to the existing CRPD analysis for EDF. These new analysis methods are based on a number of approaches originally developed for FP, discussed in Section 3.2. Through a series of evaluations, we show that our new approaches can significantly outperform the existing approach for EDF.

For background material on the system model and EDF scheduling see Section 2.1, and for some initial assumptions and definitions required for integrating CRPD analysis into EDF see Section 3.3.

While there has been significant progress towards bounding the effects of CRPD under FP scheduling, as discussed in Section 3.2, there has been little prior work for EDF. This is despite EDF offering improved schedulability over FP scheduling. There is an existing approach for calculating CRPD under EDF by Ju *et al.* [71]. This approach is discussed in Section 3.3 where we note that a source of pessimism in this approach is how it deals with nested, or indirect, pre-emptions. It always defines the CRPD between a pair of tasks and adds them together. As such, if the pre-empting tasks share the same ECBs, then the analysis could pessimistically calculate that some UCBs are evicted multiple times.

EDF is a dynamic scheduling algorithm that always schedules the job of the task with the earliest absolute deadline first. In 1974, Dertouzos [57] proved EDF to be optimal among all scheduling algorithms on a uniprocessor. However, this only applies when there are negligible context switch costs. When CRPD is taken into account, EDF is no longer optimal in the general case as shown in Section 3.3.

# 4.1 Integrating CRPD Analysis into EDF Scheduling

In order to account for CRPD using EDF scheduling, we use a component $\gamma_{t,j}$ which represents the CRPD associated with a pre-emption by a single job of task $\tau_j$ on jobs of other tasks that are both released and have their deadlines in an interval of length $t$. This component $\gamma_{t,j}$ is then included into the processor demand bound function, equation (2.2), so as to calculate the demand on the processor within an interval of length $t$ due to task execution and CRPD. Note, unlike its counterpart in CRPD analysis for FP scheduling, $\gamma_{t,j}$ refers to the pre-empting task $\tau_j$ and $t$, rather than the pre-empting and pre-empted tasks. Including $\gamma_{t,j}$ in equation (2.2) we get our revised equation for $h(t)$:

$$h(t) = \sum_{j=1}^{n} \max\left\{0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor\right\}\left(C_j + \gamma_{t,j}\right) \tag{4.1}$$

Equation (4.1) is evaluated for a bounded number of values of $t$ to ensure that the demand on the processor in an interval of length $t$, $h(t)$, is always $\leq t$. The exact method for determining which values of $t$ need to be checked is described in Section 2.1.2.

In equation (4.1), we are effectively including the CRPD caused by task $\tau_j$ as if it were part of the execution time of task $\tau_j$. Figure 4.1 and Figure 4.2 illustrate the CRPD increasing the execution time of the pre-empted task and modelling it as an increase in the execution time of the pre-empting task respectively.

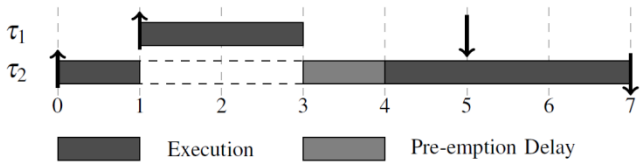



*Figure 4.1 - Including the CRPD caused by $\tau_1$ pre-empting $\tau_2$ in the execution time of $\tau_2$*

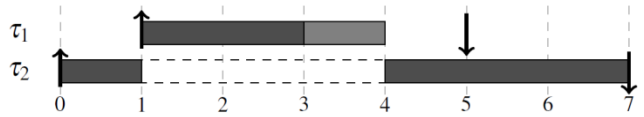



*Figure 4.2 - Representing the taskset in Figure 4.3 by including the CRPD caused by $\tau_1$ pre-empting $\tau_2$ in the execution time of $\tau_1$ which is the approach used in equation (4.1)*

We make use of the approach used to prove theorem 4 in Baruah and Burns [18] to show that if a taskset is deemed schedulable by equation (4.1), Figure 4.2, then the equivalent taskset which it represents, Figure 4.1, is also schedulable.

***Theorem 4.1:*** Let $J = \{(r_v, c_v\ d_v)\}$ denote a collection of independent jobs represented by a release time $r_v$ execution time $c_v$ and absolute deadline $d_v$. Let $S$ be an EDF schedule of $J$. Let $w$ and $x$ be jobs of $J$, such that $r_w \leq r_x$ and $d_w \geq d_x$, i.e. job $x$ is a job that pre-empts job $w$. Let $J'$ be obtained from $J$ by modifying jobs $w$ and $x$ to obtain jobs $y$ and $z$ such that $c_z = c_x - a$ and $c_y = c_w + a$ where $a \leq c_x$. (The release times and absolute deadlines of the jobs in $J'$ are identical to their counterpart jobs in $J$). If $J$ is schedulable by EDF, then so is $J'$.

***Proof:*** $J$ is equivalent to $K$ where $K$ is a set of sub-jobs containing $c_v$ sub-jobs of unit length for each job $v$ in $J$. Each sub-job $q_{v\,q}$ is described by ($r_{vq} = r_v$, $c_{vq} = 1$, $d_{vq} = d_v$). Let $K'$ be a transformation of $K$ such that $a$ sub-jobs $q_{xq}$ have their deadline increased from $d_{xq} = d_x$ to $d_z$. Hence, $K'$ is equivalent to $J'$. As $S$ is a valid schedule for $J$, it is also a valid schedule for $K$. It follows that $S$ is also a valid schedule for $K'$ and hence $J'$. Therefore, the EDF schedule $S$ of $J$ proves the feasibility of $J'$. Since EDF is optimal on pre-emptive uniprocessors, it is therefore guaranteed to successfully schedule $J'$ to meet all deadlines □

We need to define the set of tasks that can be pre-empted by jobs of task $\tau_j$ in an interval of length $t$, aff($t$, $j$). For EDF, this set is based on the relative deadlines of the tasks. We therefore want to capture all of the tasks whose relative deadlines are greater than the relative deadline of task $\tau_j$ giving our initial definition of aff($t$, $j$) as:

$$\mathrm{aff}(t, j) = \{\tau_i \in \Gamma \mid D_i > D_j\} \tag{4.2}$$

However, we can refine this by excluding tasks whose deadlines are larger than $t$ as they do not need to be included when calculating $h(t)$:

$$\mathrm{aff}(t, j) = \{\tau_i \in \Gamma \mid t \geq D_i > D_j\} \tag{4.3}$$

as shown by Theorem 4.2.

***Theorem 4.2:*** When evaluating the processor demand $h(t)$, equation (4.1), for taskset $\Gamma$, the execution requirement of any task $\tau_k$, where $D_k > t$, is not considered. Therefore, we may exclude any contribution to $\gamma_{t,j}$ due to the CRPD incurred by any task $\tau_k$ (where $D_k > t$) as a result of its pre-emption without impacting the soundness of the result.

***Proof:*** We use the proof by Baruah *et al.* [20] that was used to prove that equation (2.2) is necessary. Assume that taskset Γ satisfies equation (4.1) and yet τ is not feasible. Let $S$ be an EDF schedule of Γ where there is a missed deadline. Let $t_2$ be the time of the first missed deadline and let $t_1$ be the last time prior to $t_2$ such that there is no task with a deadline ≤ $t_2$ scheduled at $t_1$ - 1 in $S$. Since the deadline $t_2$ is not met, there is an active task at $t_2$ - 1, so some task must be scheduled at $t_2$ - 1. By definition of $t_1$ it follows that there is a task scheduled at every time in [$t_1$, $t_2$]. By the choice of $t_1$ and $t_2$, only jobs with deadlines ≤ $t_2$ execute during [$t_1$, $t_2$] and all jobs released by tasks with relative deadlines < $t_2$ - $t_1$ = $t$ prior to $t_1$ will have completed by $t_1$. Therefore, as there is a task scheduled at every time in [$t_1$, $t_2$] and the deadline $t_2$ is missed, $h(t_2 - t_1) > t_2 - t_1$, which contradicts our original assumption that Γ satisfies equation (4.1). Note in the case of a missed deadline, no job of a task $\tau_k$ with $D_k > t_2 - t_1$ executes in the interval [$t_1$, $t_2$], hence it is not necessary to include any CRPD arising in such a task □

We now show how a number of existing approaches for calculating CRPD for FP scheduling, discussed in Section 3.2, can be adapted to work with EDF scheduling.

## ECB-Only

We start with the ECB-Only approach by Busquets et al. [37], see equation (3.2) in Section 3.2. It captures the worst case effect of task $\tau_j$ pre-empting any task regardless of that task's UCBs, by assuming that every block evicted by task $\tau_j$ will have to be reloaded. For EDF, ECB-Only is simply:

$$\gamma_{t,j}^{ecb} = \mathrm{BRT} \bullet \left| \mathrm{ECB}_j \right| \tag{4.4}$$

## UCB-Only

The alternative UCB-Only approach by Lee *et al.* [77], see equation (3.3) in Section 3.2, considers just the UCBs of the pre-empted task(s). The UCB-only approach accounts for nested pre-emptions by calculating the maximum number of UCBs that may need to be reloaded by *any* task that may be directly pre-empted by task $\tau_j$. For EDF, this equates to the maximum number of UCBs belonging to any task that can be pre-empted by task $\tau_j$ and can also have a job with a release time and absolute deadline within an interval of length $t$. This set of tasks is given by aff($t$, $j$). Hence we can define the UCB-Only approach for EDF as:

$$\gamma_{t,j}^{ucb} = \mathrm{BRT} \bullet \max_{\forall k \in \mathrm{aff}(t,j)} \left\{ \left| \mathrm{UCB}_k \right| \right\} \tag{4.5}$$

### UCB-Union

The UCB-Union approach of Tan and Mooney [115], see equation (3.4) in Section 3.2, accounts for the effects of nested pre-emptions by assuming that the UCBs of any tasks that could be pre-empted, including nested pre-emptions, by task $\tau_j$ are evicted by the ECBs of task $\tau_j$. When adapting this approach for EDF, we are interested in the UCBs of any tasks that may be pre-empted by task $\tau_j$ and can also have a job with a release time and absolute deadline within an interval of length $t$. This set of tasks is again given by aff($t$, $j$), hence, we can define the UCB-Union approach for EDF as:

$$\gamma_{t,j}^{ucb-u} = \mathrm{BRT} \bullet \left| \left( \bigcup_{\forall k \in \mathrm{aff}(t,j)} \mathrm{UCB}_k \right) \cap \mathrm{ECB}_j \right| \tag{4.6}$$

### ECB-Union

The ECB-Union approach by Altmeyer *et al.* [6], see equation (3.5) in Section 3.2, accounts for nested pre-emptions by making the pessimistic assumption that in any pre-emption by task $\tau_j$, task $\tau_j$ may itself have already been pre-empted by all of the other tasks that may pre-empt it. For EDF, this set of tasks is given by $hp(j) \cup \{j\}$. Note in general this is different to the set of tasks with relative deadlines less than or equal to that of task $\tau_j$, as tasks with the same deadline as task $\tau_j$ cannot pre-empt it. Pre-emption by task $\tau_j$ is therefore assumed to potentially evict blocks in the set $\bigcup_{h \in \mathrm{hp}(j) \cup \{j\}} \mathrm{ECB}_h$. The maximum number of blocks that may be evicted as a result of an already nested pre-emption by task $\tau_j$ is then obtained by considering the maximum number of UCBs that may need to be reloaded by *any* task that may be directly pre-empted by task $\tau_j$, as in the UCB-Only case. Hence we can define the ECB-Union approach for EDF as:

$$\gamma_{t,j}^{ecb-u} = \mathrm{BRT} \bullet \max_{\forall k \in \mathrm{aff}(t,j)} \left\{ \left| \mathrm{UCB}_k \cap \left( \bigcup_{h \in \mathrm{hp}(j) \cup \{j\}} \mathrm{ECB}_h \right) \right| \right\} \tag{4.7}$$

## 4.1.1 Effect on Task Utilisation and h(t) Calculation

We have shown how ECB-only, UCB-Only, UCB-Union, and ECB-Union CRPD analysis can be integrated into the calculation of the processor demand $h(t)$.

However, to obtain a schedulability test for EDF incorporating these CRPD analyses, we also have to adjust how we calculate task utilisation and the upper bound on the values of $t$ that must be checked. Effectively, we are increasing $C_j$ by $\gamma_{t,j}$. To account for this we introduce a modified utilisation $U_j^*$ for task $\tau_j$ that includes the CRPD:

$$U_j^* = \frac{C_j + \gamma_{t,j}}{T_j} \tag{4.8}$$

We then adjust the two upper bounds for $t$ by substituting $U_j^*$ for $U_j$ in equation (2.3) and substituting $C_j^* = C_j + \gamma_{t,j}$ for $C_j$ in equation (2.4). (Note, when calculating $\gamma_{t,j}$ to include in $C_j^*$ and $U_j^*$, we use $t = D_{\max}$, the largest relative deadline, as it gives the maximum value for $\gamma_{t,j}$). This gives our revised bounds as:

$$L_a = \max\left\{D_1,...,D_n, \frac{\sum_{j=n}^{n}(T_j - D_j)U_j^*}{1-U}\right\} \tag{4.9}$$

and

$$w^{\alpha+1} = \sum_{j=1}^{n}\left\lceil \frac{w^{\alpha}}{T_j} \right\rceil C_j^* \tag{4.10}$$

Finally, we note that $\gamma_{t,j}$ is monotonically non-decreasing in $t$ and hence using the above bounds, equation (4.1) can be used with the QPA method to obtain an efficient schedulability test for EDF scheduling accounting for CRPD. We note that this test is no longer exact as the CRPD analysis is only sufficient.

We observe that for implicit deadline tasksets, a sufficient schedulability test is simply:

$$U^* \leq 1 \tag{4.11}$$

## 4.2 Improved CRPD Analysis for EDF

In this section, we present improved CRPD analysis for EDF based on the multiset approaches to CRPD analysis for FP scheduling by Altmeyer *et al.* [7], discussed in Section 3.2.1.

In the following analysis, we use $\gamma'_{t,j}$ to represent the cost of the maximum number $E_j(t)$ of pre-emptions by jobs of task $\tau_j$ that have their release times and

absolute deadlines in an interval of length $t$. It is therefore included in the processor demand bound function, equation (2.2), as follows:

$$h(t) = \sum_{j=1}^{n} \left( \max\left\{ 0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor \right\} C_j + \gamma'_{t,j} \right) \tag{4.12}$$

## ECB-Union Multiset Approach

We now present the ECB-Union Multiset approach for EDF which is derived from the ECB-Union Multiset approach for FP scheduling by Altmeyer *et al.* [7], equations (3.14), (3.15) and (3.16) in Section 3.2.

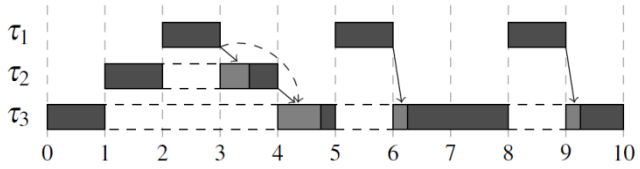



*Figure 4.4 - Illustration of possible pessimism with the ECB-Union approach. The pre-emption cost of task $\tau_1$ pre-empting task $\tau_2$ contributes three times to the total pre-emption cost of task $\tau_1$ pre-empting other tasks in an interval of length 10; despite it only really contributing at most once*

The ECB-Union approach is pessimistic in that it assumes that task $\tau_j$ can pre-empt any task $\tau_k \in \text{aff}(t, j)$ up to $E_j(t)$ times in an interval of length $t$. While this is potentially true if $D_k = t$, it can be a pessimistic assumption when $D_k < t$ and particularly when $D_k << T_k < t$. We can calculate a tighter bound on the number of times that jobs of task $\tau_k$ can be pre-empted by jobs of task $\tau_j$ in an interval of length $t$. This can be found by multiplying the maximum number of times task $\tau_j$ can pre-empt a single job of task $\tau_k$, given by $P_j(D_k)$, by the number of jobs of task $\tau_k$ that are released and have their deadlines in an interval of length $t$, given by $E_k(t)$.

First we form a multiset $M_{t,j}$ that contains the cost:

$$\left| \text{UCB}_k \cap \left( \bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \tag{4.13}$$

of task $\tau_j$ pre-empting task $\tau_k$ repeated $P_j(D_k)E_k(t)$ times, for each task $\tau_k \in \text{aff}(t, j)$, hence:

$$M_{t,j} = \bigcup_{\forall k \in \text{aff}(t,j)} \left( \bigcup_{P_j(D_k)E_k(t)} \left| \text{UCB}_k \cap \left( \bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \right) \tag{4.14}$$

As there are only $E_j(t)$ jobs of task $\tau_j$ with release times and deadlines in an interval of length $t$, the maximum CRPD is obtained by summing the $E_j(t)$ largest values in $M_{t,j}$.

$$\gamma'^{\,ecb-m}_{t,j} = \text{BRT} \bullet \sum_{l=1}^{E_j(t)} \left| M^l_{t,j} \right| \tag{4.15}$$

## UCB-Union Multiset Approach

The UCB-Union approach is also pessimistic in that it assumes that task $\tau_j$ can pre-empt any task $\tau_k \in \text{aff}(t, j)$ up to $E_j(t)$ times. The UCB-Union Multiset approach for EDF removes this source of pessimism. It is based on the UCB-Union Multiset approach for FP scheduling by Altmeyer *et al*. [7], see equation (3.11), (3.12) and (3.13) in Section 3.2

First we form a multiset $M^{ucb}_{t,j}$ containing $P_j(D_k)E_k(t)$ copies of the $\text{UCB}_k$ of each task $\tau_k \in \text{aff}(t, j)$. This multiset reflects the fact that jobs of task $\tau_j$ cannot evict the UCBs of jobs of task $\tau_k$ that have both their release times and deadlines in an interval of length $t$ more than $P_j(D_k)E_k(t)$ times. Hence:

$$M^{ucb}_{t,j} = \bigcup_{\forall k \in \text{aff}(t,j)} \left( \bigcup_{P_j(D_k)E_k(t)} \text{UCB}_k \right) \tag{4.16}$$

Next we form a multiset $M^{ecb}_{t,j}$ containing $E_j(t)$ copies of the $\text{ECB}_j$ of task $\tau_j$. This multiset reflects the fact that there are at most $E_j(t)$ jobs of task $\tau_j$ that have their release times and deadlines in an interval of length $t$, each of which can evict ECBs in the set $\text{ECB}_j$.

$$M^{ecb}_{t,j} = \bigcup_{E_j(t)} \left( \text{ECB}_j \right) \tag{4.17}$$

$\gamma'^{\,ucb-m}_{t,j}$ is then given by the size of the multiset intersection between $M^{ucb}_{t,j}$ and $M^{ecb}_{t,j}$:

$$\gamma'^{\,ucb-m}_{t,j} = \text{BRT} \bullet \left| M^{ucb}_{t,j} \cap M^{ecb}_{t,j} \right| \tag{4.18}$$

### Combined Multiset Approach

The ECB-Union Multiset and UCB-Union Multiset approaches are incomparable, we can therefore calculate $h(t)$ at each stage of the QPA algorithm using both approaches and take the minimum to form a combined approach:

$$h(t) = \min\left(h(t)^{ucb-m}, h(t)^{ecb-m}\right) \tag{4.19}$$

## 4.2.1 Effect on Task Utilisation and h(t) Calculation

The multiset approaches calculate the CRPD for all of the tasks in one go. Therefore, inflating the upper bounds on $t$ used in the schedulability test, equation (2.3) and (2.4), by substituting in $U_j^*$ and $C_j^*$ to give equation (4.9) and (4.10) as described in Section 4.1.1 is not possible. This is because the test that $U^* < 1$ may pass even though one or more tasks may have utilisations > 1, causing them to miss a deadline. Therefore, we need a new upper bound.

The method we use to determine a suitable upper bound is based on using an upper bound on the utilisation due to CRPD that is valid for all intervals of length greater than some value $L_c$. We then use this CRPD utilisation value to inflate the taskset utilisation and thus compute an upper bound $L_d$ on the maximum length of the synchronous busy period. This upper bound is valid provided that it is greater than $L_c$, otherwise the actual maximum length of the busy period may lie somewhere in the interval [$L_d$, $L_c$], hence we can use max($L_c$, $L_d$) as a bound.

We choose a value of $t = L_c = 100\ T_{max}$ which limits the overestimation of the CRPD utilisation $U^\gamma = \gamma'_t / t$ to at most 1%. We then calculate $\gamma'_t$ using equation (4.15) for ECB-Union Multiset and equation (4.18) for UCB-Union Multiset. However, in equation (4.14), (4.16) and (4.17), we substitute $E_x^{max}(t)$ for $E_x(t)$ to ensure that the computed value of $U^\gamma$ is a valid upper bound for all intervals of length $t \geq L_c$.

$$E_x^{max}(t) = \max\left(0, 1 + \left\lceil \frac{t - D_x}{T_x} \right\rceil\right) \tag{4.20}$$

We then check that $U + U^{\gamma} < 1$, if not then we deem the taskset unschedulable, otherwise we compute an upper bound on the length of the busy period via a modified version of equation (2.4):

$$w^{\alpha+1} \leq \sum_{\forall j} \left( \frac{w^{\alpha}}{T_j} + 1 \right) C_j + w^{\alpha} U^{\gamma} \tag{4.21}$$

rearranged to give:

$$w \leq \frac{1}{\left(1 - \left(U + U^{\gamma}\right)\right)} \sum_{\forall j} U_j T_j \tag{4.22}$$

Then, substituting in $T_{max}$ for each value of $T_j$ we get our upper bound:

$$L_d = \frac{U \bullet T_{max}}{\left(1 - \left(U + U^{\gamma}\right)\right)} \tag{4.23}$$

We then use $L = \max(L_c, L_d)$ as the maximum value of $t$ to check in the EDF schedulability test.

# 4.3 Comparability and Dominance

The CRPD analyses for EDF scheduling have similar comparability relationships to their counterparts presented in [7] for FP scheduling. The UCB-Union approach dominates the ECB-Only approach, and the ECB-Union approach dominates the UCB-Only approach. The JCR approach by Ju *et al.* [71], discussed in Section 3.3, is incomparable with all of the non-multiset approaches. However, if we re-write the JCR approach, equation (2.27), so that it calculates the cost of all $E_j(t)$ pre-emptions at once, then it can be seen that the UCB-Union Multiset approach dominates it.

$$\gamma'^{jcr}_{t,i} = \mathrm{BRT} \bullet \sum_{\forall j \in D_i \geq D_j \wedge i \neq j} P_j(D_i) E_j(t) \left| \mathrm{UCB}_i \cap \mathrm{ECB}_j \right| \tag{4.24}$$

Furthermore, the UCB-Union Multiset approach dominates the UCB-Union approach and the ECB-Union Multiset approach dominates the ECB-Union approach. This is because the sum of the $E_j(t)$ largest pre-emption costs will always be less than or equal to $E_j(t)$ multiplied by the largest pre-emption cost. The combined multiset approach dominates all other approaches as shown in **Error! Reference source not found.**. Furthermore, because the combined approach uses the two multiset approaches at each stage of the QPA algorithm,

the number of tasksets that it deems schedulable can is greater than a simple union of the two multiset approaches.

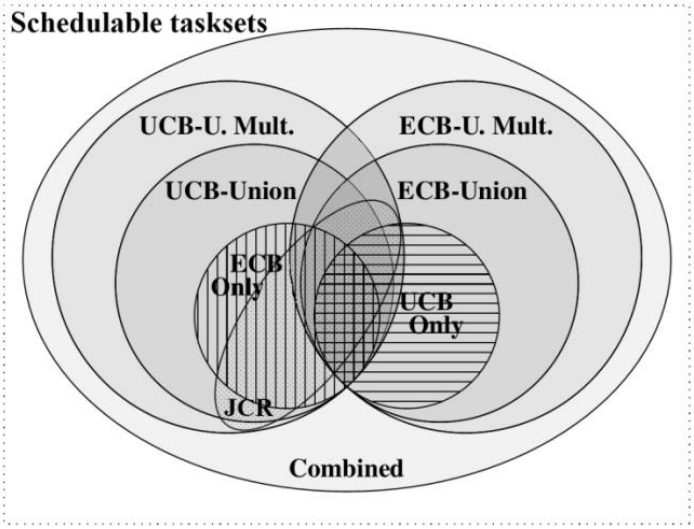



*Figure 4.5 - Venn diagram illustrating the relationship between the different approaches used to calculate CRPD. The larger the area, the more tasksets deemed schedulable by the approach*

We note that including the CRPD as if it were additional execution time of the pre-empting task, as we have done in all of the non-multiset approaches, has the potential for significant pessimism if the execution time of a task $\tau_i$ is close to its deadline such that:

$$C_j \leq D_j < C_j + \gamma_{t,j} \tag{4.25}$$

In this case task $\tau_i$ would be deemed unschedulable when it may not be. This problem is avoided by the multiset approaches.

## 4.4 Case Study

In this section we evaluate the schedulability tests for EDF including integrated CRPD analysis using the approaches introduced in this chapter: ECB-Only, UCB-Only, UCB-Union, ECB-Union, ECB-Union Multiset, UCB-Union Multiset and the combined multiset approaches, as well as the JCR approach of Ju *et al.* [71] on a case study. For comparison purposes, we also used the EDF schedulability test assuming no pre-emption costs.

The case study is the same one used in Altmeyer *et al.* [6] to evaluate CRPD analysis for systems using FP scheduling. The case study comprises a number of tasks from the Mälardalen benchmark suite[1] [68]. While these tasks do not represent a real taskset, they do represent typical code found in real-time systems. For each task, the WCET and number of ECBs and UCBs are taken from [4], details for each task can be found in Table 4.1. The system was setup to model an ARM processor clocked at 100MHz with a 2KB direct-mapped instruction cache. The cache was setup with a line size of 8 Bytes, giving 256 cache sets, 4 Byte instructions, and a BRT of 8μs. This configuration was chosen so as to give representative results when using the relatively small benchmarks that were available to us.

| | WCET | #UCBs | #ECBs |
|---|---|---|---|
| bs | 445 | 5 | 35 |
| minmax | 504 | 9 | 79 |
| fac | 1252 | 4 | 24 |
| fibcall | 1351 | 5 | 24 |
| insertsort | 6573 | 10 | 41 |
| loop3 | 13449 | 4 | 817 |
| select | 17088 | 15 | 151 |
| qsort-exam | 22146 | 15 | 170 |
| fir | 29160 | 9 | 105 |
| sqrt | 39962 | 14 | 477 |
| ns | 43319 | 13 | 64 |
| qurt | 214076 | 14 | 484 |
| crc | 290782 | 14 | 144 |
| matmult | 742585 | 23 | 100 |
| bsort100 | 1567222 | 35 | 62 |

*Table 4.1 - WCET and number of UCBs and ECBs for a selection of tasks from the Mälardalen benchmark suite*

The taskset was created by assigning periods and implicit deadlines such that all 15 tasks had equal utilisation. The periods were generated by multiplying the execution times by a constant $c$ such that $T_i = c\ C_i$ for all tasks. We varied $c$ from 15 upwards in steps of 0.25, which varied the utilisation from 1.0 downwards. In order to evaluate different approaches, we found the *breakdown utilisation* [78] of the tasksets. By scaling the deadlines and periods of the tasks, we simulated scaling the speed of the CPU and memory. Using this technique

[1] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

the breakdown utilisation, the point at which the taskset is deemed unschedulable, can be found.

| | Breakdown utilisation |
|---|---|
| No pre-emption cost | 1 |
| Combined Multiset | 0.659 |
| ECB-Union Multiset | 0.659 |
| UCB-Union Multiset | 0.594 |
| ECB-Union | 0.612 |
| UCB-Union | 0.583 |
| UCB-Only | 0.462 |
| ECB-Only | 0.364 |
| JCR | 0.488 |

*Table 4.2 - Breakdown utilisation for the case study taskset for the different approaches used to calculate the CRPD*

The breakdown utilisation for each approach is shown in Table 4.2. The ECB-Union Multiset, and hence the Combined Multiset, approach performed the best with a breakdown utilisation of 0.659. The JCR approach outperformed the ECB-Only and UCB-Only approaches with a breakdown utilisation of 0.488, but did worse than the other approaches that we have presented.

## 4.5 Evaluation

In addition to the case study, we evaluated the schedulability tests for EDF with integrated CRPD analysis using synthetically generated tasksets. This enabled us to investigate the behaviour of the different approaches as we varied a number of key parameters. We did so by generating a large number of tasksets with representative but varied timings and cache usage so that we could get an overall picture for how the different approaches performed. To determine the margin of error we re-ran a typical evaluation 100 times for each of the different number of tasksets used, using different random seeds for each run, and then computed the margin of error in each case. We note that the maximum margin of error is observed when approximately half of the tasksets are schedulable, as this is where there is the maximum variation. For a typical evaluation depicting the number of schedulable tasksets, the margin of error based on a 95% confidence interval is around ±0.1% for 10,000 tasksets per utilisation level and hence per data point and ±0.3% for 1,000 tasksets. For the weighted schedulability evaluations introduced in Section 4.5.2 the margin of error based on a 95% confidence interval is around ±0.1% for 1,000 tasksets per utilisation

level with 40 utilisation levels per data point, and ±0.25% for 100 tasksets per utilisation level again with 40 utilisation levels per data point.

The UUnifast algorithm [32] was used to calculate the utilisation, $U_i$ of each task so that the utilisations add up to the desired utilisation level for the taskset. Task periods $T_i$, were generated at random between 5ms and 500ms according to a log-uniform distribution. From this, $C_i$ was calculated via $C_i = U_i T_i$.

We generated two sets of tasksets, one with implicit deadlines and one with constrained deadlines. We used $D_i = \min(T_i, 2C_i + x(T_i - 2C_i))$ to generate the constrained deadlines, where $x$ is a random number between 0 and 1. In the following sections we assume implicit deadline tasksets unless stated otherwise. In general, using constrained deadlines resulted in an overall reduction in schedulable tasksets compared to implicit deadline tasksets.

The UCB percentage for each task was based on a random number between 0 and a maximum UCB percentage specified for the experiment. UCBs were placed in a continuous group at the start of the tasks' ECBs.

### 4.5.1 Baseline Evaluation

We investigated the effects of the following parameters:

- Cache utilisation (default of 10)
- Maximum UCB percentage (default of 30%)
- Number of tasks (default of 10)
- Number of cache sets (default of 256)
- Block Reload Time (BRT) (default of 8µs)

First we evaluated how the integrated CRPD and EDF schedulability analysis performed under the default configuration for implicit deadline tasksets. We generated 10,000 tasksets and then varied the utilisation, excluding any pre-emption cost, from 0.025 to 1 in steps of 0.025 and recorded how many tasksets were deemed schedulable by the EDF schedulability test. The results for implicit deadline tasksets are shown in Figure 4.6 and in Table 4.3 in the form of weighted schedulability measures, see the next sub-section, Section 4.5.2 for a definition of weighted schedulability.

The results follow a similar pattern to the equivalent CRPD analyses for FP scheduling, see Figure 9 in [7]. Furthermore, the results confirm the dominance relationships between approaches with the Combined Multiset approach performing the best. Additionally, with the exception of ECB-Only, all of the

approaches presented outperformed JCR with the Combined Multiset approach achieving a weighted schedulability measure of 0.528 compared to 0.333 for JCR.

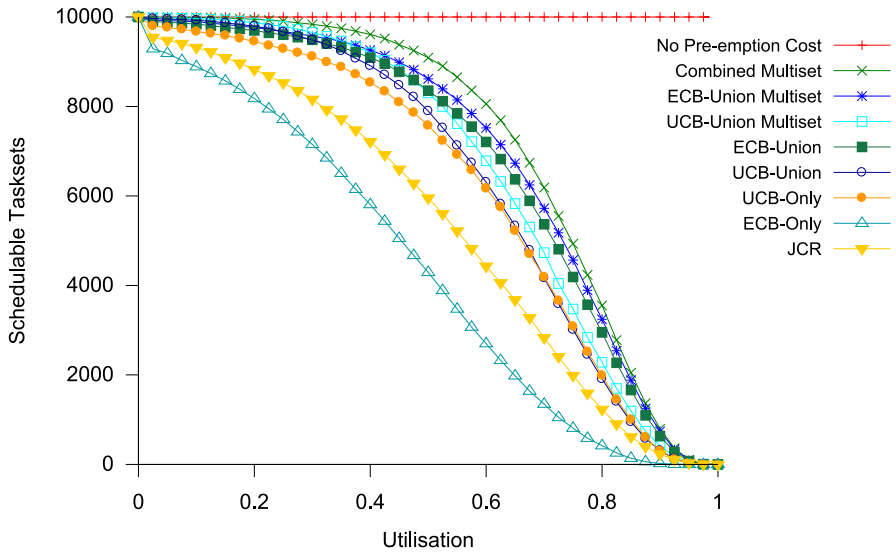



*Figure 4.6 - Schedulable tasksets vs Utilisation for the baseline parameters under implicit deadlines*

We then repeated the first evaluation with constrained deadlines. The results showed an overall reduction in the number of schedulable tasksets due to the tighter deadlines. However, the JCR approach performs better than with implicit deadlines, outperforming ECB-Only and UCB-Only. This is because the number of times task $\tau_j$ pre-empts task $\tau_k$, given by $P_j(D_k)$, is reduced. (As $D_k$ is now smaller than $T_k$, and smaller in relation to $T_j$, there is a smaller window in which task $\tau_j$ can pre-empt task $\tau_k$). The results are shown in Figure 4.7.

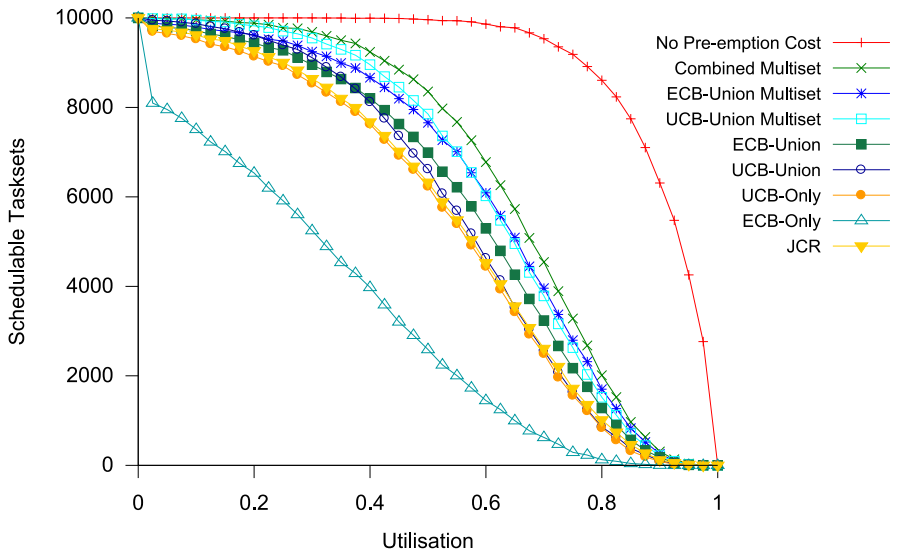



*Figure 4.7 - Schedulable tasksets vs Utilisation for the baseline parameters under constrained deadlines*

### 4.5.2 Weighted Schedulability

Evaluating all combinations of different parameters would take a significant amount of time. Therefore, the majority of our evaluation focused on varying one parameter at a time. To present the results, weighted schedulability measures [21] are used. This allows a graph to be drawn which shows the weighted schedulability, $W_l(p)$, for each method used to obtain a layout $l$ as a function of parameter $p$. For each value of $p$, this measure combines the data for all of the generated tasksets τ for all of a set of equally spaced utilisation levels, where the utilisation is without including CRPD. The schedulability test returns a binary result of 1 or 0 for each layout at each utilisation level. If this result is given by $S_l(\tau,p)$, and $u(\tau)$ is the utilisation of taskset τ, then:

$$W_l(p) = \frac{\left(\sum_{\forall \tau} u(\tau) \bullet S_l(\tau, p)\right)}{\sum_{\forall \tau} u(\tau)} \tag{4.26}$$

The benefit of using a weighted schedulability measure is that it reduces a 3-dimensional plot to 2 dimensions. Individual results are weighted by taskset utilisation to reflect the higher value placed on a being able to schedule higher utilisation tasksets.

Table 4.3 gives the weighted schedulability measures for the baseline experiment under implicit deadlines shown in Figure 4.6.

| | Weighted schedulability |
|---|---|
| No pre-emption cost | 1 |
| Combined Multiset | 0.528 |
| ECB-Union Multiset | 0.501 |
| UCB-Union Multiset | 0.455 |
| ECB-Union | 0.481 |
| UCB-Union | 0.427 |
| UCB-Only | 0.416 |
| ECB-Only | 0.236 |
| JCR | 0.333 |

*Table 4.3 - Weighted schedulability measures for the baseline experiments show in Figure 4.6*

### 4.5.3 Implicit Deadline Tasksets

In this section, we present the results for a number of weighted schedulability evaluations with implicit deadline tasksets. In each evaluation we varied one parameter and fixed all other parameters at the default values, described in Section 4.5.1, unless otherwise stated. For all the weighted schedulability evaluations, we used 1,000 generated tasksets.

#### Cache Utilisation

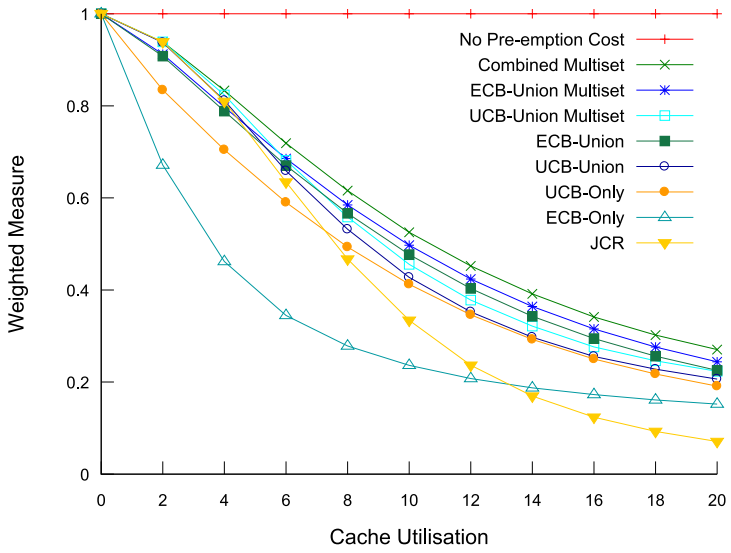



*Figure 4.8 - Weighted measure for varying cache utilisation from 0 to 20 in steps of 2 for implicit deadline tasksets*

As the cache utilisation increases, see Figure 4.8, all approaches that consider CRPD show a decrease in schedulability. In particular, the ECB-Only approach shows a very rapid decrease because the cache utilisation directly correlates

with the number of ECBs which is all that the approach considers. Additionally, the JCR approach starts to drop off at around a cache utilisation of 8, and by a cache utilisation of 14, it performs the worst. This is due to the pessimistic handling of nested pre-emptions leading to it calculating that the same UCBs are evicted multiple times as tasks share an increasing number of cache blocks.

### Maximum UCB Percentage

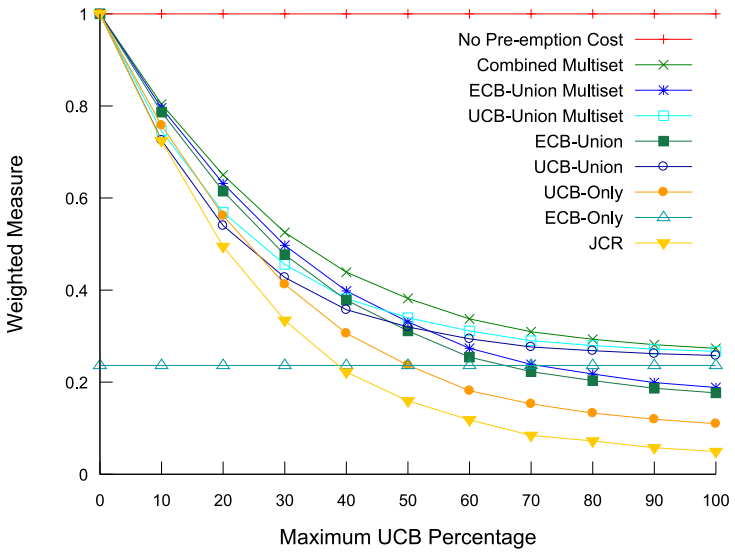



*Figure 4.9 - Weighted measure for varying the maximum UCB percentage from 0 to 100% in steps of 10% for implicit deadline tasksets*

As the maximum UCB percentage increases, see Figure 4.9, all approaches except ECB-Only show a decrease in schedulability. The ECB-Only approach shows no change because it does not consider any tasks' UCBs. The UCB-Only approach is particularly vulnerable to high numbers of UCBs. Additionally, the JCR approach also shows a large decrease in the number of schedulable tasksets. This is because it deals with nested pre-emptions by considering the pre-empting and intermediate tasks individually. As the number of UCBs increases, the chances of the analysis assuming that the UCBs get evicted more than once increases. UCB-Union, UCB-Union Multiset and Combined Multiset all tend to similar performance to ECB-Only as the number of UCBs is increased as they dominate ECB-Only. All other approaches are incomparable and perform worse than ECB-Only under very high numbers of UCBs.

### Number of Tasks

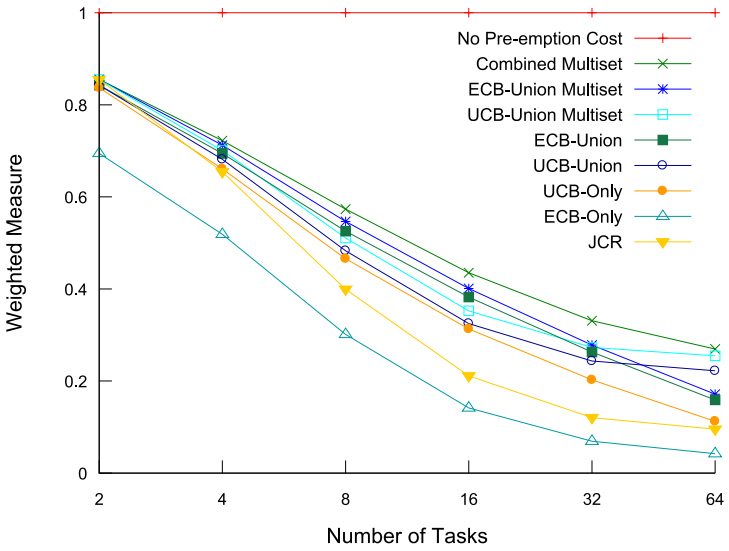



*Figure 4.10 - Weighted measure for varying the number of tasks from $2^1 = 2$ to $2^6 = 64$ for implicit deadline tasksets*

As the number of tasks increases, see Figure 4.10, all approaches that consider pre-emption cost show a decrease in schedulability due to the increased number of pre-emptions. We note that as the number of tasks becomes very high, some of the approaches level off. This is due to the fact that the other parameters, specifically cache utilisation and maximum UCB percentage are fixed. As the number of tasks increases, the size of the tasks and therefore the number of UCBs decreases, reducing the cost of a pre-emption, especially for the approaches that rely heavily on the number of UCBs. This could be avoided by fixing the task size by increasing the cache utilisation, but then this would also affect the results as shown previously in Figure 4.8.

### Cache Size

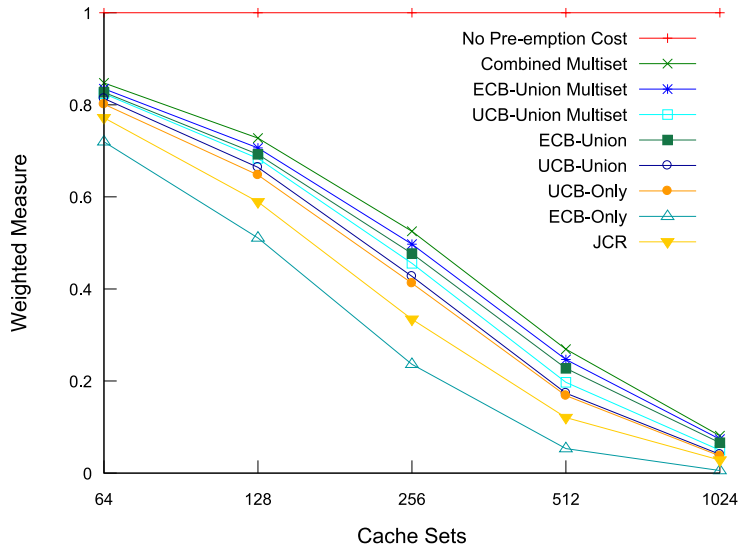



*Figure 4.11 - Weighted measure for varying the number of cache sets from $2^6 = 64$ to $2^{10} = 1024$ for implicit deadline tasksets*

The cache size also has an effect on the schedulability of tasksets, see Figure 4.11. As the number of cache sets increases, all approaches show a decrease in schedulability because the potential impact of a pre-emption increases.

### Block Reload Time (BRT)

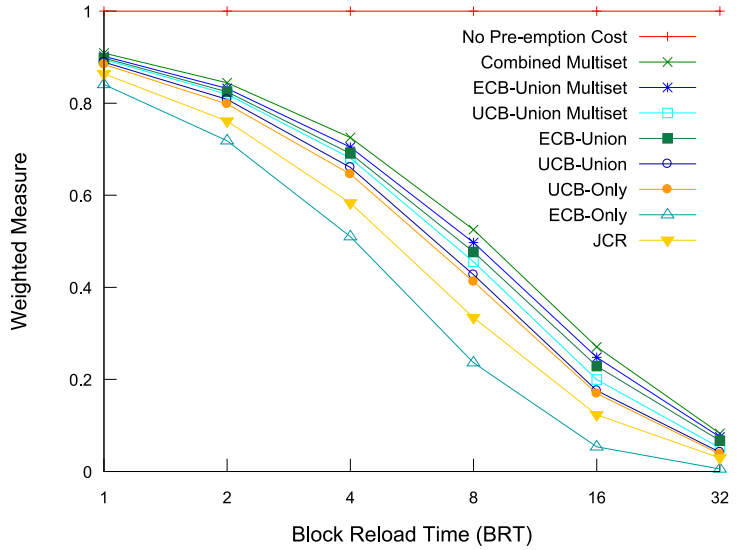



*Figure 4.12 - Weighted measure for varying the block reload time from $2^0 = 1\mu s$ to $2^5 = 32\mu s$ for implicit deadline tasksets*

Varying the BRT also has a similar effect of increasing the cost of a pre-emption which in turn results in fewer tasksets being deemed schedulable, as seen in Figure 4.12.

### 4.5.4 Constrained Deadline Tasksets

We now briefly present the results for the weighted schedulability evaluation under constrained deadlines. In general, using constrained deadlines resulted in an overall reduction in the number of schedulable tasksets compared to implicit deadline tasksets. However, we note that the JCR approach shows an improvement compared to the implicit deadline case for the reason noted in Section 4.5.1, because the number of times task $\tau_j$ pre-empts task $\tau_k$, $P_j(D_k)$, is reduced. (As $D_k$ is now smaller than $T_k$, and smaller in relation to $T_j$, there is a smaller window in which task $\tau_j$ can pre-empt task $\tau_k$). Nevertheless, while it does better than the ECB-Only and UCB-Only approach, the JCR approach is still outperformed by the other approaches presented in this chapter in almost all cases. Furthermore, the Combined Multiset approach presented always outperforms the JCR approach.

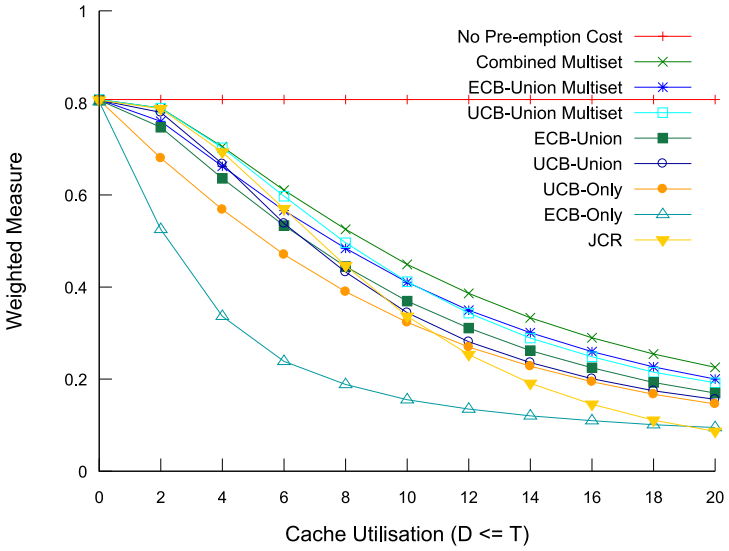



*Figure 4.13 - Weighted measure for varying cache utilisation from 0 to 20 in steps of 2 for constrained deadline tasksets*

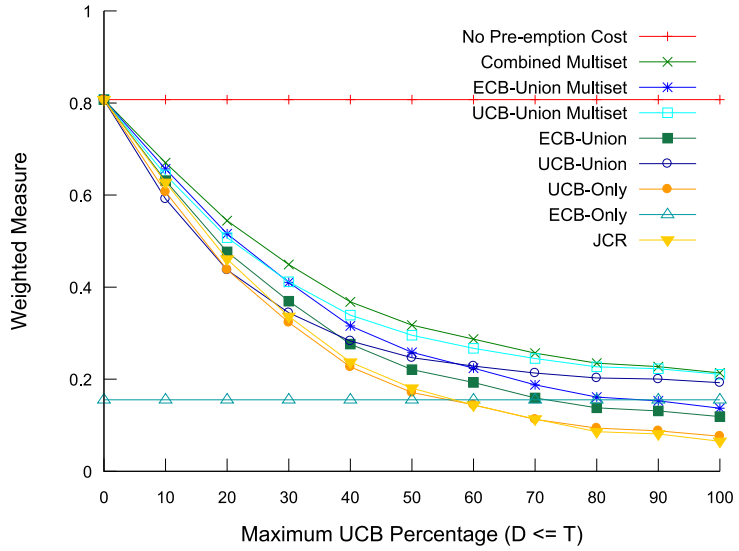



*Figure 4.14 - Weighted measure for varying the maximum UCB percentage from 0 to 100% in steps of*

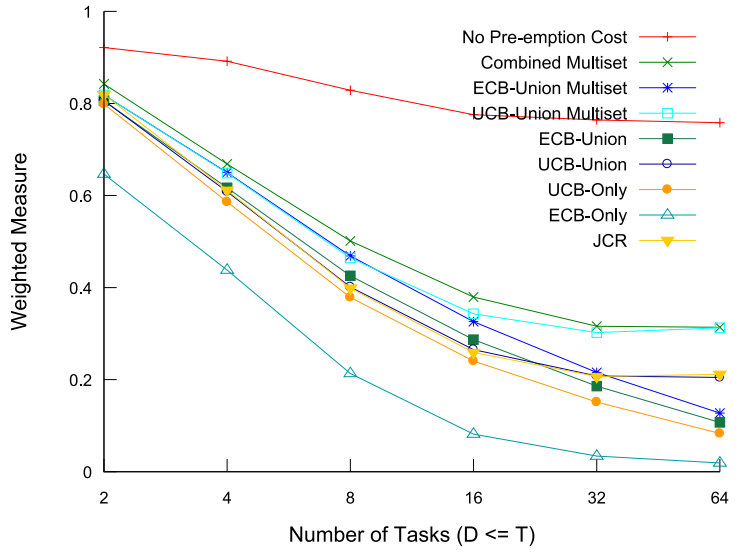



*Figure 4.15 - Weighted measure for varying the number of tasks from $2^1 = 2$ to $2^6 = 64$ for constrained deadline tasksets*

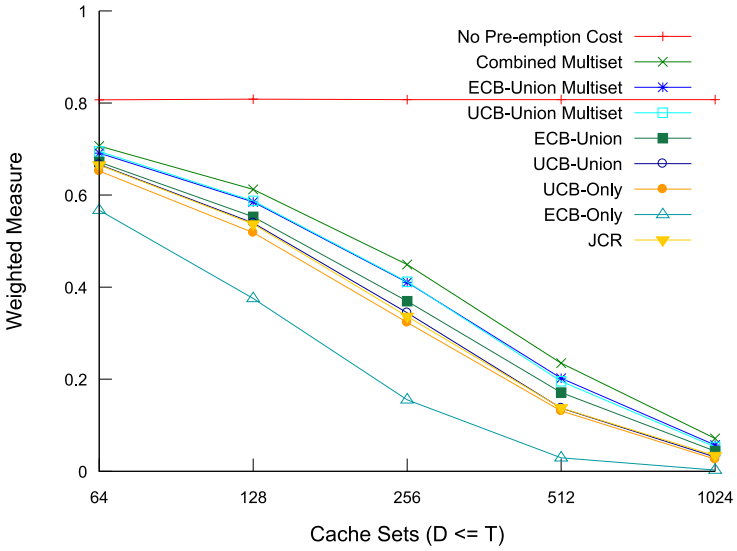



*Figure 4.16 - Weighted measure for varying the number of cache sets from $2^6$ = 64 to $2^{10}$ = 1024 for constrained deadline tasksets*

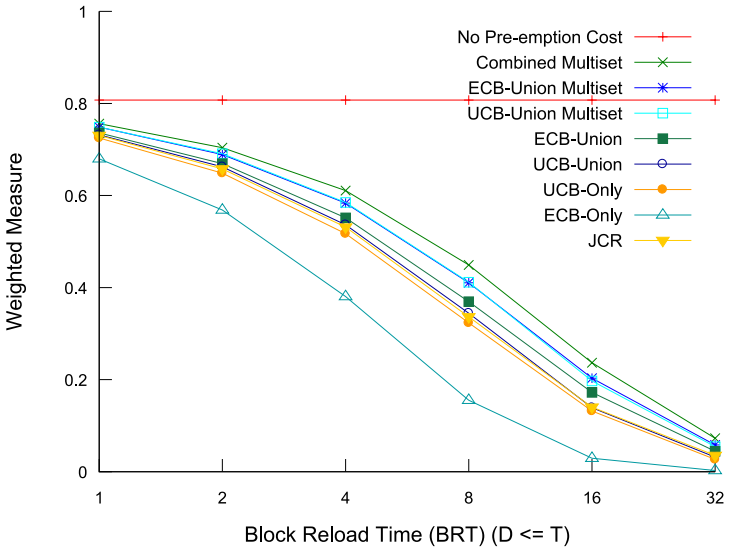



*Figure 4.17 - Weighted measure for varying the block reload time from 20 = 1µs to 25 = 32µs for constrained deadline tasksets*

## 4.6 Summary

In this chapter we have presented new CRPD aware analysis for the EDF scheduling algorithm based on similar work for FP scheduling. We compared our new approaches against an existing approach for EDF by Ju *et al.* [71], referred to as JCR, and showed that our Combined Multiset approach dominates the JCR approach. This was confirmed in both a case study and a series of evaluations based on synthetically generated tasksets. We examined

the effects of different cache and taskset parameters on the different approaches, highlighting the strengths and weaknesses of the different approaches. We found that the JCR approach was especially vulnerable to high numbers of tasks, high cache utilisation and high UCB percentages. In all of our evaluations, our new Combined Multiset approach was able to schedule the highest number of tasksets out of the approaches that consider CRPD.

# CHAPTER 5. TASK LAYOUT OPTIMISATION

If a pre-empting task does not share any cache sets with a task that it is pre-empting, then the pre-emption will not result in any CRPD. In most cases it would not be possible to avoid all conflicts, but it is feasible to try to minimise them. In this chapter, we present a technique for optimising task layout in memory so as to increase system schedulability via reduced CRPD. By evaluating layouts using schedulability analysis which accounts for CRPD, we are able to discover layouts that help to maximise the schedulability of a taskset.

## 5.1 Introduction

Tasks are stored in memory and then loaded into cache when needed. As the size of the cache is usually smaller than the size of the memory and in some cases the size of the tasks, blocks from one task will often be mapped to the same location as blocks from other tasks. During a pre-emption, CRPD is introduced when the ECBs from the pre-empting task evict UCBs belonging to the pre-empted task(s). It is therefore desirable to organise tasks in memory, so that when they are loaded into cache, the UCBs of lower priority tasks do not share the same locations in cache as the ECBs of higher priority tasks that can pre-empt them. This is particularly important with respect to the ECBs of high priority tasks with relatively short periods that may pre-empt numerous times. In most cases it is not possible to completely avoid such mappings to the same location in cache. Nevertheless, layouts can be found that increase the schedulability of the taskset.

### Example Layouts

Figure 5.1 shows how five tasks scheduled under FP ordered by priority could be laid out in cache. Task $\tau_1$ has the highest priority, so its UCBs can never be

evicted as it cannot be pre-empted. Task $\tau_2$ and $\tau_3$'s UCBs are safe from eviction as they are not mapped to the same location in cache as higher priority tasks' ECBs. However, task $\tau_4$'s UCBs could be evicted by task $\tau_1$, and $\tau_5$'s UCBs could be evicted by task $\tau_1$, $\tau_2$ or $\tau_4$.

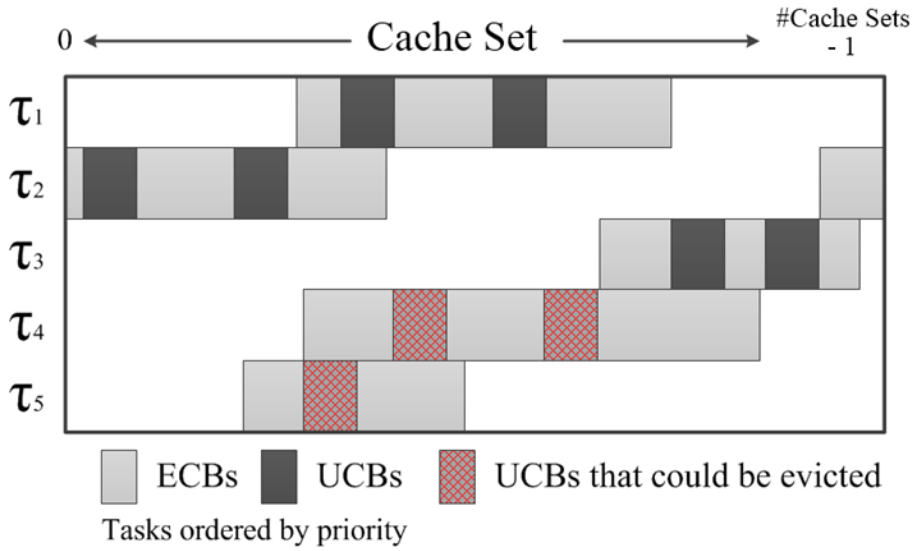



*Figure 5.1 - Example layout showing how the position of tasks in cache affects whether their UCBs could be evicted during pre-emption.*

An improved layout is shown in Figure 5.2. Although the UCBs of task $\tau_5$ could still be evicted, they can now only be evicted by the ECBs of task $\tau_3$, rather than tasks $\tau_1$ $\tau_2$ and $\tau_4$.

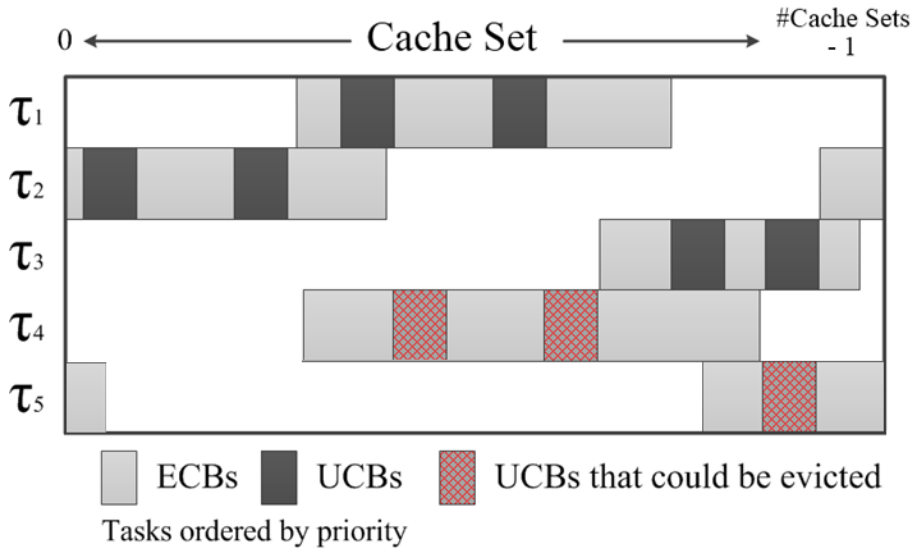



*Figure 5.2 - Improved version of the layout shown in Figure 5.1. While the UCBs of task $\tau_5$ could still be evicted, they cannot only be evicted by the ECBs of task $\tau_3$, rather than tasks $\tau_1$ $\tau_2$ and $\tau_4$.*

The aim of this approach is to find a layout for a given taskset that results in the taskset being schedulable. Good layouts reduce the CRPDs experienced by those tasks that are close to missing their deadlines. The code itself is not modified, only the start positions of each task in memory. This can be implemented in practice by controlling the linker or simply the order in which task objects files are passed to it.

In order to evaluate different layouts for a taskset, a schedulability test that can account for CRPD can be used. As a taskset has a fixed utilisation defined by the execution times and periods of the tasks, a schedulability test can only check if the taskset is, or is not schedulable with a given layout. This boolean result is not enough information to distinguish between layouts that result in the taskset being only just schedulable, and better layouts that are robust to changes in the processor speed or task execution times. We therefore use the breakdown utilisation of the taskset as an indicator of the quality of the layout. Scaling the deadlines and periods of the tasks simulates slowing down or speeding up the speed of the CPU and memory. Using this technique the breakdown utilisation, the point at which the taskset becomes unschedulable, can be found for each layout. This gives a numerical value that can be used to compare layouts for each taskset.

## 5.2 Optimising Task Layout

It would not be feasible to evaluate every possible layout for a taskset. We therefore developed an approach that uses a *simulated annealing* (SA) to discover improved task layouts. The SA works by starting with an initial layout, and then on each iteration making a random change and then evaluating the effect of that change. In this case we make a random change to the layout of tasks in memory, and then evaluate the effect that that change has had on the breakdown utilisation of the taskset.

We started with an initial layout where tasks were ordered sequentially based on their priority without any gaps between them. To apply this initial layout under EDF scheduling, tasks can be ordered based on their unique task index.

### Layout changes

The possible changes to the task layout are *swap near*, *swap far*, and *random gap*.

### Swap near

Swap near swaps the position of two neighbouring tasks by picking a random task and swapping it with the task that is in the next location in memory to it. If the selected task is the last in memory, it is swapped with the first task.

### Swap far

Swap far swaps the position of two randomly chosen tasks. These tasks are usually not adjacent in memory, but they can be. These two tasks are swapped and if necessary the start positions of the tasks in between them are adjusted. This effectively shifts the start positions in memory of all of the tasks in-between the two chosen tasks by the difference in the size of the two tasks.

### Random gap

Random gap adds a gap between two adjacent tasks in memory by up to ±*half cache size* based on a random value. Tasks cannot overlap in memory, but if a gap already exists it can be reduced. If the gap between tasks becomes greater than the size of the cache, it is reduced so as not to waste space. This is because for a direct mapped cache the position in cache is calculated by taking the position in memory modulo the size of the cache. If a task with a gap after it is swapped with another task its gap is maintained so. the gap is moved with the task.

## Layout Evaluation

Changes are made to the layout of tasks in memory, and then mapped to their cache layout for evaluation. The breakdown utilisation of the taskset is then evaluated for each layout generated by the SA. A binary search can be used to find the breakdown utilisation. The binary search starts with a maximum utilisation of 1 and a minimum utilisation of 0. The search then terminates once the minimum value is within 0.01 of the maximum. After each change to the utilisation the schedulability analysis is re-run, and the process repeats until the breakdown utilisation is found for the layout. The optimum layout is the layout which has the highest breakdown utilisation.

An initial temperature, *temp*, of 100 is defined for the SA and after every iteration the temperature is reduced by multiplying it by a cooling rate of 0.98 until it reaches the target temperature of 0.05. While the temperature is high the algorithm is more open to negative changes, which are required to escape local minima. The start and end values were chosen to balance accepting negative changes, and the cooling rate was chosen to give enough generations for the

algorithm to find a near optimal solution, without having an excessive number of iterations. The total number of iterations based on the initial and end temperature and cooling rate is 377 per taskset. The exception to this rule is that if the SA finds a layout with a breakdown utilisation of 1, it will terminate early. This is because the utilisation cannot be higher than 1 for a single core processor, and so the SA algorithm can stop having found an optimal solution.

If the change in breakdown utilisation, ΔBU, from the last iteration is positive then the layout is always accepted. If the change is negative then the layout may still be accepted based on how negative a change it is and the temperature. The probability of accepting a negative change, $P_{accept\ neg\ \Delta}$ is defined as:

$$P_{accept\ neg\ \Delta} = e^{\frac{\Delta BU}{temp}} \tag{5.1}$$

The complete processes is summarised in a flow chart shown in Figure 5.3.

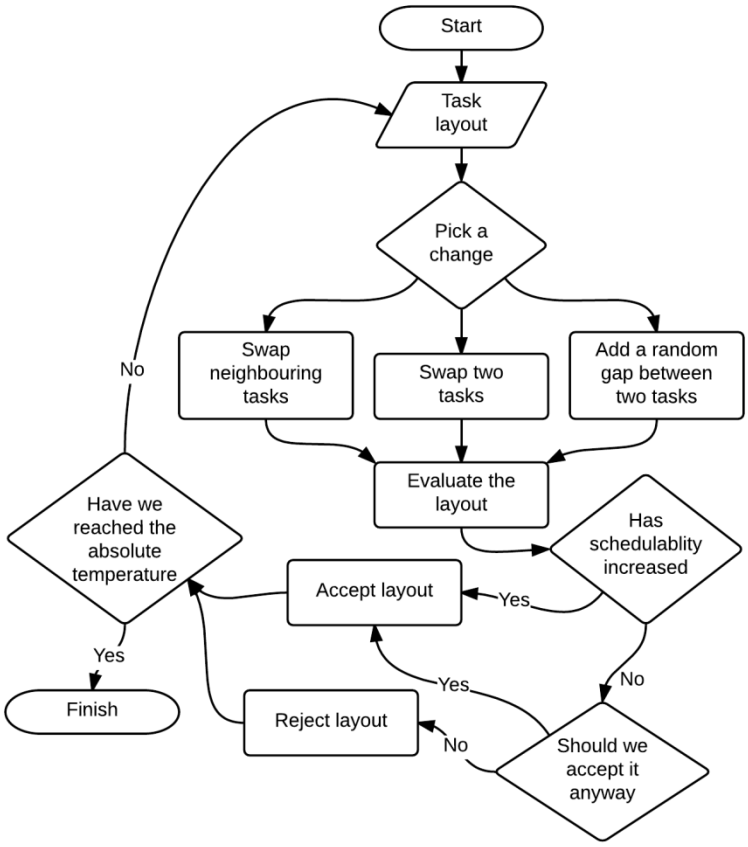



*Figure 5.3 - Task layout optimisation process flow chart*

### 5.2.1 Memory Limitations

To limit increases in the amount of memory required due to gaps introduced between tasks, the algorithm can also factor in how much free space may be introduced when finding the memory layout. If this is above the amount specified, then the new layout will be rejected and will not be evaluated by the schedulability test. For example, memory overheads would be 0% for no additional free space, 10% for a small amount of free space, or 100% for as much free space as used space.

## 5.3 Case Study

In this section we describe the results of a case study used to evaluate the task layouts produced by the SA algorithm. The case study is the same one used in Section 4.4 to evaluate CRPD analysis for EDF scheduling. For each task the derived WCET, ECBs and UCBs are shown again in Table 5.1. The system was setup to model the same ARM processor. It was clocked at 100MHz with a 2KB direct-mapped instruction cache with a line size of 8 Bytes giving 256 cache sets, 4 Byte instructions, and a block reload time of 8µs.

| | WCET | #UCBs | #ECBs |
|---|---|---|---|
| bs | 445 | 5 | 35 |
| minmax | 504 | 9 | 79 |
| fac | 1252 | 4 | 24 |
| fibcall | 1351 | 5 | 24 |
| insertsort | 6573 | 10 | 41 |
| loop3 | 13449 | 4 | 817 |
| select | 17088 | 15 | 151 |
| qsort-exam | 22146 | 15 | 170 |
| fir | 29160 | 9 | 105 |
| sqrt | 39962 | 14 | 477 |
| ns | 43319 | 13 | 64 |
| qurt | 214076 | 14 | 484 |
| crc | 290782 | 14 | 144 |
| matmult | 742585 | 23 | 100 |
| bsort100 | 1567222 | 35 | 62 |

*Table 5.1 - WCET and number of UCBs and ECBs for a selection of tasks from the Mälardalen benchmark suite*

We scheduled the taskset using FP scheduling and performed schedulability analysis using the Combined Multiset approach by Altmeyer *et al.* [7], described

in Section 3.2, when evaluating the task layouts. However, we note that the approach is not dependent on the scheduling algorithm provided it is capable of accounting for CRPD. In Chapter 6 we compare FP and EDF and apply this task layout technique to both.

The taskset was created by assigning periods and implicit deadlines such that all 15 tasks had equal utilisation. The periods were generated by multiplying the execution times by a constant $c$ such that $T_i = c\ C_i$ for all tasks. For example, $c$ = 15 gave a utilisation of 1.0 and $c$ = 30 gave a utilisation of 0.5. Tasks were assigned priorities in deadline monotonic priority order.

We compared the following layouts:

- *SA* - The layout with the highest breakdown utilisation as found by the SA algorithm with an allowed memory overhead of 0%, so that adding a random gap between tasks was not allowed.
- *Sequential ordered by priority (SeqPO)* - Lays out tasks one after another with no gaps in-between them. Tasks are in priority order with the highest priority task first. This is the starting layout for the SA.
- *Random* - 1000 different random tasks orderings in memory are evaluated and the average breakdown utilisation for them is used.
- *CS[i]=0* - Aligns the start of every task to the first cache set. This is almost always the worst possible layout, especially when UCBs are grouped at the start of the task. Note the CS[i]=0 layout has no restriction on how much memory it can use.

For comparison the analysis is also performed on the taskset with the pre-emption cost ignored.

The results showing the breakdown utilisation for each layout are given in Table 5.2. In this case, the layout obtained via SA provides a significant increase in the breakdown utilisation over that obtained by SeqPO of 0.876 versus 0.698. The results obtained from 1000 random layouts give some interesting results. First, the best layout found via a random approach did result in a slightly higher breakdown utilisation than the layout found by the SA in this case; although at the expense of evaluating more layouts than the SA. Secondly, SeqPO resulted in a breakdown utilisation that was similar to the average of the 1000 random layouts. Finally, aligning all tasks at the start of the cache resulted in a breakdown utilisation that performed similarly to the worst random layout. The slight variation is due to the fact that the UCBs of tasks are not all located at the same position within the tasks.

| | Breakdown utilisation |
|---|---|
| No pre-emption cost | 0.984 |
| SA | 0.876 |
| SeqPO | 0.698 |
| Random (min, average, max) | 0.526,0.685, 0.882 |
| CS[i]=0 | 0.527 |

*Table 5.2 - Breakdown utilisation for the taskset in Table 5.1*

### 5.3.1 Discussion

Figure 5.4 shows a representation of the initial layout of the taskset in Table 5.1, where tasks are laid out sequentially based on their priority. Figure 5.5 shows the layout chosen by the SA for this particular taskset. Although the layout generated by the SA algorithm has a larger number of UCBs in conflict compared to the SeqPO layout, it improves taskset schedulability. This is because of how the UCBs are organised. In the layout generated by the SA algorithm the likelyhood of the UCBs of lower priority tasks being evicted is reduced in comparison to their positions in the SeqPO layout. This is due to the fact that high priority tasks, especially tasks $\tau_1$ to $\tau_5$, have much shorter periods than the lowest priority tasks and can therefore pre-empt them many times.

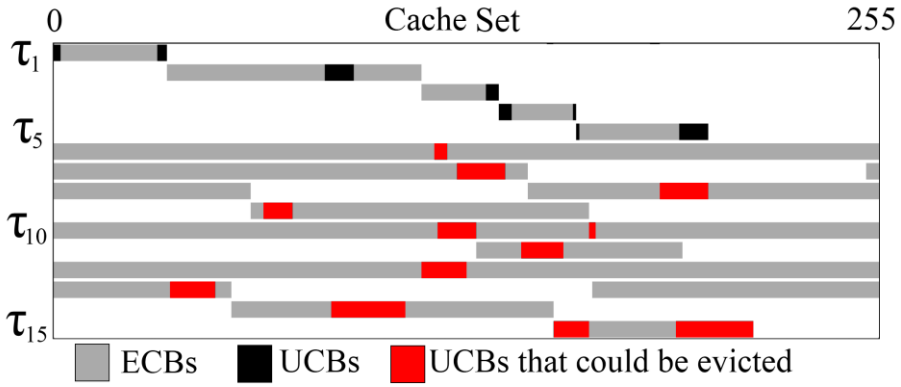



*Figure 5.4 - Initial (SeqPO) layout for the taskset in Table 5.1*

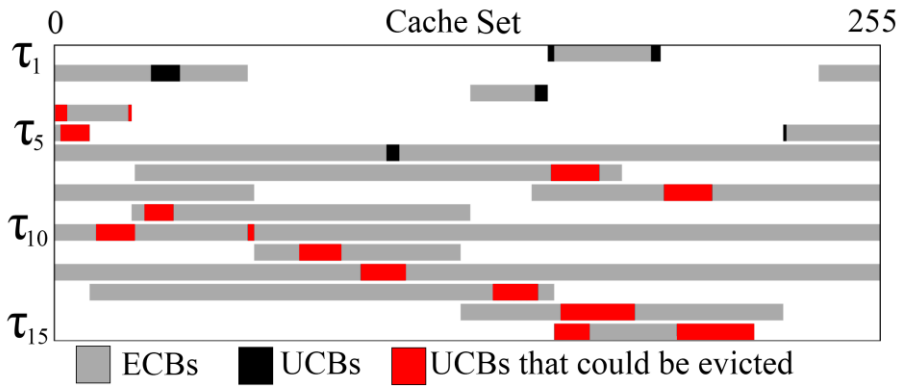



*Figure 5.5 - Optimised layout chosen by the SA for the taskset in Table 5.1*

Figure 5.6 shows a graph of the total CRPD for each task for the layout chosen by the SA algorithm and for the SeqPO layout at the breakdown utilisation for SeqPO. Note that because the Combined Multiset approach used in the evaluation is a combination of two approaches, UCB-Union Multiset and ECB-Union Multiset [7], the CRPD shown is for each of the approaches. It can be seen that the SA algorithm significantly minimises the CRPD for the low priority tasks, $\tau_{13}$, $\tau_{14}$, and $\tau_{15}$, which are close to missing their deadlines at the expense of the higher priority tasks, $\tau_4$ and $\tau_5$, which have plenty of slack time.

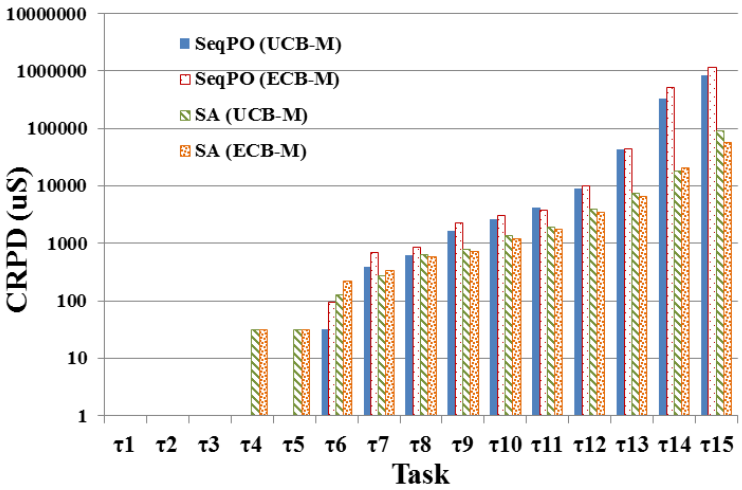



*Figure 5.6 - Graph of the total CRPD/task for the taskset in Table 5.1 under the initial SeqPO layout vs the optimisised layout chosen by the SA*

## 5.4 Evaluation

In addition to the case study, in this section we describe the results of a number of evaluations aimed at investigating the performance of the SA algorithm in

terms of the quality of the layouts it produces for synthetically generated tasksets, controlled by a random seed for repeatability.

We used the UUnifast algorithm [32] to calculate the utilisation, $U_i$, of each task so that the task utilisations added up to the desired utilisation level for the taskset. Task periods $T_i$, were generated at random between 5ms and 500ms according to a log-uniform distribution. From this, $C_i$ was calculated such that $C_i = U_i T_i$. As implicit deadlines were used, $D_i = T_i$.

UCBs were distributed through each task. Figure 5.7 shows two different distributions of UCBs.

A) Consolidates all of the UCBs into a single block at the start of the task.
B) Groups the UCBs into blocks throughout the task. Distribution A is a special case where the number of groups is 1 and the starting position is fixed to 0.

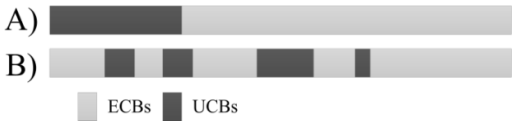



*Figure 5.7 - Two different distributions of UCBs throughout a task*

A single group of UCBs at the start of a task, represented by distribution A, is not representative of real code. Therefore the majority of the evaluations were performed and presented using distribution B.

For distribution B the UUnifast algorithm was used to generate a random distribution of UCBs throughout the tasks. This required two parameters, the number of UCBs and the number of groups of UCBs. The number of UCBs for each task was found by multiplying the UCB percentage by the number of ECBs. The UCB percentage for each task was based on a random number between 0 and a maximum UCB percentage specified for the evaluation.

The number of UCB groups used was a random number between 1 and the given maximum number of UCB groups. Because UUnifast returns floating point numbers for the number of blocks in each UCB group, the number of blocks was rounded down to the nearest whole number with the remainder carried forward and added to the next group. The final group of UCBs then had either 0 or 1 extra block added on the end. In some cases, the final number of UCB groups was less than the number given to UUnifast. This happened when

the number of UCBs in a group was less than 1.0 or the number of blocks in a gap between UCBs was less than 1.0.

UUnifast was first used to generate the size of the groups of UCBs. It was then re-run to generate the gaps between the groups of UCBs, at which point the UCBs were then laid out using a random starting position.

### 5.4.1 Baseline Evaluation

A number of evaluations were run in order to investigate the quality of the task layouts produced by the SA for different cache and task configurations. These evaluations looked at varying the following parameters:

- Distribution of UCBs
- Maximum number of UCB groups when using distribution B
- Maximum UCB percentage
- Cache utilisation
- Number of cache sets
- Number of tasks
- Allowed memory overhead

Cache utilisation describes the ratio of the total size of the tasks to the size of the cache. A cache utilisation of 1 means that the tasks fit exactly in the cache, whereas a cache utilisation of 5 means the total size of the tasks is 5 times the size of the cache.

Unless otherwise stated, the parameters were fixed to the following default values during the evaluations:

- Allowed memory overhead was fixed to 0% such that adding a random gap between tasks was not allowed
- 10 tasks per taskset
- 1000 tasksets per evaluation
- Cache size of 512 sets
- Cache utilisation of 5
- Maximum UCB percentage of 30%
- UCBs distributed using distribution B with a maximum of 5 groups

The case study used a single taskset. Therefore, 1000 random layouts were evaluated and averaged out. As the evaluations using synthetically generated tasksets used a large number of tasksets, only one random layout per taskset

was used. Any bias by using one random layout per taskset is then averaged out over the large number of tasksets.

The first evaluation investigates the quality of the task layouts produced by the SA algorithm compared to the other layouts. Figure 5.8 shows results for distribution B. This graphs shows the number of schedulable tasksets versus utilisation for no pre-emption cost, SA, SeqPO, random and CS[i]=0.

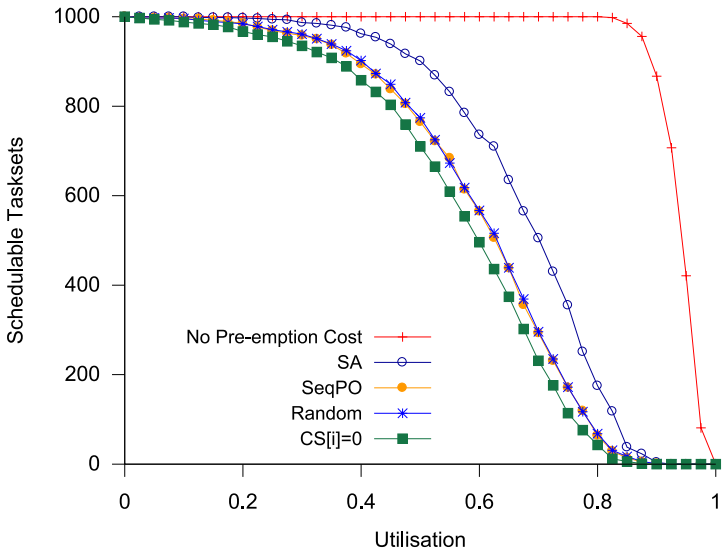



*Figure 5.8 - Schedulable tasksets vs Utilisation for UCB distribution B with a maximum of 5 groups of UCBs.*

It can be seen that aligning all tasks at a the start of the cache, CS[i]=0, results in the worst performance. SeqPO and random were very similar, and the layout generated by the SA algorithm resulted in the highest success rate when accounting for pre-emption costs.

### UCB Distribution

Table 5.3 shows the weighted schedulability measures, described in Section 3.5.2, for the baseline evaluation using distribution A and B. The table shows that distribution A results in a larger number of tasksets being schedulable at higher utilisations than distribution B for all taskset layouts; except no pre-emption cost which is not affected by the UCB distribution. This is expected as it is much harder to layout tasks with the more realistic fragmented distribution B in a way that reduces conflicts between the ECBs of high priority tasks and the UCBs of the lower priority tasks. Nevertheless, in both cases the SA algorithm was able to improve the weighted measure of 0.581 and 0.377 for

SeqPO to 0.665 and 0.465. This is a significant improvement as can be seen in Figure 5.8.

| | Distribution A | Distribution B |
|---|---|---|
| No pre-emption cost | 0.859 | 0.859 |
| SA | 0.665 | 0.465 |
| SeqPO | 0.581 | 0.377 |
| Random | 0.578 | 0.379 |
| CS[i]=0 | 0.475 | 0.347 |

*Table 5.3 - Weighted schedulability measures for the baseline evaluations*

### 5.4.2 Detailed Evaluation

Evaluating all combinations of different task parameters is not possible. Therefore, the majority of our evaluations focused on varying one parameter at a time. To present these results weighted schedulability measures [7] are used, which are described in Section 4.5.2. For these weighted schedulability evaluations, we used 100 tasksets rather than 1000 tasksets at each utilisation level.

#### Maximum UCB Groups

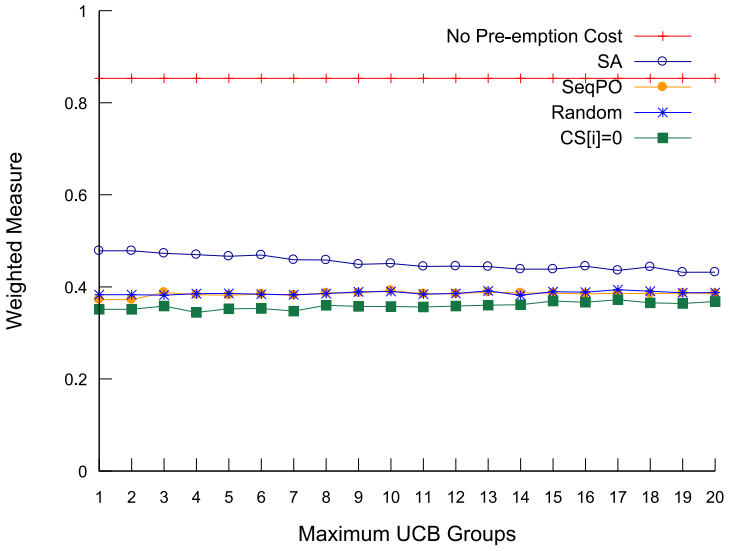



*Figure 5.9 - Weighted measure for varying the number of maximum number of UCB groups from 1 to 20*

Figure 5.9 show the impact on the schedulability of the tasksets as the maximum number of UCBs groups is varied from 1 to 20. As noted in Section 5.4, the actual number of UCB groups is chosen at random between 1 and the maximum. For small numbers of UCB groups, the weighted measure is slightly higher as the tasks are easier to layout in a way that reduces conflicts between

the ECBs of pre-empting tasks and the UCBs of pre-empted tasks. This is because the UCBs are less fragmented. As the number of groups increased, the weighted measure levels off and the SA algorithm continued to perform well in terms of the quality of the layouts it produced. The weighted measure does not decrease as the number of UCB groups becomes very large because the UCBs effectively become uniformly spread throughout the ECBs of each task. This leads to the CRPD becoming dependent only on how the ECBs are laid out.

### Maximum UCB Percentage

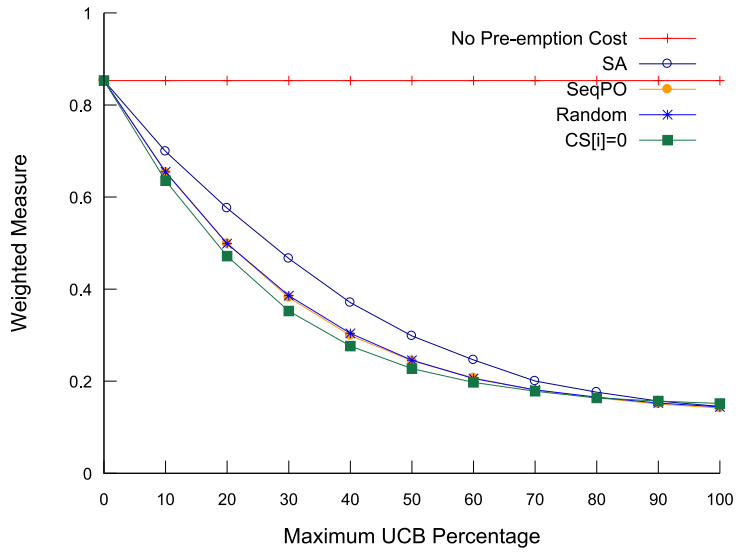



*Figure 5.10 - Weighted measure for varying the maximum UCB percentage from 0% to 100%*

The results for varying the maximum UCB percentage from 0% to 100% are shown in Figure 5.10. As expected, when the maximum UCB percentage is 0% the layout has no effect on the schedulability of the taskset and all of the weighted measures are equal to the no pre-emption cost measure. This is because there are no UCBs to be evicted, resulting in zero CRPD. As the maximum UCB percentage increases, the SA algorithm is able to find improved layouts with respect to the SeqPO layout which increases the schedulability of the taskset. When the maximum UCB percentage gets very high (>90%), there are so many UCBs that there is little that can be done to the layout to improve the schedulability of the taskset. This results in similar performance for all layouts.

### Cache Utilisation

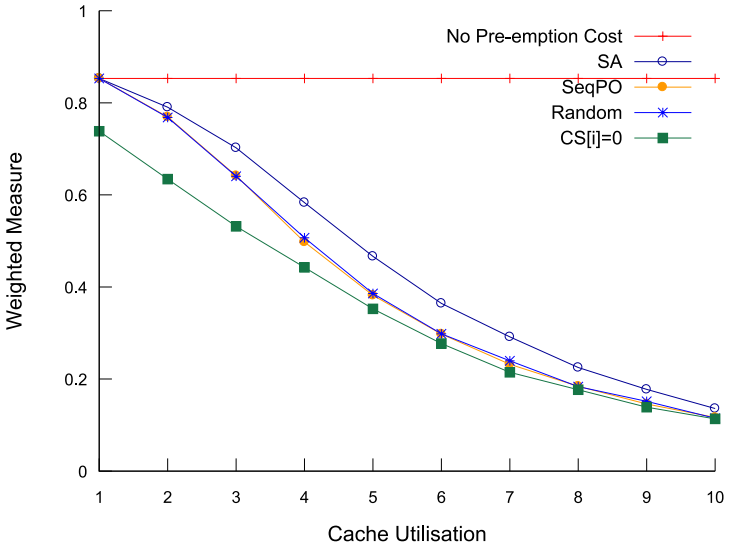



*Figure 5.11 - Weighted measure for varying the cache utilisation from 1 to 10*

The cache utilisation can also have a significant impact on the schedulability of tasksets. The results for varying the cache utilisation from 1 to 10 are shown in Figure 5.11. A cache utilisation of 1 represents all the tasks fitting into the cache therefore any layout which does not include gaps between tasks is an optimal layout. Such a layout therefore gives a weighted measure that is the equal to the no pre-emption cost case. This is why CS[i]=0 does not have the same weighted measure with a cache utilisation of 1, as does not maximise the available cache size. As the cache utilisation increases, the weighted measure decreases for all layouts with the layouts generated by the SA algorithm giving improved results up to a cache utilisation of 10.

### Cache Sets

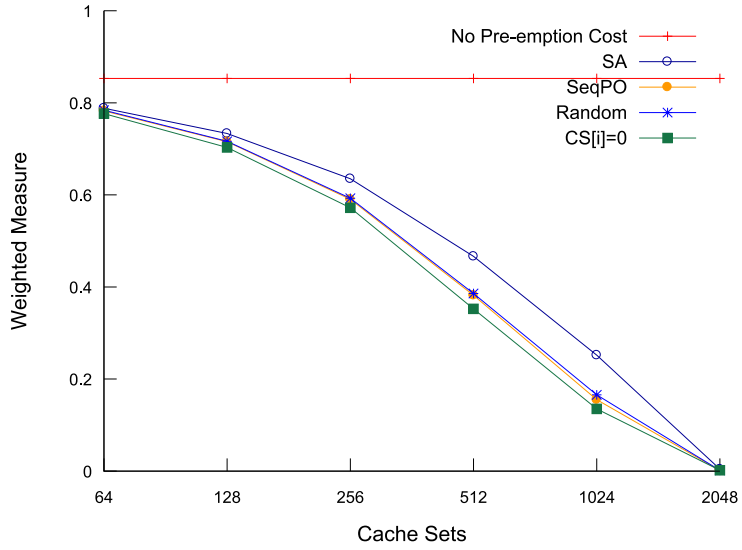



*Figure 5.12 - Weighted measure for varying the number of cache sets from 64 to 2048*

The results for varying the number of cache sets from 64 to 2048 are shown in Figure 5.12. For a given cache utilisation and BRT, as the number of cache sets increases, the impact of a pre-emption can increase as the number of evicted blocks increases. This is what causes the weighted measures to decrease until 2048 cache sets, when almost all the tasksets become unschedulable at most utilisations when accounting for pre-emption costs. When varying the number of cache sets the layouts generated by the SA algorithm outperformed the other task layouts, until 2048 cache sets where the pre-emption cost became too great.

### Number of Tasks

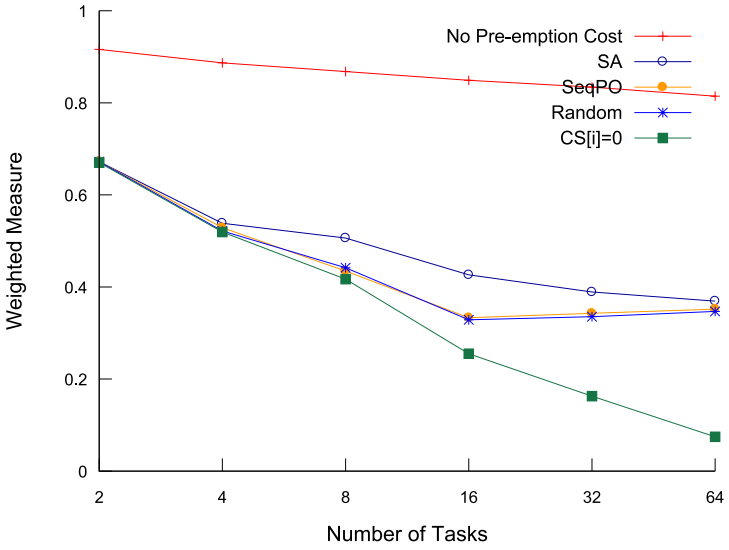



*Figure 5.13 - Weighted measure for varying the number of tasks from 2-64 in powers of 2*

As the number of tasks increases, the number of schedulable tasksets decreases as expected because of the increased number of pre-emptions. Figure 5.13 shows that after about 20 tasks the schedulability of the tasksets levels out for all the layouts except for CS[i]=0. CS[i]=0 performs increasingly worse as the number of tasksets are increased, as it aligns all of the tasks on top of each other in the cache. The result that the weighted measure levels off for SA, SeqPO and random layouts is counter-intuitive. This is most likely due to the fact that the cache utilisation was fixed. Therefore, as the number of tasks increased, the size of the tasks decreased to a point where they were relatively easy to layout.

### Discussion

Finding an improved layout for a taskset with 10 tasks took around 10 seconds on average, and 60 seconds on average for 24 tasks, using a single thread on a processor running at 2.8GHz. We felt this was an acceptable amount of time so did not pursue a more complex algorithm which could reduce the number of layouts that must be evaluated.

We also investigated the distribution of CRPD per task for our default values under different layouts. We found that it followed a very similar pattern to the results of the case study presented in Section 5.3.

All of the evaluations were run with three different memory restrictions on the SA algorithm, 0%, 10% and 100%. However, we have only presented the results for 0%. This is because for the majority of our results, allowing the SA algorithm to add gaps between tasks had little effect. When changing the allowed memory overhead from 0% to 100%, the weighted measure for the baseline evaluation with distribution B only varied from 0.463 to 0.469. Because these values are close, the lines on the graphs are not shown as they are indistinguishable. This is due to a combination of factors, including the fact that the UCBs are scattered throughout the tasks and the high cache utilisation, which means there will always be a large number of conflicts.

### 5.4.3 Brute Force Comparison

As we found that allowing gaps between tasks did not significantly impact the breakdown utilisation, we compared the layouts produced by the SA algorithm against a brute force approach of trying every permutation of task ordering. As the majority of the computational effort goes to evaluating a layout using the schedulability test, the SA algorithm can be roughly compared against a brute force approach based on the number of layouts it evaluates. The number of layouts that must be evaluated for a taskset with $n$ tasks is equal to $n!$. With 7 tasks, evaluating every permutation results in 5040 (7!) different layouts, compared to the fixed 377 layouts[1] for the SA algorithm. This approach is feasible for 7 tasks, but becomes infeasible when the number of tasks increases.

[1] See Section 5.2 for an explanation of the SA algorithm, how many iterations it goes through, and why.

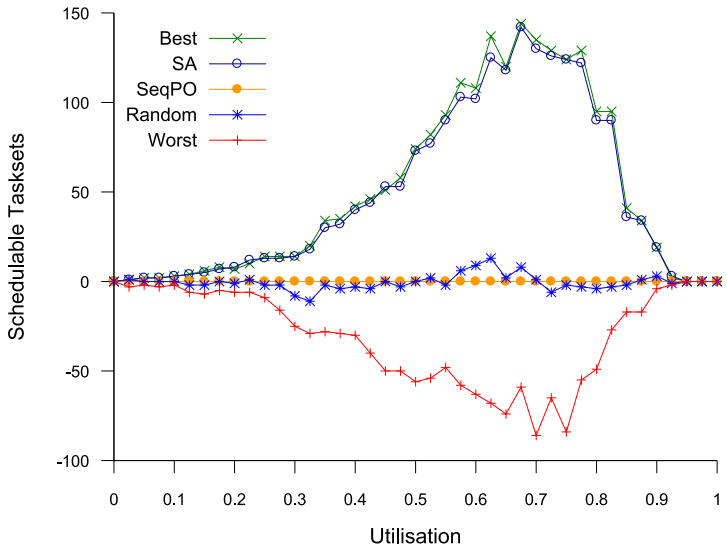



*Figure 5.14 - Comparing the SA algorithm at swapping tasks against a brute force approach of trying every permutation*

Figure 5.14 shows the results for 1000 tasksets normalised against the initial SeqPO layout. The value indicates the number of tasksets that were deemed schedulable under an approach relative to those deemed schedulable under SeqPO. The graph shows that while the SA algorithm does not always find an optimal layout, the layouts are near optimal and are discovered in significantly less time. At low utilisation levels, the variation in schedulable tasksets is very small, as almost all tasksets are schedulable regardless of task layout. Conversely, at high utilisation levels, all tasksets are unschedulable regardless of task layout.

## 5.5 Summary

In this chapter, we have presented a new technique that uses *simulated annealing* (SA) driven by CRPD aware schedulability analysis to find task layouts that increase system schedulability. This is important because the position of tasks in memory affects the *worst-case response time* of the tasks due to CRPD. While the SA algorithm did not always find the optimum solution, it did find a near optimal solution. We built functionality into our SA algorithm to add gaps between tasks in memory, but found that this had little effect on the schedulability of tasksets for all but the most trivial cases. The fact that adding gaps made little difference is beneficial for a number of reasons. Firstly, the search space is significantly reduced when just considering the order of tasks. Secondly, it is easier to setup a linker when combining object files to layout

tasks with no gaps between them. This is also an important practical point, in that it means that no additional memory space is required.

When no gaps are added between tasks we showed for 7 tasks that the SA algorithm was able to find a near optimal ordering of tasks; compared with a brute force approach which tried every permutation. We therefore did not focus on optimising the SA any further. However, alternative solutions such as using a genetic algorithm, instead of a SA, may be more suitable for the relatively flat search space as many layouts gave similar breakdown utilisations. The algorithm could also be improved by accounting for how much progress has made recently when determining whether to stop.

We evaluated our technique and showed that it was able to find layouts that allowed the tasksets to be schedulable at a higher utilisation level than other layouts. This included the default sequential layout with tasks ordered by priority (SeqPO). Using the default values for the parameters used to generate our synthetic tasksets, the layouts produced by the SA algorithm achieved a weighted schedulability measure of 0.465, compared to 0.377 for SeqPO. This is a significant difference as shown in Figure 5.8.

This work has a number of important uses. It can firstly be used when optimising an unschedulable taskset. If a layout can be found that makes the taskset schedulable then the problem is solved. Even if the taskset is still not schedulable, the work required to optimise the individual tasks and procedures to achieve schedulability will have been reduced. Alternatively, many embedded systems have stringent power usage requirements. It may be that an improved layout can allow the CPU and memory to be clocked at a lower frequency to reduce power usage, while still maintaining the schedulability of the taskset.

# CHAPTER 6. COMPARISON BETWEEN FP AND EDF

Two popular scheduling algorithms for real-time systems are FP and EDF. In this chapter we build on the work by Buttazzo [38] and use state of the art CRPD analysis for FP [7] and EDF to perform a comprehensive study of the performance of FP and EDF scheduling when accounting for CRPD. The analysis for FP [7] is discussed in Section 3.2 and the analysis for EDF is introduced in Chapter 4.

FP scheduling uses statically defined priorities to run the task with the highest priority first. In comparison, EDF is a dynamic scheduling algorithm that schedules the task with the earliest absolute deadline first. EDF is an optimal scheduling algorithm without pre-emption costs, whereas FP is not, and EDF is therefore typically able to schedule tasksets at a higher processor utilisation than FP [85]. However, despite the significant performance benefits over FP, EDF is not widely used in commercial real-time operating systems.

In 2005, Buttazzo [38] performed a detailed study of FP and EDF scheduling. This work covered both schedulability under a variety of scenarios, in addition to practical implementation considerations. Results showed that the FP scheduling algorithm introduces more pre-emptions than EDF, especially at high processor utilisation levels. This leads to FP performing worse than EDF. Yet, FP has an advantage over EDF, in that it is generally simpler to implement in commercial kernels which do not provide explicit support for timing constraints. Despite being a very detailed study, these comparisons where done under the assumption that there were no pre-emption costs due to CRPD.

# 6.1 Case Studies

In this section we compare the different approaches for calculating CRPD using a set of case studies based on PapaBench[1], the Mälardalen[2] benchmark suite and a set of SCADE[3] tasks. These are different from the single taskset case study used in Chapter 3 and 4. However, in all cases the system was set up to model the same ARM processor clocked at 100MHz with a 2KB direct-mapped instruction cache and a line size of 8 Bytes, giving 256 cache sets, 4 Byte instructions, and a BRT of 8μs.

## 6.1.1 Single Taskset Case Study

PapaBench is a real-time embedded benchmark based on the software of a GNU-license UAV, called Paparazzi. PapaBench contains two sets of tasks, *fly-by-wire* and *autopilot*. We used the *autopilot* tasks for which the WCETs, periods, UCBs, and ECBs were collected using aiT, see Table 6.1. We made the following assumptions in our evaluation to handle the interrupt tasks:

- Interrupts have higher priority than the normal tasks, but they cannot pre-empt each other
- Interrupts can occur at any time
- All interrupts have the same deadline which must be greater than or equal to the sum of their execution times in order for them to be schedulable
- The cache is disabled whenever an interrupt is executing and enabled again after it completes

In the case of FP scheduling the interrupts can be modelled as normal tasks with no UCBs or ECBs. Due to the interrupts having the same deadline, which is large enough to accommodate the interrupts execution times, no other changes need to be made to the analysis. For EDF scheduling a number of adjustments must be made to correctly account for the interrupts not being able to pre-empt each other. First we modify equation (4.12) to exclude interrupts when calculating the *processor demand*, h($t$). We then calculate the execution time of each interrupt, $I_x$, in the interval $t$ using equation (2) of [34]:

[1] http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97
[2] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[3] Esterel SCADE http://www.esterel-technologies.com/

$$h(I_x,t) = \left\lfloor \frac{t}{T_x} \right\rfloor \bullet C_x + \min\left( C_x, t - \left\lfloor \frac{t}{T_x} \right\rfloor T_x \right) \tag{6.1}$$

The result of which is then added onto the result of the modified version of equation (4.12), giving the processor demand for both tasks and interrupts as:

$$h(t) = \sum_{j=1}^{n} \left( \max\left\{ 0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor \right\} C_j + \gamma'_{t,j} \right) + \sum_{x=1} h(I_x,t) \tag{6.2}$$

We then adjust the upper bound $L$ used when checking $h(t)$. This is implemented by substituting $U = U^{tasks} + U^{interrupts}$ into equation (4.21) when calculating $L_d$ to give:

$$L_d = \frac{\left(U^{tasks} + U^{interrupts}\right) \bullet T_{max}}{\left(1 - \left(U^{tasks} + U^{interrupts} + U^{\gamma}\right)\right)} \tag{6.3}$$

Note that we leave $U^{\gamma}$ to represent the utilisation of the CRPD caused by just tasks. This is because we assume that the cache is disabled while the interrupts are executing and as such they cannot cause any CRPD.

We assigned a deadline of 2ms to all of the interrupt tasks, and implicit deadlines so that $D_i = T_i$, to the normal tasks. We then calculated the total utilisation for the system, and then effectively scaled the clock speed in order to reduce the total utilisation to the target utilisation for the system. We used the number of UCBs and ECBs obtained via analysis, placing the UCBs in a group at a random location in each task.

In each evaluation the taskset utilisation, not including pre-emption costs, was varied from 0.025 to 1 in steps of 0.001. For each utilisation value the schedulability of the taskset was determined under both FP and EDF. Specifically, we compared each scheduling algorithm (i) assuming no pre-emption cost, (ii) using CRPD analysis using the standard task layout, and (iii) using CRPD analysis after optimising the task layout using the approach presented in Chapter 5. The standard task layout is obtained by ordering tasks sequentially in memory based on their unique task indices.

Table 6.3 shows the breakdown utilisation for the single taskset based on PapaBench. There are a few interesting points to note. Firstly the breakdown utilisation is very high for both FP and EDF, this is due to the nearly harmonic periods and small range of task periods, with EDF outperforming FP. Secondly, the CRPD is very low when scheduled using either FP or EDF due to the small

number of UCBs. As the CRPD is very low, the layout optimisation makes little to no difference.

| Task | | UCBs | ECBs | WCET | Period |
|---|---|---|---|---|---|
| I4 | interrupt_modem | 2 | 10 | 0.303 ms | 100 ms |
| I5 | interrupt_spi_1 | 1 | 10 | 0.251 ms | 50 ms |
| I6 | interrupt_spi_2 | 1 | 4 | 0.151 ms | 50 ms |
| I7 | interrupt_gps | 3 | 26 | 0.283 ms | 250 ms |
| T5 | altitude_control | 20 | 66 | 1.478 ms | 250 ms |
| T6 | climb_control | 1 | 210 | 5.429 ms | 250 ms |
| T7 | link_fbw_send | 1 | 10 | 0.233 ms | 50 ms |
| T8 | navigation | 10 | 256 | 4.432 ms | 250 ms |
| T9 | radio_control | 0 | 256 | 15.681 ms | 25 ms |
| T10 | receive_gps_data | 22 | 194 | 5.987 ms | 250 ms |
| T11 | reporting | 2 | 256 | 12.222 ms | 100 ms |
| T12 | stabilization | 11 | 194 | 5.681 ms | 50 ms |

*Table 6.1 - Execution times, periods and number of UCBs and ECBs for the tasks from PapaBench*

| Source | Description | UCBs | ECBs | WCET |
|---|---|---|---|---|
| M | adpcm | 24 | 226 | 5.541 s |
| M | compress | 25 | 114 | 3.664 s |
| M | edn | 56 | 98 | 244.9 ms |
| M | fir | 28 | 50 | 21.53 ms |
| M | jfdctinit | 40 | 162 | 62.53 ms |
| M | ns | 17 | 26 | 73.38 ms |
| M | nsichneu | 53 | 256 | 149.6 ms |
| M | statemate | 3 | 256 | 77.96 ms |
| S | cruise control system | 25 | 107 | 1.959 s |
| S | flight control system | 70 | 256 | 2.138 s |
| S | navigation system | 45 | 82 | 1.409 s |
| S | stopwatch | 58 | 130 | 3.786 s |
| S | elevator simulation | 40 | 114 | 1.586 s |
| S | robotics systems | 68 | 256 | 4.311 s |

*Table 6.2 - Execution times and number of UCBs and ECBs for the largest benchmarks from the Mälardalen Benchmark Suite (M), and SCADE Benchmarks (S)*

| | Breakdown Utilisation |
|---|---|
| EDF - No Pre-emption Cost | 0.999 |
| EDF- Optimised Layout | 0.985 |
| EDF - Standard Layout | 0.985 |
| FP - No Pre-emption Cost | 0.977 |
| FP - Optimised Layout | 0.970 |
| FP - Standard Layout | 0.969 |

*Table 6.3 - Breakdown utilisation under the different approaches for the single PapaBench taskset*

### 6.1.2 Multiple Taskset Case Studies

The single taskset case study provides one specific example based on the PapaBench taskset. The remaining case studies used tasksets generated by randomly selecting tasks from a set of benchmarks. In the case of the PapaBench tasks, we treated the interrupts as normal tasks. We obtained tasksets by randomly selecting 10 tasks from Table 6.1, PapaBench benchmarks, or 10 tasks from Table 6.2, Mälardalen and SCADE benchmarks, or 15 tasks from the two tables, Mixed benchmarks. Using the UUnifast algorithm [32], we calculated the utilisation, $U_i$, of each task so that the utilisations added up to the desired utilisation level for the taskset. Based on the target utilisation and task execution times, $T_i$ was calculated such that $C_i = U_i T_i$. We used $D_i = y + x(T_i - y)$ to generate constrained deadlines, where $x$ is a random number between 0 and 1, and $y = \max(T_i/2, 2C_i)$. This generates constrained deadlines that are no less than half the period of the tasks. Note that allowing deadlines to be as small as $C_i$ would result in tasks that were unschedulable once CRPD were introduced. We used the number of UCBs and ECBs obtained using aiT, and placed the UCBs in a group at a random location in each task.

We generated 1000 tasksets for the multiple taskset case studies and evaluated them using the same method as the single taskset case study. The only difference was that we varied the utilisation excluding pre-emption costs from 0.025 to 1 in steps of 0.0125.

#### PapaBench Benchmark

The tasks in the PapaBench benchmarks are simple, short control tasks with limited computations and data accesses. Figure 6.1 shows the percentage of tasksets that were deemed schedulable by each approach for the 1000 tasksets of cardinality 10 that we randomly selected from Table 6.1. The results are similar to those obtained using the single taskset PapaBench case study. Specifically, EDF outperformed FP as it deemed a higher number of tasksets schedulable at each utilisation level. Because the range of execution times is relatively small, so is the typical range of task periods for the generated tasksets. Hence the number of pre-emption is also relatively small. Furthermore, the number of UCBs is small resulting in low CRPD. Therefore the task layout optimisation was only able to make a small improvement, but did so for both FP and EDF.

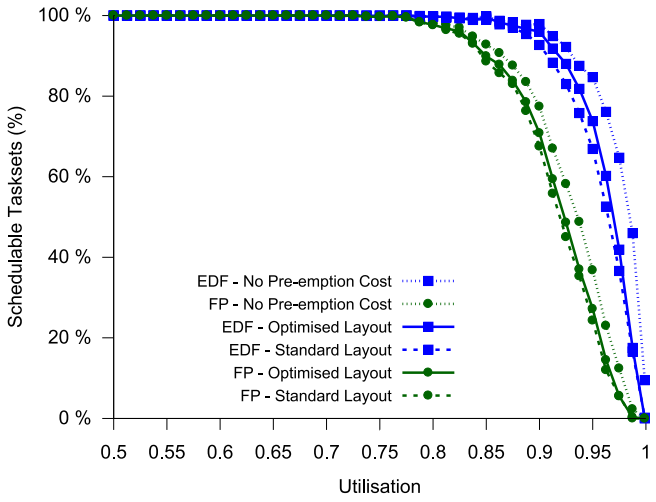



*Figure 6.1 - Percentage of schedulable tasksets at each utilisation level for the PapaBench benchmark for tasksets of cardinality 10*

### Mälardalen and SCADE Benchmarks

The second multiple taskset case study was based on tasks from the Mälardalen and SCADE benchmarks, shown in Table 6.2. Compared to the tasks from PapaBench, these tasks have higher execution times, high amounts of computation, and a larger number of UCBs. Figure 6.2 shows the percentage of tasksets that were deemed schedulable by each approach for the 1000 tasksets of cardinality 10 that we randomly selected from Table 6.2. As with the PapaBench benchmarks, EDF outperformed FP scheduling as it has a higher percentage of schedulable tasksets at each utilisation level. Likewise, because the range of task periods was also relatively small, CRPD is minimised.

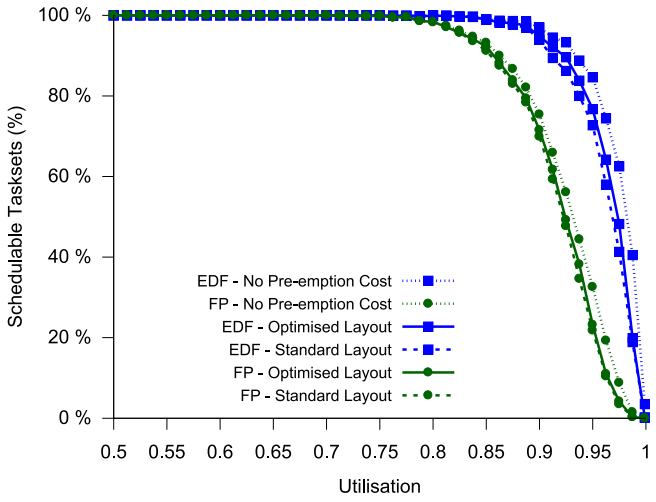



*Figure 6.2 - Percentage of schedulable tasksets at each utilisation level for the Mälardalen and SCADE benchmarks for tasksets of cardinality 10*

### Mixed Benchmark

The third multiple taskset case study was based on a mixture of the small and short PapaBench tasks, and the large and long Mälardalen and SCADE tasks. Here the tasksets had 15 tasks each and represent systems with background tasks combined with short control tasks. As we mixed tasks from both tables, it also allowed us to generate tasksets with a higher number of tasks.

The results, shown in Figure 6.3, show that when a taskset contains tasks with a wide range of periods CRPD can become a significant factor in the schedulability of the taskset. This is because short high priority tasks are able to pre-empt long running low priority tasks multiple times.

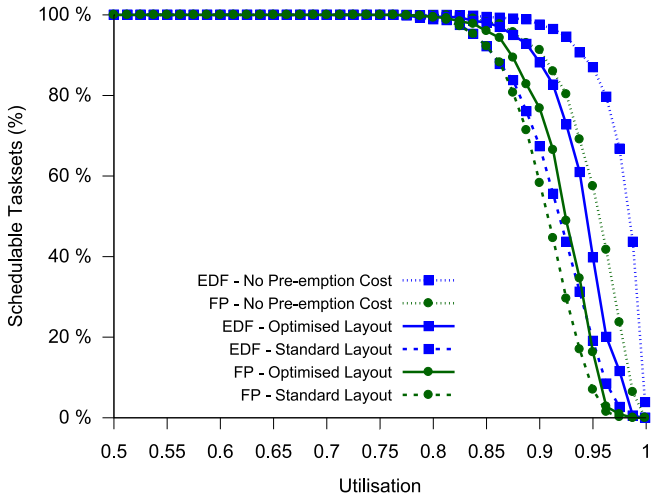



*Figure 6.3 - Percentage of schedulable tasksets at each utilisation level for the mixed case study with tasks randomly selected from both the PapaBench and Mälardalen and SCADE benchmarks (taskset cardinality 15)*

While EDF still outperformed FP, the gain in schedulability of using EDF over FP was diminished once CRPD was taken into account. Optimising the task layout resulted in a significant improvement for both FP and EDF, showing the task layout optimisation can be effectively applied to both EDF and FP scheduling. Furthermore, by optimising the task layout, FP was able to schedule a similar number of tasksets to EDF with the standard layout. In other words, in cases where the CRPD is relatively high, selecting an optimised task layout can be as effective as changing the scheduling algorithm. The results are summarised in Table 6.4 using weighted schedulability measures, as discussed in Section 3.5.2. They show that for these tasksets, FP with an optimised layout achieved a weighted measure of 0.784, outperforming EDF with the standard layout as it achieved a weighted measure of 0.771.

| | Weighted Schedulability |
|---|---|
| EDF - No Pre-emption cost | 0.922 |
| FP - No Pre-emption cost | 0.855 |
| EDF - Optimised layout | 0.830 |
| EDF - Standard layout | 0.771 |
| FP - Optimised layout | 0.784 |
| FP - Standard layout | 0.747 |

*Table 6.4 - Weighted schedulability measures for the mixed case study shown in Figure 6.3.The higher the weighted schedulability measure, the more tasksets deemed schedulable by the approach*

## 6.2 Evaluation

In addition to the case studies based on the PapaBench, Mälardalen and SCADE benchmarks, we evaluated FP and EDF with CRPD analysis using synthetically generated tasksets. This enabled us to investigate the effect of varying key parameters under each scheduling algorithm.

The UUnifast algorithm [32] was again used to calculate the utilisation, $U_i$ of each task so that the utilisations added up to the desired utilisation level for the taskset. Task periods $T_i$, were generated at random between 5ms and 500ms according to a log-uniform distribution. $C_i$ was then calculated via $C_i = U_i T_i$.

We generated two sets of tasksets, one with implicit deadlines and one with constrained deadlines. In the following section, we present the results for constrained deadline tasksets. In general, the results for implicit deadline tasksets gave a higher number of schedulable tasksets for every approach compared to the constrained deadline tasksets. Additionally, the task layout had a similar or slightly larger effect on schedulability in relation to the chosen scheduling algorithm.

The UCB percentage for each task was based on a random number between 0 and a maximum UCB percentage specified for the evaluation. UCBs were split into N groups, where N was chosen at random between 1 and 5, and placed at a random starting point within the task's ECBs.

### 6.2.1 Baseline Evaluation

To investigate the effect of key cache and taskset configurations we varied the following parameters:

- Cache utilisation (default of 10)

- Maximum UCB percentage (default of 30%)
- Number of tasks (default of 15)
- Block Reload Time (BRT) (default of 8μs)
- Period range (default of [5, 500]ms)

We used 1,000 randomly generated tasksets for the evaluation.

In addition to testing the different analyses as done for the case study, we also performed a simulation of the schedule for the tasksets[1]. Our aim with the simulation was to minimise schedulability by maximising the number of pre-emptions. As noted in previous chapters, traditional methods for generating the worst case arrival pattern will not necessarily generate them in the presence of CRPD. For FP the simulation tested each task $\tau_i$ in turn by releasing it at time $t = 0$. It then released all of the other tasks that have a higher priority than task $\tau_i$, sorted by lowest to highest priority, at $t = 1$, $t = 2$, $t = 3$ etc… If all tasks were schedulable it also performed the same test, but instead of staggering the other tasks, released them at random. For EDF we tried to maximise pre-emptions by releasing tasks so that their deadlines were staggered. The first step is to determine the interval that needs to be checked, $L$, which can be achieved by using equation (4.24). Then for each task $\tau_i$ in turn, we scheduled a job of task $\tau_i$ so that it has a deadline at $t = L$. We then scheduled a job of all of the other tasks, sorted by longest to shortest deadline, so that they have their deadlines at $t = L - 1$, $t = L - 2$, $t = L - 3$ etc… Based on the final jobs' deadlines we then calculated when the first jobs for each task need to be released. If all tasks are schedulable, we repeated the process using $t = L - 1$ for all of the other tasks' jobs, and also using a random schedule.

The results for the baseline configuration are shown in Figure 6.4 and are summarised in Table 6.5 using weighted schedulability measures. The results follow a similar pattern to the results from the case study. EDF outperformed FP finding a higher number of tasksets schedulable. The results for the simulations show that the CRPD affects both FP and EDF, with the CRPD being slightly lower for EDF. Specifically, the simulation shows that CRPD reduced the weighted measure by at least 0.129 for EDF, 0.925-0.795, and 0.141 for FP, 0.774-0.633, in this case. However, once the CRPD obtained via analysis is taken into account, the performance gains of using EDF over FP are diminished. This is most likely caused by increased pessimism in the CRPD analysis for EDF. The

[1] Note that the simulation effectively provides a necessary, but not sufficient test of schedulability

results also showed that the layout optimisation improved the schedulability of tasksets scheduled under both FP and EDF.

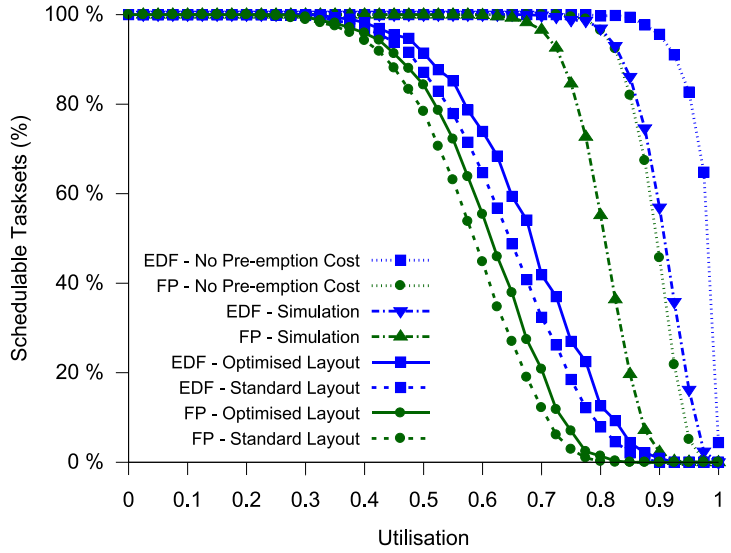



*Figure 6.4 - The percentage of schedulable tasksets at each utilisation level for the baseline configuration (taskset cardinality 15)*

| | Weighted Schedulability |
|---|---|
| EDF - No Pre-emption cost | 0.925 |
| EDF - Simulation | 0.796 |
| FP - No Pre-emption cost | 0.774 |
| FP - Simulation | 0.633 |
| EDF - Optimised layout | 0.455 |
| EDF - Standard layout | 0.413 |
| FP - Optimised layout | 0.369 |
| FP - Standard layout | 0.336 |

*Table 6.5 - Weighted schedulability measures for the baseline configuration study shown in Figure 6.4. The higher the weighted schedulability measure, the more tasksets deemed schedulable by the approach*

### 6.2.2 Detailed Evaluation

Evaluating all combinations of different task parameters is not possible. Therefore, the majority of our evaluations focused on varying one parameter at a time. To present these results, weighted schedulability measures [21] are used, which are described in Section 4.5.2.

#### Cache Utilisation

As the cache utilisation increases the likelihood of tasks evicting each other from cache increases, this causes higher CRPD reducing the number of schedulable tasksets. It can be seen in Figure 6.5 that task layout optimisation is

effective for FP and EDF across the same range of cache utilisations. In both cases it becomes less effective once the cache utilisation becomes high. We note that because the number of tasks is fixed, that the average size of the tasks is equal to the cache utilisation divided by the number of tasks. This means that as the cache utilisation increases, so does the size of the tasks and therefore, the number of UCBs. This in turn makes it harder to find an improved layout.

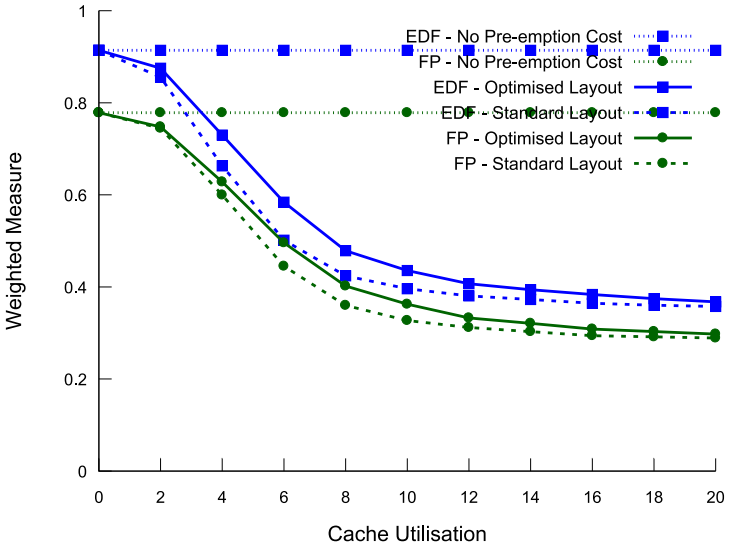



*Figure 6.5 - Weighted measure for varying the cache utilisation from 0 to 20 in steps of 2*

### Maximum UCB Percentage

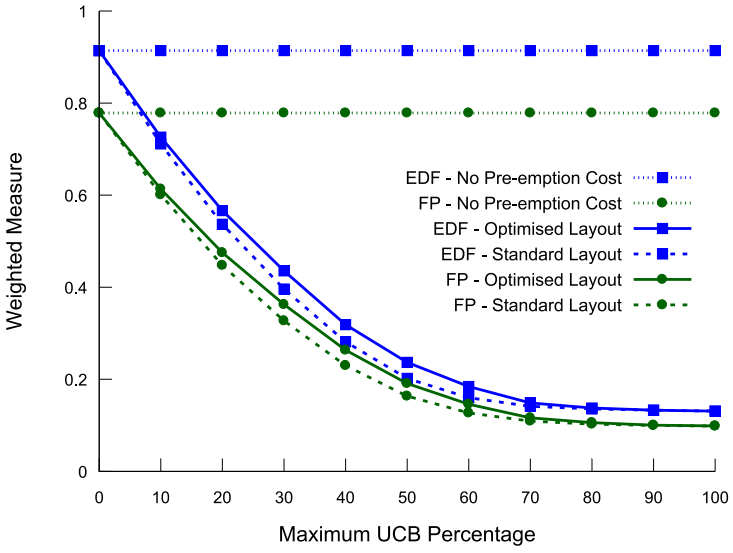



*Figure 6.6 - Weighted measure for varying the maximum UCB percentage from 0 to 100 in steps of 10*

As the maximum UCB percentage increases, the CRPD increases resulting in a reduction in the number of tasksets that are deemed schedulable, as can be seen

in Figure 6.6. With a low percentage of UCBs, the CRPD is low which means there is little benefit from layout optimisation. When the UCB percentage is very high there are a significant number of conflicts that there is very little that can be done to improve the layout. When the maximum UCB percentage is at 40-60% there is a notable amount of CRPD, but there is also room for the task layout algorithm to optimise the layout. This allows FP using an optimised task layout to schedule a similar number of tasksets as EDF using the standard layout.

### Number of Tasks

When varying the number of tasks, as seen in Figure 6.7, we scaled the cache utilisation to keep the average size of tasks constant based on a cache utilisation of 10 for 15 tasks. This is because it would be unrealistic for the size of tasks to decrease as more tasks are added to the system. Hence with 8 tasks the cache utilisation is equal to 5.33, whereas for 32 tasks, it is equal to 21.33. As the number of tasks increases, it becomes harder to schedule all tasks. This leads to a decrease in the overall weighted measure. The task layout optimisation performs best when there is a moderate number of tasks as there are enough conflicts that optimising the layout can give an improvement; but not so many that there is nothing that can be done to avoid the conflicts.

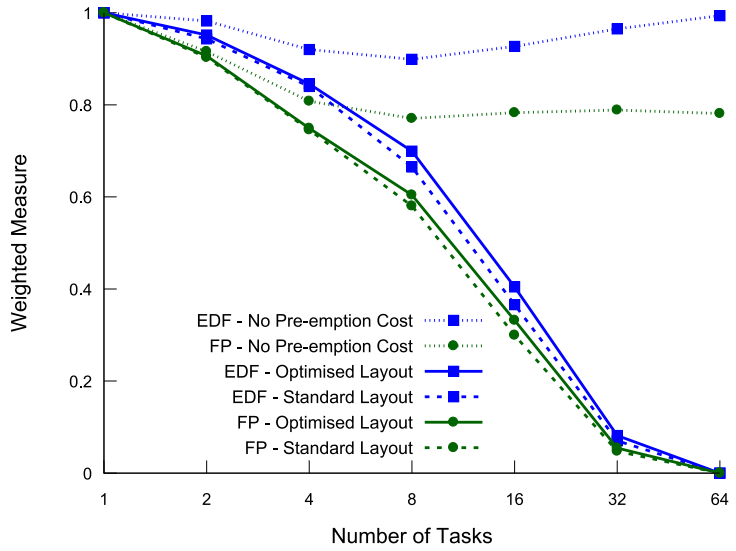



*Figure 6.7 - Weighted measure for varying the number of tasks from $2^0$ to $2^6$ while maintaining a constant ratio of number of tasks to cache utilisation*

### Block Reload Time

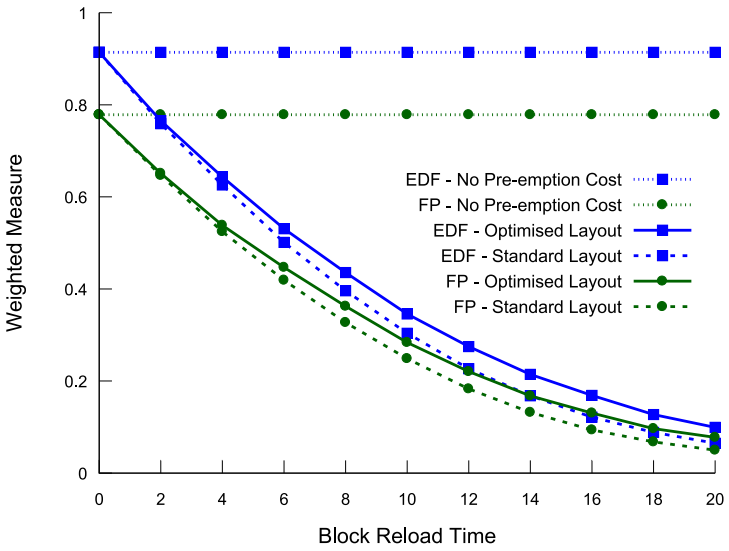



*Figure 6.8 - Weighted measure for varying the block reload time (BRT) from 0 to 20µs in steps of 2*

As the block reload time is increased, it becomes more costly to reload a block, which causes an increase in CRPD. It can be seen in Figure 6.8 that as the block reload time is increased, the analysis that takes into account the pre-emption cost shows a decrease in the overall weighted measure. We note that as the cost of reloading a block increases, the potential gains of optimising the layout increases. Once the block reload time exceeds 14µs, using an optimised layout under FP scheduling outperforms using a standard layout under EDF scheduling.

### Period Range

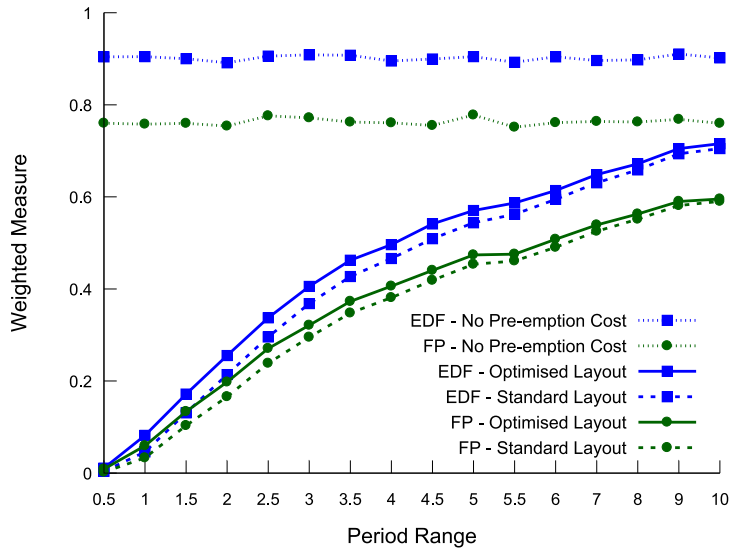



*Figure 6.9 - Weighted measure for varying the scaling factor used to generate periods, w, in w[1, 100]ms, from 0.5 to 10*

We also investigated the effect of the scaling factor used to generate task periods, to simulate tasksets with shorter to longer execution times. We varied the scaling factor, $w$, from 0.5 to 10 and hence the range of task periods given by $w$[1, 100]ms. A lower scaling factor resembles tasks with shorter execution times, as seen in the PapaBench benchmark. A higher scaling factor resembles tasks with higher execution times and commensurately longer periods, as seen in the Mälardalen and SCADE benchmarks. The results in Figure 6.9 show the layout optimisation performs best when task periods are relatively short, as that is when the pre-emption costs are highest. Once the period range is greater than [10, 1000]ms, the relative pre-emption costs are low enough that performing the layout optimisation only makes a very small improvement on the schedulability of the tasksets.

## 6.3 Summary

The EDF scheduling algorithm is an optimal scheduling algorithm assuming negligible pre-emption costs for single processors. However, it has been largely disregarded by industry. Whereas FP despite offering lower theoretical schedulable processor utilisation, is relatively popular with many commercial real- time operating systems supporting it.

Previous work by Buttazzo [38] has compared the two algorithms, but it did not take into account CRPD which can have a significant effect on the schedulability of a taskset.

In this chapter we performed a detailed comparison of FP and EDF taking into account CRPD using state-of-the-art CRPD analysis for FP [7], and EDF, presented in Chapter 3. This showed the feasibility of simple, yet effective, task layout optimisation techniques for EDF. We found that when CRPD is considered, the performance gains offered by EDF over FP, while still significant, are somewhat diminished. This is most likely due to greater pessimism in the CRPD analysis for EDF than FP. We also discovered that in configurations that cause relatively high CRPD, optimising task layout can be just as effective as changing the scheduling algorithm from FP to EDF. This is important in an industrial setting as it is considerably simpler and cheaper to control the task layout via the linker than it is to change the scheduler. Nevertheless, our evaluations showed that changing to an EDF scheduler and optimising the task layout provides a gain over FP scheduling. Although this gain was not as pronounced as the advantage that EDF has over FP when pre-emption costs are not accounted for via analysis.

# CHAPTER 7. CRPD ANALYSIS FOR HIERARCHICAL SCHEDULING

There is a growing need in industry to combine multiple applications together to build complex embedded real-time systems. This is driven by the need to re-use legacy applications that once ran on slower, but dedicated processors. Typically, it is too costly to go back to the design phase resulting in a need to use applications as-is. Furthermore, there are often a number of vendors involved in implementing today's complex embedded real-time systems, each supplying separate applications which must then be integrated together. Hierarchical scheduling provides a means of composing multiple applications onto a single processor, such that the temporal requirements of each application are met. Each application, or component, has a dedicated server. A global scheduler then allocates processor time to each server, during which the associated component can use its own local scheduler to schedule its tasks.

In pre-emptive multi-tasking systems, CRPD is caused by the need to re-fetch cache blocks belonging to the pre-empted task which were evicted from the cache by the pre-empting task. This is further complicated when using hierarchical scheduling as servers will often be suspended while their components' tasks are still active. In this case they have started, but have not yet completed executing. While a server is suspended the cache can be polluted by the tasks belonging to other components. When the global scheduler then switches back to the first server, tasks belonging to the associated component may have to reload blocks into cache that were in use before the global context switch.

In this chapter we present new analysis that bounds the CRPD caused by blocks being evicted from cache by other components in hierarchical systems. The

analysis is for a hierarchical system with a global non-pre-emptive scheduler and a local pre-emptive *Fixed Priority* (FP) or *Earliest Deadline First* (EDF) scheduler.

### Related Work on Hierarchical Scheduling

Hierarchical scheduling has been studied extensively in the past 15 years. Deng and Liu [56] were the first to propose such a two-level scheduling approach. Later Feng and Mok [60] proposed the resource partition model and schedulability analysis based on the supply bound function. Shih and Lee [111] introduced the concept of a temporal interface and the periodic resource model, and refined the analysis of Feng and Mok. Kuo and Li [76] and Saewong *et al*. [108] specifically focused on fixed priority hierarchical scheduling. Lipari and Bini [83] solved the problem of computing the values of the partition parameters to make an application schedulable. Davis and Burns [50] proposed a method to compute the response time of tasks running on a local fixed priority scheduler. Later, Davis and Burns [49] investigated selecting optimal server parameters for fixed priority pre-emptive hierarchical systems. When using a local EDF scheduler Lipari *et al.* [82] [84] investigated allocating server capacity to components, proposing an exact solution. Recently Fisher and Dewan [64] developed a polynomial-time approximation with minimal over provisioning of resources.

Hierarchical systems have been used mainly in the avionics industry. The *Integrated Modular Avionics* (IMA) [119] [10] is a set of standard specifications for simplifying the development of avionics software. Among other requirements it allows different independent applications to share the same hardware and software resources [11]. The ARINC 653 standard [11] defines temporal partitioning for avionics applications. The global scheduler is a simple *Time Division Multiplexing* (TDM), in which time is divided into frames of fixed length, each frame is divided into slots and each slot is assigned to one application.

## 7.1 System Model Extension

In this section we describe the extension to our system model presented in Section 2.1.1 for hierarchical scheduling.

We assume a single processor system comprising $m$ applications or components, each with a dedicated server ($S^1..S^m$) that allocates processor

capacity to it. We use $\Psi$ to represent the set of all components in the system. $G$ is used to indicate the index of the component that is being analysed. Each server $S^G$ has a budget $Q^G$ and a period $P^G$, such that the associated component will receive $Q^G$ units of execution time from its server every $P^G$ units of time. Servers are assumed to be scheduled globally using a non-pre-emptive scheduler, as found in systems that use time partitioning to divide up access to the processor. While a server has remaining capacity and is allocated the processor, we assume that the tasks of the associated component are scheduled according to the local scheduler policy. If there are no tasks in the associated component to schedule, we assume that the processor idles until the server exhausts all of its capacity, or a new task in the associated component is released.

The system comprises a taskset $\Gamma$ made up of a fixed number of tasks ($\tau_1..\tau_n$) divided between the components. Each component contains a strict subset of the tasks, represented by $\Gamma^G$. For simplicity, we assume that the tasks are independent and do not share resources requiring mutually exclusive access, other than the processor. We note that global and local resource sharing has been extensively studied for hierarchical systems [51] [23] [13]. Resource sharing and its effects on CRPD have also been studied for single level systems [6] [7].

In the case of a local FP scheduler, we use the notation hp($G$,$i$) and hep($G$,$i$) to restrict hp($i$) and hep($i$) to just tasks of component $G$.

Each component $G$ also has a set of UCBs, $UCB^G$ and a set of ECBs, $ECB^G$, that contain respectively all of the UCBs, and all of the ECBs, of the associated tasks, $\mathrm{UCB}^G = \bigcup_{\forall \tau_i \in \Gamma^G} \mathrm{UCB}_i$ and $\mathrm{ECB}^G = \bigcup_{\forall \tau_i \in \Gamma^G} \mathrm{ECB}_i$.

## 7.2 Hierarchical Schedulability Analysis

Hierarchical scheduling is a technique that allows multiple independent components to be scheduled on the same system. A global scheduler allocates processing resources to each component via server capacity. Each component can then utilise the server capacity by scheduling its tasks using a local scheduler.

### Supply Bound Function

In hierarchical systems components do not have dedicated access to the processor, but must instead share it with other components. The *supply bound function* [111], or specifically the inverse of it, can be used to determine the maximum amount of time needed by a specific server to supply some capacity *c*.

Figure 7.1 shows an example for server $S^G$ with $Q^G$ = 5 and $P^G$ = 8. Here we assume the worst case scenario where a task is activated just after the server's budget is exhausted. In this case the first instance of time at which tasks can receive some supply is at $2(P^G - Q^G) = 6$.

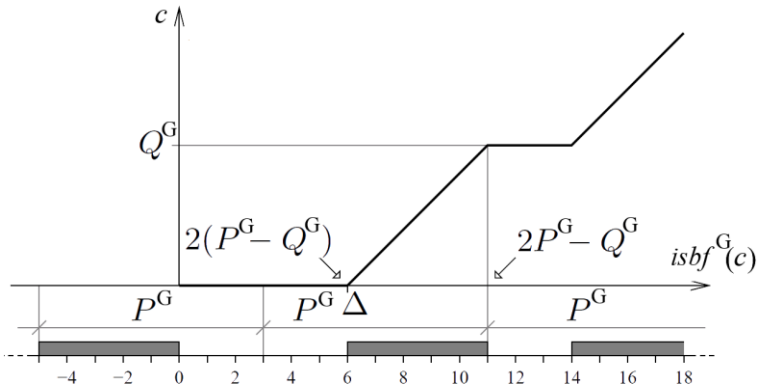



*Figure 7.1 – Example showing how server capacity can be supplied to components. General case of a server where $Q^G$ = 5 and $P^G$ = 8 showing it can take up to 6 time units before a task receives supply*

We define the *inverse supply bound function*, *isbf,* for component *G* as $isbf^G$ [106]:

$$isbf^G(c) = c + (P^G - Q^G)\left(\left\lceil \frac{c}{Q^G} \right\rceil + 1\right) \tag{7.1}$$

In order to account for component level CRPD we must define two terms. We use $E^G(t)$ to denote the maximum number of times server $S^G$ can be both suspended and resumed within an interval of length *t*:

$$E^G(t) = 1 + \left\lfloor \frac{t}{P^G} \right\rfloor \tag{7.2}$$

Figure 7.2 shows an example global schedule for three components, *G*, *Z* and *Y*. When $t > 0$ server $S^G$ can be suspended and resumed at least once. Then for each increase in *t* by $P^G$, server $S^G$ could be suspended and resumed one additional time per increase in *t* by $P^G$. We note that technically the number of times a server can be both suspended and resumed increases by one at $t = P^G + 2$,

$t = (2 \times P^G) + 2$, etc… Therefore equation (7.2) is a conservative bound on the number of times that a server is both suspended and resumed within an interval of length $t$.

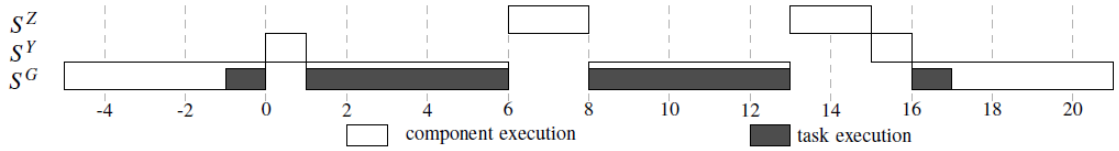



*Figure 7.2 - Example global schedule to illustrate the server suspend and resume calculation with* $P^G = P^Z = P^Y = 8$, $Q^G = 5$, $Q^Z = 2$, $Q^Y = 1$

We use the term *disruptive execution* to describe an execution of server $S^Z$ while server $S^G$ is suspended that results in tasks from component $Z$ evicting cache blocks that tasks in component $G$ may have loaded and need to reload. Note that if server $S^Z$ runs more than once while server $S^G$ is suspended, its tasks cannot evict the same blocks twice. As such, the number of disruptive executions is bounded by the number of times that server $S^G$ can be both suspended and resumed, $E^G(t)$. We use $X^Z$ to denote the maximum number of such disruptive executions.

$$X^Z(S^G, t) = \min\left(E^G(t), 1 + \left\lceil \frac{t}{P^Z} \right\rceil\right) \tag{7.3}$$

Figure 7.3 shows an example global schedule for components $G$ and $Z$. Between $t$=0 and $t$=6, component $Z$ executes twice, but can only evict cache blocks that tasks in component $G$ might have loaded and need to reload once.

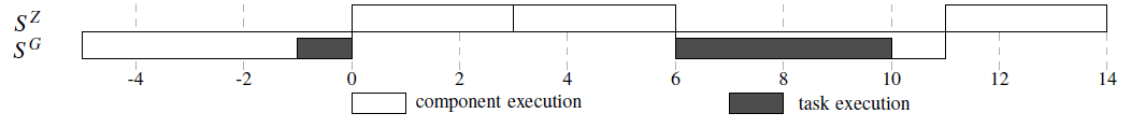



*Figure 7.3 - Example global schedule to illustrate the disruptive execution calculation with* $P^G = P^Z = 8$, $Q^G = 5$, $Q^Z = 3$

## 7.3 CRPD Analysis for Hierarchical Systems: Local FP Scheduler

In this section, we describe how CRPD analysis can be extended for use in hierarchical systems with a local FP scheduler and integrated into the schedulability analysis for it. We do so by extending the concepts of ECB-Only, UCB-Only, UCB-Union and UCB-Union Multiset analysis introduced in [37],

[77], [115] and [7], described in Section 3.2, respectively to hierarchical systems. This analysis assumes a non-pre-emptive global scheduler such that the capacity of a server is supplied without pre-emption, but may be supplied starting at any time during the server's period. It assumes that tasks are scheduled locally using a pre-emptive fixed priority scheduler. We explain a number of different methods, building up in complexity.

The analysis needs to capture the cost of reloading any UCBs into cache that may be evicted by tasks belonging to other components; in addition to the cost of reloading any UCBs into cache that may be evicted by tasks in the same component. For calculating the intra-component CRPD, we use the Combined Multiset approach by Altmeyer *et al.* [7], which is described in Section 3.2. This can be achieved by combining the intra-component CRPD due to pre-emptions between tasks within the same component via the Combined Multiset approach, equation (3.6), with modified response time analysis for non-dedicated processor access, with a new term, $\gamma_i'^{G}$ :

$$R_i^{\alpha+1} = isbf^G\left(C_i + \sum_{\forall j \in \mathrm{hp}(G,i)}\left(\left\lceil \frac{R_i^{\alpha} + J_j}{T_j} \right\rceil C_j + \gamma'_{i,j}\right) + \gamma_i'^{G}\right) \tag{7.4}$$

Here, $\gamma_i'^{G}$ represents the CRPD on task $\tau_i$ in component $G$ caused by tasks in the other components running while the server, $S^G$, for component $G$ is suspended. Use of the inverse supply bound function gives the response time of $\tau_i$ under server, $S^G$, taking into account the shared access to the processor.

## ECB-Only

A simple approach to calculate component CPRD is to consider the maximum effect of the other components by assuming that every block evicted by the tasks in the other components has to be reloaded. There are two different ways to calculate this cost.

### ECB-Only-All

The first option is to assume that every time server $S^G$ is suspended, all of the other servers run and their tasks evict all the cache blocks that they use. We therefore take the union of all ECBs belonging to the other components to get the number of blocks that could be evicted. We then sum them up $E^G(R_i)$ times, where $E^G(R_i)$ upper bounds the number of times server $S^G$ could be both suspended and resumed during the response time of task $\tau_i$, see equation (7.2). We can calculate the CRPD impacting task $\tau_i$ of component $G$ due to the other components in the system as:

$$\gamma_i'^{G} = \text{BRT} \bullet E^{G}(R_i) \bullet \left| \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \text{ECB}^{Z} \right| \tag{7.5}$$

### ECB-Only-Counted

The above approach works well when the global scheduler uses a TDM schedule, such that each server has the same period and/or components share a large number of ECBs. If some servers run less frequently than server $S^G$, then the number of times that their ECBs can evict blocks may be over counted. One solution to this problem is to consider each component separately. This is achieved by calculating the number of disruptive executions that server $S^Z$ can have on task $\tau_i$ in component $G$ during the response time of task $\tau_i$, given by $X^{Z}\left(S^{G}, R_i\right)$, see equation (7.3). We can then calculate an alternative bound for the CRPD incurred by task $\tau_i$ of component $G$ due to the other components in the system as:

$$\gamma_i'^{G} = \text{BRT} \bullet \sum_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left( X^{Z}\left(S^{G}, R_i\right) \bullet \left| \text{ECB}^{Z} \right| \right) \tag{7.6}$$

Note that the ECB-Only-All and ECB-Only-Counted approaches are incomparable.

## UCB-Only

Alternatively we can focus on the tasks in component $G$, hence calculating which UCBs could be evicted if the entire cache was flushed by the other components in the system. However, task $\tau_i$ may have been pre-empted by higher priority tasks. So we must bound the pre-emption cost by considering the number of UCBs over all tasks in component $G$ that may pre-empt task $\tau_i$, and task $\tau_i$ itself, given by $\tau_k \in \text{hep}(G, i)$.

$$\bigcup_{\forall k \in \text{hep}(G,i)} \text{UCB}_k \tag{7.7}$$

We multiply the number of UCBs, equation (7.7), by the number of times that server $S^G$ can be both suspended and resumed during the response time of task $\tau_i$ to give:

$$\gamma_i'^{G} = \text{BRT} \bullet E^{G}(R_i) \bullet \left| \bigcup_{\forall k \in \text{hep}(G,i)} \text{UCB}_k \right| \tag{7.8}$$

This approach is incomparable with the ECB-Only-All and ECB-Only-Counted approaches.

## UCB-ECB

While it is a sound to only consider the ECBs of the tasks in the other components, or only the UCBs of the tasks in the component of interest, these approaches are clearly pessimistic. We can tighten the analysis by considering both.

### UCB-ECB-All

We build upon the ECB-Only-All and UCB-Only methods. For task $\tau_i$ and all tasks that could pre-empt it in component $G$, we first calculate which UCBs could be evicted by the tasks in the other components, this is given by equation (7.7). We then take the union of all ECBs belonging to the other components to get the number of blocks that could potentially be evicted. We then calculate the intersection between the two unions to give an upper bound on the number of UCBs evicted by the ECBs of the tasks in the other components.

$$\left| \left( \bigcup_{\forall k \in \mathrm{hep}(G,i)} \mathrm{UCB}_k \right) \cap \left( \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \mathrm{ECB}^Z \right) \right| \tag{7.9}$$

This is then multiplied by the number of times that the server $S^G$ could be both suspended and resumed during the response time of task $\tau_i$ to give:

$$\gamma_i'^G = \mathrm{BRT} \bullet E^G(R_i) \bullet \left| \left( \bigcup_{\forall k \in \mathrm{hep}(G,i)} \mathrm{UCB}_k \right) \cap \left( \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \mathrm{ECB}^Z \right) \right| \tag{7.10}$$

By construction, the UCB-ECB-All approach dominates the ECB-Only-All and UCB-Only approaches.

### UCB-ECB-Counted

Alternatively, we can consider each component in isolation by building upon the ECB-Only-Counted and UCB-Only approaches. For task $\tau_i$ and all tasks that could pre-empt it in component $G$, we start by calculating an upper bound on the number of blocks that could be evicted by component $Z$:

$$\left| \left( \bigcup_{\forall k \in \text{hep}(G,i)} \text{UCB}_k \right) \cap \text{ECB}^Z \right| \tag{7.11}$$

We then multiply this number of blocks by the number of disruptive executions that server $S^Z$ can have during the response time of task $\tau_i$, and sum this up for all components to give:

$$\gamma_i'^G = \text{BRT} \bullet \sum_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left( X^Z\left(S^G, R_i\right) \bullet \left| \left( \bigcup_{\forall k \in \text{hep}(G,i)} \text{UCB}_k \right) \cap \text{ECB}^Z \right| \right) \tag{7.12}$$

By construction, the UCB-ECB-Counted approach dominates the ECB-Only-Counted approach, but is incomparable with the UCB-Only approach.

## UCB-ECB-Multiset

The UCB-ECB approaches are pessimistic in that they assume that each component can, directly or indirectly, evict UCBs of each task $\tau_k \in \text{hep}(G,i)$ in component $G$ up to $E^G(R_i)$ times during the response time of task $\tau_i$. While this is potentially true when $\tau_k = \tau_i$, it can be a pessimistic assumption in the case of intermediate tasks which may have much shorter response times. The UCB-ECB-Multiset approaches, described below, remove this source of pessimism by upper bounding the number of times intermediate task $\tau_k \in \text{hep}(G,i)$ can run during the response time of $\tau_i$. They then multiply this value by the number of times that the server $S^G$ can be both suspended and resumed during the response time of task $\tau_k$, $E^G(R_k)$.

### UCB-ECB-Multiset-All

First we form a multiset $M_{G,i}^{ucb}$ that contains the UCBs of task $\tau_k$ repeated $E^G(R_k)E_k(R_i)$ times for each task $\tau_k \in \text{hep}(G,i)$. This multiset reflects the fact that the UCBs of task $\tau_k$ can only be evicted and reloaded $E^G(R_k)E_k(R_i)$ times during the response time of task $\tau_i$ as a result of server $S^G$ being suspended and resumed.

$$M_{G,i}^{ucb} = \bigcup_{\forall k \in \text{hep}(G,i)} \left( \bigcup_{E^G(R_k)E_k(R_i)} \text{UCB}_k \right) \tag{7.13}$$

Then we form a second multiset $M_{G,i}^{ecb-A}$ that contains $E^G(R_i)$ copies of the ECBs of all of the other components in the system. This multiset reflects the fact that the other servers' tasks can evict blocks that may subsequently need to be reloaded at most $E^G(R_i)$ times within the response time of task $\tau_i$.

$$M_{G,i}^{ecb-A} = \bigcup_{E^G(R_i)} \left( \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \mathrm{ECB}^Z \right) \tag{7.14}$$

The total CRPD incurred by task $\tau_i$, in component $G$ due to the other components in the system is then bounded by the size of the multiset intersection of $M_{G,i}^{ucb}$, equation (7.13), and $M_{G,i}^{ecb-A}$, equation (7.14).

$$\gamma_i'^G = \mathrm{BRT} \bullet \left| M_{G,i}^{ucb} \cap M_{G,i}^{ecb-A} \right| \tag{7.15}$$

### UCB-ECB-Multiset-Counted

For the UCB-ECB-Multiset-Counted approach, we keep equation (7.13) for calculating the set of UCBs; however, we form a second multiset $M_{G,i}^{ecb-C}$ that contains $X^Z\left(S^G, R_i\right)$ copies of the ECBs of each other component $Z$ in the system. This multiset reflects the fact that tasks of each server $S^Z$ can evict blocks at most $X^Z\left(S^G, R_i\right)$ times within the response time of task $\tau_i$.

$$M_{G,i}^{ecb-C} = \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left( \bigcup_{X^Z\left(S^G, R_i\right)} \mathrm{ECB}^Z \right) \tag{7.16}$$

The total CRPD incurred by task $\tau_i$, in component $G$ due to the other components in the system is then bounded by the size of the multiset intersection of $M_{G,i}^{ucb}$, equation (7.13), and $M_{G,i}^{ecb-C}$, equation (7.16).

$$\gamma_i'^G = \mathrm{BRT} \bullet \left| M_{G,i}^{ucb} \cap M_{G,i}^{ecb-C} \right| \tag{7.17}$$

### UCB-ECB-Multiset-Open

In *open* hierarchical systems the other components may not be known *a priori* as they can be introduced into a system dynamically. Additionally, even in *closed* systems, full information about the other components in the system may not be available until the final stages of system integration. In both of these cases, only the UCB-Only approach can be used as it requires no knowledge of the other components. We therefore present a variation called UCB-ECB-Multiset-Open that improves on UCB-Only while bounding the maximum component CRPD that could be caused by other unknown components. This approach draws on the benefits of the Multiset approaches, by counting the number of intermediate pre-emptions, while also recognising the fact that the cache utilisation of the

other components can often be greater than the size of the cache. As such, the precise number of ECBs does not matter.

For the UCB-ECB-Multiset-Open approach we keep equation (7.13) for calculating the set of UCBs. Furthermore, we form a second multiset $M_{G,i}^{ecb-O}$ that contains $E^G(R_i)$ copies of all cache blocks. This multiset reflects the fact that server $S^G$ can be both suspended and resumed, and the entire contents of the cache evicted at most $E^G(R_i)$ times within the response time of task $\tau_i$.

$$M_{G,i}^{ecb-O} = \bigcup_{E^G(R_i)} (\{1,2,..N\}) \tag{7.18}$$

Where $N$ is the number of cache sets.

The total CRPD incurred by task $\tau_i$, in component $G$ due to the other unknown components in the system is then bounded by the size of the multiset intersection of $M_{G,i}^{ucb}$, equation (7.13), and $M_{G,i}^{ecb-O}$, equation (7.18).

$$\gamma_i'^{G} = \text{BRT} \bullet \left| M_{G,i}^{ucb} \cap M_{G,i}^{ecb-O} \right| \tag{7.19}$$

## 7.3.1 Comparison of Approaches

We have presented a number of approaches that calculate the CRPD due to global context switches, server switching, in a hierarchical system. Figure 7.4 shows a Venn diagram representing the relationships between the different approaches. The larger the area, the more tasksets the approach deems schedulable. The diagram highlights the incomparability between the '-All' and '-Counted' approaches. The diagram also highlights dominance. For example, by construction, UCB-ECB-Multiset-All dominates UCB-ECB-Multiset-Open and UCB-ECB-All, and UCB-All dominates ECB-Only-All.

We now give worked examples illustrating both incomparability and dominance relationships between the different approaches.

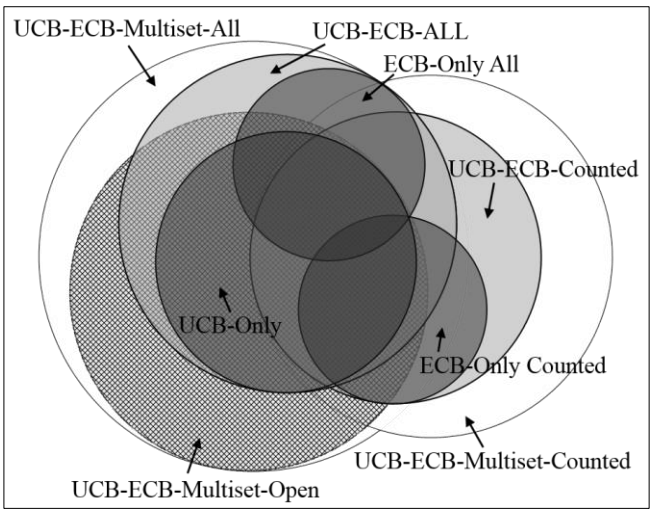



*Figure 7.4 - Venn diagram showing the relationship between the different approaches*

Consider the following example with three components, *G*, *A* and *B*, where component *G* has one task, Let BRT=1, $E^G(R_1)=10$, $X^A\left(S^G,R_1\right)=10$, $X^B\left(S^G,R_1\right)=2$, $ECB^A=\{1,2\}$ and $ECB^B=\{3,4,5,6,7,8,9,10\}$. In this example components *A* and *G* run at the same rate, while component *B* runs at a tenth of the rate of component *G*.

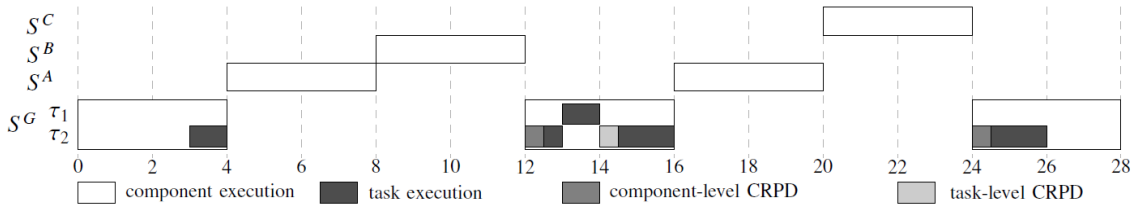
ECB-Only-All considers the ECBs of component *B* effectively assuming that component *B* runs at the same rate as component *G*:

$$\begin{aligned}\gamma_1'^G &= \mathrm{BRT}\times E^G(R_1)\times\left|\mathrm{ECB}^A\cup\mathrm{ECB}^B\right| \\ \gamma_1'^G &= 1\times10\times\left|\{1,2\}\cup\{3,4,5,6,7,8,9,10\}\right| \\ &= 10\times\left|\{1,2,3,4,5,6,7,8,9,10\}\right|=10\times10=100\end{aligned}$$

By comparison ECB-Only-Counted considers components *A* and *B* individually, and accounts for the ECBs of component *B* based on the number of disruptive executions that it may have.

$$\begin{aligned}\gamma_1'^G &= \mathrm{BRT}\times\begin{pmatrix}X^A\left(S^G,R_1\right)\times\left|\mathrm{ECB}^A\right|+\\ X^B\left(S^G,R_1\right)\times\left|\mathrm{ECB}^B\right|\end{pmatrix} \\ \gamma_1'^G &= 1\times\begin{pmatrix}10\times\left|\{1,2\}\right|+\\ 2\times\left|\{3,4,5,6,7,8,9,10\}\right|\end{pmatrix} \\ &= (10\times2)+(2\times8)=36\end{aligned}$$

We now present a more detailed worked example for all approaches where the ECB-Only-All approach outperforms the ECB-Only-Counted approach. This confirms the incomparability of the -All and -Counted approaches.




| | UCB | ECB |
|---|---|---|
| $S^C$ | - | {4,5,6,7,8,9,10} |
| $S^B$ | - | {2,3,4,5} |
| $S^A$ | - | {2,3,4,5,6,7,8} |
| $\tau_1$ | {2} | {2,3} |
| $\tau_2$ | {1,2,3} | {1,2,3,4} |

*Figure 7.5 - Example schedule and UCB/ECB data for four components to demonstrate how the different approaches calculate CRPD*

Figure 7.5 shows an example schedule for four components, *G*, *A*, *B* and *C*, where component *G* has two tasks. Let BRT=1, $E^G(R_1)=1$, $E^G(R_2)=2$, $E_1(R_2)=1$ and $E_2(R_2)=1$, and the number of disruptive executions be:

$X^A\left(S^G,R_1\right)=1$, $X^B\left(S^G,R_1\right)=1$, $X^C\left(S^G,R_1\right)=1$ and $X^A\left(S^G,R_2\right)=2$, $X^B\left(S^G,R_2\right)=2$, $X^C\left(S^G,R_2\right)=2$.

The following examples show how some of the approaches calculate the component CRPD for task $\tau_2$ of component *G*.

ECB-Only-All:

$$\begin{aligned}\gamma_2'^G &= \text{BRT}\times E^G(R_2)\times\left|\text{ECB}^A\cup\text{ECB}^B\cup\text{ECB}^C\right| \\ \gamma_2'^G &= 1\times 2\times\left|\{2,3,4,5,6,7,8\}\cup\{2,3,4,5\}\cup\{4,5,6,7,8,9,10\}\right| \\ &= 2\times\left|\{2,3,4,5,6,7,8,9,10\}\right| = 2\times 9 = 18\end{aligned}$$

ECB-Only-Counted:

$$\gamma_2'^G = \text{BRT}\times\begin{pmatrix} X^A\left(S^G,R_2\right)\times\left|\text{ECB}^A\right|+ \\ X^B\left(S^G,R_2\right)\times\left|\text{ECB}^B\right|+ \\ X^C\left(S^G,R_2\right)\times\left|\text{ECB}^C\right| \end{pmatrix}$$

$$\begin{aligned}\gamma_2'^G &= 1\times\begin{pmatrix} 2\times\left|\{2,3,4,5,6,7,8\}\right|+ \\ 2\times\left|\{2,3,4,5\}\right|+ \\ 2\times\left|\{4,5,6,7,8,9,10\}\right| \end{pmatrix} \\ &= (2\times 7)+(2\times 4)+(2\times 7)=36\end{aligned}$$

UCB-Only:

$$\begin{aligned}\gamma_2'^{G} &= \mathrm{BRT} \times E^{G}(R_2) \times \left|\mathrm{UCB}_1 \cup \mathrm{UCB}_2\right| \\ \gamma_2'^{G} &= 1 \times 2 \times \left(\left|\{2\} \cup \{1,2,3\}\right|\right) \\ &= 2 \times \left|\{1,2,3\}\right| = 6\end{aligned}$$

All of those approaches overestimated the CRPD, although UCB-Only achieves a much tighter bound than the ECB-Only-All and ECB-Only-Counted approaches. The bound can be tightened further by using the more sophisticated approaches, for example, UCB-ECB-Multiset-Counted:

$$M_{G,2}^{ucb} = \left(\bigcup_{E^{G}(R_1)E_1(R_2)} \mathrm{UCB}_1\right) \cup \left(\bigcup_{E^{G}(R_2)E_2(R_2)} \mathrm{UCB}_2\right)$$
$$M_{G,2}^{ucb} = \{2\} \cup \{1,2,3\} \cup \{1,2,3\} = \{1,1,2,2,2,3,3\}$$

$$M_{G,2}^{ecb-C} = \left(\bigcup_{X^{A}(S^{G},R_2)} \mathrm{ECB}^{A}\right) \cup \left(\bigcup_{X^{B}(S^{G},R_2)} \mathrm{ECB}^{B}\right) \cup \left(\bigcup_{X^{C}(S^{G},R_2)} \mathrm{ECB}^{C}\right)$$
$$\begin{aligned}M_{G,2}^{ecb-C} &= \{2,3,4,5,6,7,8\} \cup \{2,3,4,5,6,7,8\} \cup \{2,3,4,5\} \\ &\cup \{2,3,4,5\} \cup \{4,5,6,7,8,9,10\} \cup \{4,5,6,7,8,9,10\} \\ &= \{2,2,2,2,3,3,3,3,4,4,4,4,4,4,5,5,5,5,5,5, \\ &\quad 6,6,6,6,7,7,7,7,,8,8,8,8,9,9,10,10\} \\ \gamma_2'^{G} &= \mathrm{BRT} \times \left|M_{G,2}^{ucb} \cap M_{G,2}^{ecb-C}\right| = 1 \times \left|\{2,2,2,3,3\}\right| = 5\end{aligned}$$

In this specific case, the UCB-ECB-Multiset-All approach calculates the tightest bound:

$$M_{G,2}^{ecb-A} = \bigcup_{E^{G}(R_2)} \left(\mathrm{ECB}^{A} \cup \mathrm{ECB}^{B} \cup \mathrm{ECB}^{C}\right)$$
$$\begin{aligned}M_{G,2}^{ecb-A} &= \bigcup_{2} \left(\{2,3,4,5,6,7,8\} \cup \{2,3,4,5\} \cup \{4,5,6,7,8,9,10\}\right) \\ &= \bigcup_{2} \left(\{2,3,4,5,6,7,8,9,10\}\right) \\ &= \{2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10\} \\ \gamma_2'^{G} &= \mathrm{BRT} \times \left|M_{G,2}^{ucb} \cap M_{G,2}^{ecb-A}\right| = 1 \times \left|\{2,2,3,3\}\right| = 4\end{aligned}$$

Assuming there are 12 cache sets in total[1], the UCB-ECB-Multiset-Open approach gives:

[1] Although we used 12 cache sets in this example, we note that the result obtained is in fact independent of the total number of cache sets.

$$M_{G,2}^{ecb-O} = \bigcup_{E^G(R_2)} \left( \{1,2,3,4,5,6,7,8,9,10,11,12\} \right)$$

$$M_{G,2}^{ecb-O} = \bigcup_{2} \left( \{1,2,3,4,5,6,7,8,9,10,11,12\} \right)$$

$$= \begin{Bmatrix} 1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8, \\ 9,9,10,10,11,11,12,12 \end{Bmatrix}$$

$$\gamma_2'^G = \text{BRT} \times \left| M_{G,2}^{ucb} \cap M_{G,2}^{ecb-O} \right| = 1 \times \left| \{1,1,2,2,3,3\} \right| = 6$$

# 7.4 CRPD Analysis for Hierarchical Systems: Local EDF Scheduler

In this section we present CRPD analysis for hierarchical systems with a local EDF scheduler by adapting the analysis that we presented for a local FP scheduler in Section 7.3.

Overall, the analysis must account for the cost of reloading any UCBs into cache that may be evicted by tasks running in the other components. This is in addition to the cost of reloading any UCBs into cache that may be evicted by tasks in the same component. For calculating the intra-component CRPD, we use the Combined Multiset approach presented in Chapter 4 for EDF scheduling of a single level system. To account for the component level CRPD, we define a new term $\gamma_t'^G$ that represents the CRPD incurred by tasks in component $G$ due to tasks in the other components running while the server, $S^G$, for component $G$ is suspended. Combining equation (4.12) with $isbf^G$, equation (7.1), and $\gamma_t'^G$, we get the following expression for the modified processor demand[1] within an interval of length $t$:

$$h(t) = isbf^G \left( \sum_{j=1}^{n} \left( \max \left\{ 0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor \right\} C_j + \gamma'_{t,j} \right) + \gamma_t'^G \right) \tag{7.20}$$

In order to account for component CRPD we must define an additional term. The set of tasks in component $G$ that can be affected by the server $S^G$ being both suspended and resumed in an interval of length $t$, aff($G$,$t$) is based on the relative deadlines of the tasks. It captures all of the tasks whose relative deadlines are less than or equal to $t$ as they need to be included when

[1] Strictly, $h(t)$ is the maximum time required for the server to provide the processing time demand.

calculating $h(t)$. See Theorem 4.2 in Section 4.1 for a proof for why tasks whose deadlines are larger than $t$ can be excluded. This gives:

$$\mathrm{aff}(G,t) = \{\tau_i \in \Gamma^G \mid t \geq D_i\} \tag{7.21}$$

## ECB-Only

Recall that the ECB-Only approach to calculate component CPRD considers the maximum effect of the other components by assuming that every block evicted by the tasks in the other components has to be reloaded. There are two different ways to calculate this cost.

### ECB-Only-All

The ECB-Only-All approach assumes that every time server $S^G$ is suspended, all of the other servers run and their tasks evict all the cache blocks that they use. We therefore take the union of all ECBs belonging to the other components to get the number of blocks that could be evicted. We then sum them up $E^G(t)$ times, where $E^G(t)$ upper bounds the number of times server $S^G$ could be both suspended and resumed during an interval of length $t$. We can calculate the CRPD impacting tasks in component $G$ due to the other components in the system as:

$$\gamma'^{G}_{t} = \mathrm{BRT} \bullet E^G(t) \bullet \left| \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \mathrm{ECB}^Z \right| \tag{7.22}$$

### ECB-Only-Counted

The ECB-Only-Counted approach considers each component separately by calculating the number of disruptive executions that server $S^Z$ can have on tasks in component $G$ during an interval of length $t$, $X^Z(S^G,t)$. We can then calculate an alternative bound for the CRPD incurred by tasks in component $G$ due to the other components in the system as:

$$\gamma'^{G}_{t} = \mathrm{BRT} \bullet \sum_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left( X^Z(S^G,t) \bullet \left| \mathrm{ECB}^Z \right| \right) \tag{7.23}$$

Note that the ECB-Only-All and ECB-Only-Counted approaches are incomparable.

## UCB-Only

The UCB-Only approach focuses on the tasks in component *G*, hence calculating which UCBs could be evicted if the entire cache was flushed by the other components in the system. With a local EDF scheduler, we must consider all tasks in component *G* that are both released and have their deadlines within an interval of length *t*. We therefore take the union of the UCBs of all tasks in component *G* that have a deadline less than *t*, $\tau_k \in \text{aff}(G,t)$, to give:

$$\bigcup_{\forall k \in \text{aff}\,(G,t)} \text{UCB}_k \tag{7.24}$$

We then multiply the number of UCBs, equation (7.24), by the number of times that server $S^G$ can be both suspended and resumed during an interval of length *t*.

$$\gamma_t'^G = \text{BRT} \bullet E^G(t) \bullet \left| \bigcup_{\forall k \in \text{aff}\,(G,t)} \text{UCB}_k \right| \tag{7.25}$$

This approach is incomparable with the ECB-Only-All and ECB-Only-Counted approaches.

## UCB-ECB

We now re-formulate the UCB-ECB approaches for a local EDF scheduler.

### UCB-ECB-All

We build upon the ECB-Only-All and UCB-Only methods. We start with the union of the UCBs of all tasks in component *G* that could be affected within an interval of length *t*, (7.24). We then take the union of all ECBs belonging to the other components to give the number of blocks that could potentially be evicted. We then calculate the intersection between the two unions to give an upper bound on the number of UCBs evicted by the ECBs of the tasks in the other components:

$$\left| \left( \bigcup_{\forall k \in \text{aff}\,(G,t)} \text{UCB}_k \right) \cap \left( \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \text{ECB}^Z \right) \right| \tag{7.26}$$

This upper bound is then multiplied by the number of times that the server $S^G$ could be both suspended and resumed during an interval of length *t* to give:

$$\gamma_t'^{G} = \text{BRT} \bullet E^G(t) \bullet \left| \left( \bigcup_{\forall k \in \text{aff}(G,t)} \text{UCB}_k \right) \cap \left( \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \text{ECB}^Z \right) \right| \tag{7.27}$$

By construction, the UCB-ECB-All approach dominates the ECB-Only-All and UCB-Only approaches.

### UCB-ECB-Counted

With the UCB-ECB-Counted approach we start by calculating an upper bound on the number of blocks that could be used by tasks in component *G* which are both released and have their deadlines within an interval of length *t*. We then take the intersection of these UCBs with the set of ECBs of component Z to give the number of blocks that could be evicted by component *Z*:

$$\left| \left( \bigcup_{\forall k \in \text{aff}(G,t)} \text{UCB}_k \right) \cap \text{ECB}^Z \right| \tag{7.28}$$

We then multiply this number of blocks by the number of disruptive executions that server $S^Z$ can have during an interval of length *t* and sum this up for all components to give:

$$\gamma_t'^{G} = \text{BRT} \bullet \sum_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left( X^Z(S^G, t) \bullet \left| \left( \bigcup_{\forall k \in \text{aff}(G,t)} \text{UCB}_k \right) \cap \text{ECB}^Z \right| \right) \tag{7.29}$$

By construction, the UCB-ECB-Counted approach dominates the ECB-Only-Counted approach, but is incomparable with the UCB-Only approach.

## UCB-ECB-Multiset

The UCB-ECB approaches are pessimistic in that they assume that each component can, directly or indirectly, evict UCBs of each task $\tau_k \in \text{aff}(G,t)$ in component *G* up to $E^G(t)$ times during an interval of length *t*. The UCB-ECB-Multiset approaches, described below, remove this source of pessimism by upper bounding the number of times server $S^G$ can be both suspended and resumed while each task $\tau_k \in \Gamma^G$ is running during an interval of length *t*.

We first calculate an upper bound on the UCBs that if evicted by tasks in the other components may need to be reloaded. We do this by forming a multiset that contains the UCBs of task $\tau_k$ repeated $E^G(D_k)E_k(t)$ times for each task in $\tau_k \in \Gamma^G$. This multiset reflects the fact that server $S^G$ can be suspended and

resumed at most $E^G(D_k)$ times during a single schedulable job of task $\tau_k$ and there can be at most $E_k(t)$ jobs of task $\tau_k$ that have their release times and absolute deadlines within the interval of length $t$.

$$M_{G,t}^{ucb} = \bigcup_{\forall k \in \Gamma^G} \left( \bigcup_{E^G(D_k) E_k(t)} \text{UCB}_k \right) \tag{7.30}$$

Note that we do not restrict the set of tasks $\tau_k \in \Gamma^G$ using $\tau_k \in \text{aff}(G,t)$, as $E_k(t)$ will be 0 for any task which has a deadline shorter than $t$.

The second step is to determine which ECBs of the tasks in the other components could evict the UCBs in equation (7.30), for which we present three different approaches.

### UCB-ECB-Multiset-All

The first option is to assume that every time server $S^G$ is suspended, all of the other servers run and their tasks evict all the cache blocks that they use. We therefore take the union of all ECBs belonging to the other components to get the set of blocks that could be evicted. We form a second multiset $M_{G,t}^{ecb-A}$ that contains $E^G(t)$ copies of the ECBs of all of the other components in the system. This multiset reflects the fact that the other servers' tasks can evict blocks, that need to be reloaded, at most $E^G(t)$ times within an interval of length $t$.

$$M_{G,t}^{ecb-A} = \bigcup_{E^G(t)} \left( \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \text{ECB}^Z \right) \tag{7.31}$$

The total CRPD incurred by tasks in component $G$ due to the other components in the system is then given by the size of the multiset intersection of $M_{G,t}^{ucb}$, equation (7.30), and $M_{G,t}^{ecb-A}$, equation (7.31):

$$\gamma_t'^G = \text{BRT} \bullet \left| M_{G,t}^{ucb} \cap M_{G,t}^{ecb-A} \right| \tag{7.32}$$

### UCB-ECB-Multiset-Counted

The second option is to consider each component separately by calculating the number of disruptive executions, $X^Z(S^G,t)$, that server $S^Z$ can have on tasks in component $G$ during $t$. We form a second multiset $M_{G,t}^{ecb-C}$ that contains $X^Z(S^G,t)$ copies of $\text{ECB}^Z$ for each of the other components $Z$ in the system. This

multiset reflects the fact that the tasks of each component $Z$ can evict blocks at most $X^Z\left(S^G,t\right)$ times within an interval of length $t$.

$$M_{G,t}^{ecb-C} = \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left( \bigcup_{X^Z\left(S^G,t\right)} \mathrm{ECB}^Z \right) \tag{7.33}$$

The total CRPD incurred by tasks in component $G$ which are released and have their deadlines in an interval of length $t$, due to the other components in the system is then given by the size of the multiset intersection of $M_{G,t}^{ucb}$, equation (7.30), and $M_{G,t}^{ecb-C}$, equation (7.33)

$$\gamma_t'^G = \mathrm{BRT} \bullet \left| M_{G,t}^{ucb} \cap M_{G,t}^{ecb-C} \right| \tag{7.34}$$

**UCB-ECB-Multiset-Open**

With the UCB-ECB-Multiset-Open approach we form a second multiset $M_{G,t}^{ecb-O}$ that contains $E^G(t)$ copies of all cache blocks. This multiset reflects the fact that server $S^G$ can be both suspended and then resumed, after the entire contents of the cache have been evicted at most $E^G(t)$ times within an interval of length $t$.

$$M_{G,t}^{ecb-O} = \bigcup_{E^G(t)} \left( \{1,2,..N\} \right) \tag{7.35}$$

Where $N$ is the number of cache sets.

The total CRPD incurred by tasks in component $G$ due to the other unknown components in the system is then given by the size of the multiset intersection of $M_{G,t}^{ucb}$, equation (7.30), and $M_{G,t}^{ecb-O}$, equation (7.35).

$$\gamma_t'^G = \mathrm{BRT} \bullet \left| M_{G,t}^{ucb} \cap M_{G,t}^{ecb-O} \right| \tag{7.36}$$

## 7.4.1 Effect on Task Utilisation and h(t) Calculation

As the component level CRPD analysis effectively inflates the execution time of tasks by the CRPD that can be incurred in an interval of length $t$, the upper bound $L$ used for calculating the processor demand $h(t)$ must be adjusted. This is an extension to the adjustment that must be made for task level CRPD as described in Section 4.2.1. This is achieved by calculating an upper bound on the utilisation due to CRPD that is valid for all intervals of length greater than some value $L_c$. This CRPD utilisation value is then used to inflate the taskset

utilisation, and thus compute an upper bound $L_d$ on the maximum length of the busy period. This upper bound is valid provided that it is greater than $L_c$, otherwise the actual maximum length of the busy period may lie somewhere in the interval $[L_d, L_c]$, hence we can use $\max(L_c, L_d)$ as a bound.

The first step is to assign $t = L_c = 100\ T_{max}$ which limits the overestimation of both the task level CRPD utilisation $U^{\gamma} = \gamma'_t / t$ and the component level CRPD utilisation $U^{\gamma G} = \gamma'^{G}_t / t$ to at most 1%. We determine $U^{\gamma G}$ by calculating $\gamma'^{G}_t$ however when calculating the multiset of the UCBs that could be affected $M^{ucb}_{G,t}$, equation (7.30), $E^{max}_x(t)$ is substituted for $E_x(t)$ to ensure that the computed value of $U^{\gamma G}$ is a valid upper bound for all intervals of length $t \geq L_c$.

$$E^{max}_x(t) = \max\left(0, 1 + \left\lceil \frac{t - D_x}{T_x} \right\rceil\right) \tag{7.37}$$

We use a similar technique of substituting $E^{max}_x(t)$ for $E_x(t)$ in the calculation of the task level CRPD, as described in Section 4.2.1, to give $U^{\gamma}$.

If $U + U^{\gamma} + U^{\gamma G} \geq 1$, then the taskset is deemed unschedulable, otherwise an upper bound on the length of the busy period can be computed via a modified version of equation (2.4):

$$w^{\alpha+1} \leq \sum_{\forall j} \left( \frac{w^{\alpha}}{T_j} + 1 \right) C_j + w^{\alpha} U^{\gamma} \tag{7.38}$$

rearranged to give:

$$w \leq \frac{1}{\left(1 - \left(U + U^{\gamma} + U^{\gamma G}\right)\right)} \sum_{\forall j} U_j T_j \tag{7.39}$$

Then, substituting in $T_{max}$ for each value of $T_j$ the upper bound is given by:

$$L_d = \frac{U \bullet T_{max}}{\left(1 - \left(U + U^{\gamma} + U^{\gamma G}\right)\right)} \tag{7.40}$$

Finally, $L = \max(L_c, L_d)$ can then be used as the maximum value of $t$ to check in the EDF schedulability test.

## 7.4.2 Comparison of Approaches

In this section we have presented a number of approaches for calculating component CRPD in a hierarchical system with a local EDF scheduler. These

approaches all have the same dominance and incomparability relationships as the approaches presented in Section 7.3 for a local FP scheduler. We therefore refer the reader to Section 7.3.1 for an explanation of the relationships between the approaches. However, the relative performance between the approaches differ from the FP variants as shown in the next section.

## 7.5 Case Study

In this section we compare the different approaches for calculating CRPD in hierarchical scheduling using tasksets based on a case study. The case study uses PapaBench[1] which is a real-time embedded benchmark based on the software of a GNU-license UAV, called Paparazzi. WCETs, UCBs, and ECBs were calculated for the set of tasks using aiT[2] based on an ARM processor clocked at 100MHz with a 2KB direct-mapped instruction cache. The cache was again setup with a line size of 8 Bytes, giving 256 cache sets, 4 Byte instructions, and a BRT of 8µs. WCETs, periods, UCBs, and ECBs for each task based on the target system are provided in Table 7.1. As in Chapter 6, we made the following assumptions in our evaluation to handle the interrupt tasks:

- Interrupts have a higher priority than the servers and normal tasks.
- Interrupts cannot pre-empt each other.
- Interrupts can occur at any time.
- All interrupts have the same deadline which must be greater than or equal to the sum of their execution times in order for them to be schedulable.
- The cache is disabled whenever an interrupt is executing and enabled again after it completes.

Based on these assumptions, we integrated interrupts into the model by replacing the server capacity $Q^G$ in equation (7.1) by $Q^G$ - $I^G$, where $I^G$ is the maximum execution time of all interrupts in an interval of length $Q^G$. This effectively assumes that the worst case arrival of interrupts could occur in any component and steals time from its budget.

[1] http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97
[2] http://www.absint.com/ait/

| Task | | UCBs | ECBs | WCET | Period |
|---|---|---|---|---|---|
| FLY-BY-WIRE | | | | | |
| I1 | interrupt_radio | 2 | 10 | 0.210 ms | 25 ms |
| I2 | interrupt_servo | 1 | 6 | 0.167 ms | 50 ms |
| I3 | interrupt_spi | 2 | 10 | 0.256 ms | 25 ms |
| T1 | check_failsafe | 10 | 132 | 1.240 ms | 50 ms |
| T2 | check_mega128_values | 10 | 130 | 5.039 ms | 50 ms |
| T3 | send_data_to_autopilot | 10 | 114 | 2.283 ms | 25 ms |
| T4 | servo_transmit | 2 | 10 | 2.059 ms | 50 ms |
| T5 | test_ppm | 30 | 255 | 12.579 ms | 25 ms |
| AUTOPILOT | | | | | |
| I4 | interrupt_modem | 2 | 10 | 0.303 ms | 100 ms |
| I5 | interrupt_spi_1 | 1 | 10 | 0.251 ms | 50 ms |
| I6 | interrupt_spi_2 | 1 | 4 | 0.151 ms | 50 ms |
| I7 | interrupt_gps | 3 | 26 | 0.283 ms | 250 ms |
| T5 | altitude_control | 20 | 66 | 1.478 ms | 250 ms |
| T6 | climb_control | 1 | 210 | 5.429 ms | 250 ms |
| T7 | link_fbw_send | 1 | 10 | 0.233 ms | 50 ms |
| T8 | navigation | 10 | 256 | 4.432 ms | 250 ms |
| T9 | radio_control | 0 | 256 | 15.681 ms | 25 ms |
| T10 | receive_gps_data | 22 | 194 | 5.987 ms | 250 ms |
| T11 | reporting | 2 | 256 | 12.222 ms | 100 ms |
| T12 | stabilization | 11 | 194 | 5.681 ms | 50 ms |

*Table 7.1 - Execution times, periods and number of UCBs and ECBs for the tasks from PapaBench*

We assigned a deadline of 2ms to all of the interrupt tasks, and implicit deadlines so that $D_i = T_i$, to the normal tasks. We then calculated the total utilisation for the system and then scaled $T_i$ and $D_i$ up for all tasks in order to reduce the total utilisation to the target utilisation for the system. We used the number of UCBs and ECBs obtained via analysis, placing the UCBs in a group at a random location in each task. We then generated 1000 systems each containing a different allocation of tasks to each component, using the following technique. We split the normal tasks at random into 3 components with four tasks in two components and five in the other. In the case of local FP scheduling, we assigned task priorities according to deadline monotonic priority assignment. Next we set the period of each component's server to 12.5ms, which is half the minimum task period. Finally, we organised tasks in each component in memory in a sequential order based on their priority for FP, or their unique task index for EDF. Due to task index assignments, this gave the same task layout. We then ordered components in memory sequentially based on their index.

### 7.5.1 Success Ratio

For each system the total task utilisation across all tasks not including pre-emption cost was varied from 0.025 to 1 in steps of 0.025. For each utilisation value we initialised each servers' capacity to the minimum possible value, the utilisation of all of its tasks. We then performed a binary search between this minimum and the maximum, 1 minus the minimum utilisation of all of the other components, until we found the server capacity required to make the component schedulable. As the servers all had equal periods, provided all components were schedulable and the total capacity required by all servers was ≤ 100%, then the system was deemed schedulable at that specific utilisation level. In addition to evaluating each of the presented approaches, we also calculated schedulability based on no component pre-emption costs, but still including task level CRPD. For every approach the intra-component CRPD, between tasks in the same component, was calculated using either the Combined Multiset approach for FP [7], described in Section 3.2, or the Combined Multiset approach, introduced in Chapter 4 for EDF.

The results for the case study for a local FP scheduler and local EDF scheduler are shown in Figure 7.6. Although we generated 1000 systems, they were all very similar as they are made up of the same set of tasks. The first point to note is that the FP approaches deem a higher number of tasksets schedulable than the EDF ones, despite EDF having a higher number of schedulable tasksets for the No-Component-Pre-emption-Cost case. In section 7.6.4, we explore the source of pessimism in the EDF analysis. Focusing on the different approaches, ECB-Only-Counted and ECB-Only-All perform the worst as they only consider the other components in the system. In the case of a local EDF scheduler, the ECB-Only-Counted approach is unable to deem any tasksets schedulable except at the lowest utilisation level. Next was UCB-ECB-Counted which though it considers all components, accounts for the other components pessimistically in this case study, since all servers have the same period. The remainder of the approaches all had very similar performance.

We note that No-Component-Pre-emption-Cost reveals that the pre-emption costs are very small for the PapaBench tasks. This is due to a number of factors including the nearly harmonic periods, small range of task periods, and relatively low number of ECBs for many tasks.

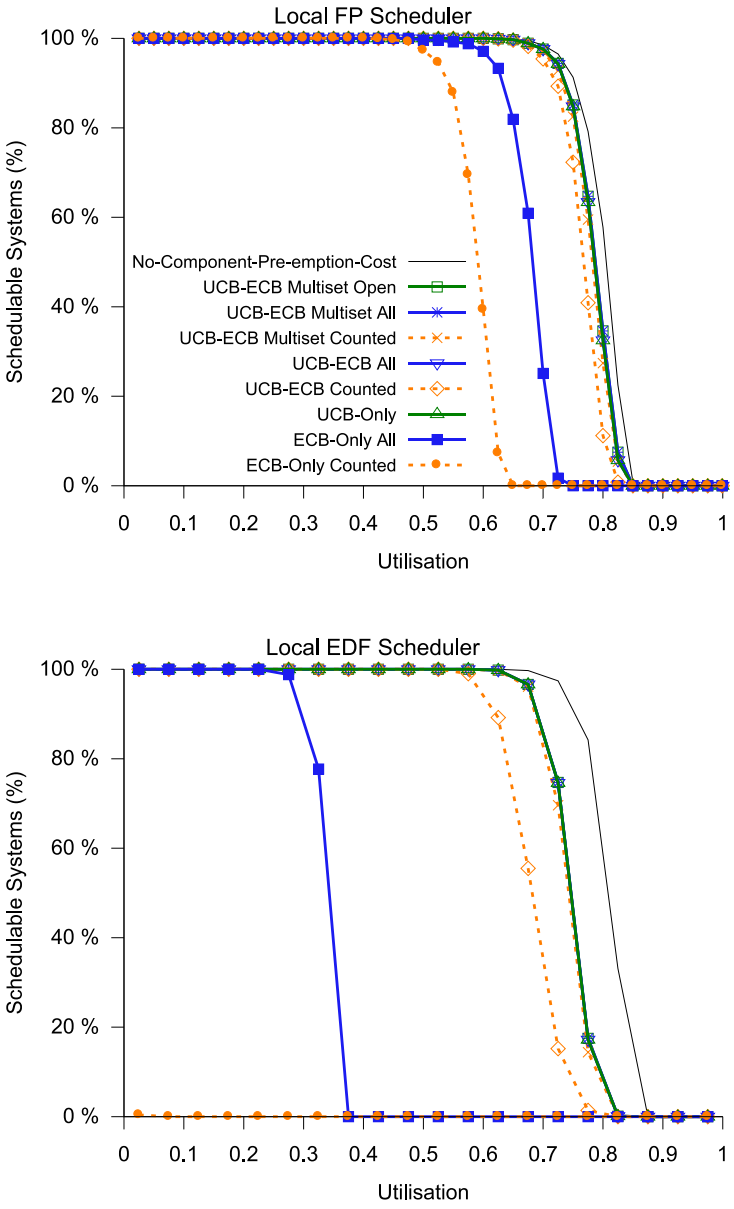



*Figure 7.6 - Percentage of schedulable tasksets at each utilisation level for the case study tasksets*

## 7.6 Evaluation

In this section we compare the different approaches for calculating CRPD in hierarchical scheduling using synthetically generated tasksets. This allows us to explore a wider range of parameters and therefore give some insight into how the different approaches perform in a variety of cases.

To generate the components and tasksets we generated $n$, default of 24, tasks using the UUnifast algorithm [32] to calculate the utilisation, $U_i$, of each task so that the utilisations added up to the desired utilisation level. Periods $T_i$, were generated at random between 10ms and 1000ms according to a log-uniform

distribution. $C_i$ was then calculated via $C_i = U_i T_i$. We generated two sets of tasksets, one with implicit deadlines, so that $D_i = T_i$, and one with constrained deadlines. We used $D_i = y + x(T_i - y))$ to generate the constrained deadlines, where $x$ is a random number between 0 and 1, and $y = \max(T_i/2, 2\ C_i)$. This generates constrained deadlines that are no less than half the period of the tasks. All results presented are for tasks with implicit deadlines. In general the results for constrained deadlines were similar with a lower number of systems deemed schedulable. The exception to this is that under a local EDF scheduler, the UCB-ECB-Multiset approaches showed an increase in schedulability when deadlines were reduced by a small amount. This behaviour is investigated and explained in Section 7.6.4.

We used the UUnifast algorithm to generate the number of ECBs for each task so that the ECBs added up to the desired cache utilisation, default of 10. The number of UCBs was chosen at random between 0% and 30% of the number of ECBs on a per task basis, and the UCBs were placed in a single group at a random location in each task.

We then split the tasks at random into 3 components with equal numbers of tasks in each. In the case of a local FP scheduler, we assigned task priorities according to Deadline Monotonic priority assignment. Next we set the period of each component's server to 5ms, which was half the minimum possible task period. Finally we organised tasks in each component in memory in a sequential order based on their priority for FP, or their unique task index for EDF, which gave the same task layout, and then ordered components in memory sequentially based on their index. We generated 1000 systems using this technique.

In our evaluations we used the same local scheduler in each component, so that all components were scheduled locally using either FP or EDF. However, we note that the analysis is not dependent on the scheduling policies of the other components and hence can be applied to a system where some components are scheduled locally using FP and others using EDF.

### 7.6.1 Success Ratio

We determined the schedulability of the synthetic tasksets using the approach described in the first paragraph of Section 7.5.1.

### 7.6.2 Baseline Evaluation

We investigated the effect of key cache and taskset configurations on the analysis by varying the following key parameters:

- Number of components (default of 3)
- Server period (default of 5ms)
- Cache Utilisation (default of 10)
- Total number of tasks (default of 24)
- Range of task periods (default of [10, 1000]ms)

The results for the baseline evaluation under implicit deadline tasksets are shown in Figure 7.7. The results again show that the analysis for determining inter-component CRPD for a local FP scheduler deems a higher number of systems schedulable than the analysis for a local EDF scheduler. In the case of a local EDF scheduler, both ECB-Only approaches deemed no tasksets schedulable. In the case of a local FP scheduler ECB-Only-Counted is least effective as it only considers the other components and does so individually, followed by ECB-Only-All. UCB-ECB-Counted deemed a higher number of tasksets schedulable, although it deemed significantly fewer for a local EDF scheduler than with a local FP scheduler. Under EDF, UCB-ECB-Multiset-Counted was next, followed by all other approaches. Under FP, UCB-ECB-Multiset-Counted performed similarly to UCB-Only and UCB-ECB-All, crossing over at a utilisation of 0.725 highlighting their incomparability. Although UCB-ECB-All dominates UCB-Only, it can only improve over UCB-Only when the cache utilisation of the other components is sufficiently low that they cannot evict all cache blocks. The UCB-ECB-Multiset-All and UCB-ECB-Multiset-Open approaches performed the best for both types of local scheduler.

Despite only considering the properties of the component under analysis, the UCB-ECB-Multiset-Open approach proved highly effective. The reason for this is that once the size of the other components that can run while a given component is suspended is equal to or greater than the size of the cache then UCB-ECB-Multiset-All and UCB-ECB-Multiset-Open become equivalent.

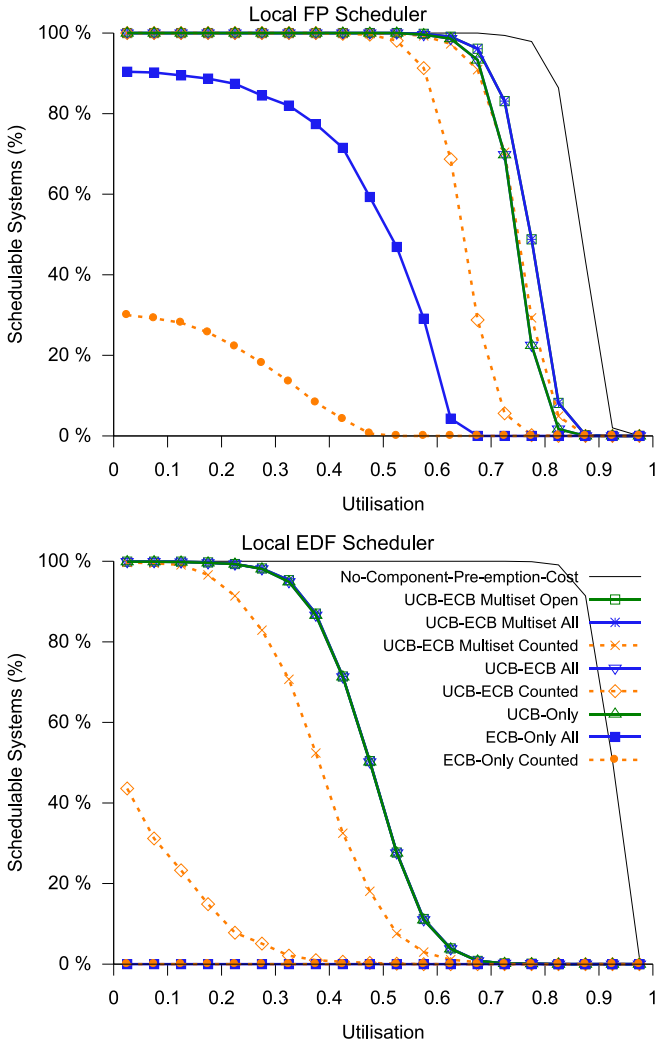



*Figure 7.7 - Percentage of schedulable tasksets at each utilisation level for the synthetic tasksets*

Consider the UCB-ECB-Multiset approaches under a local EDF scheduler. Examining equation (7.30) , we note that $E^G(D_k)E_k(t)$ is based on the deadline of a task. Therefore, the analysis under implicit deadlines effectively assumes the UCBs of all tasks in component G could be in use each time the server for component G is suspended. Whereas, under a local FP scheduler the analysis is able to bound how many times the server for component G is suspended and resumed based on the computed response time of each task which for many tasks is much less than its deadline, and period. Figure 7.8 shows a subset of the results presented in Figure 7.7. When component CRPD is not considered, EDF outperforms FP. However, once component CRPD is taken into account, the analysis for FP significantly outperforms the analysis for EDF.

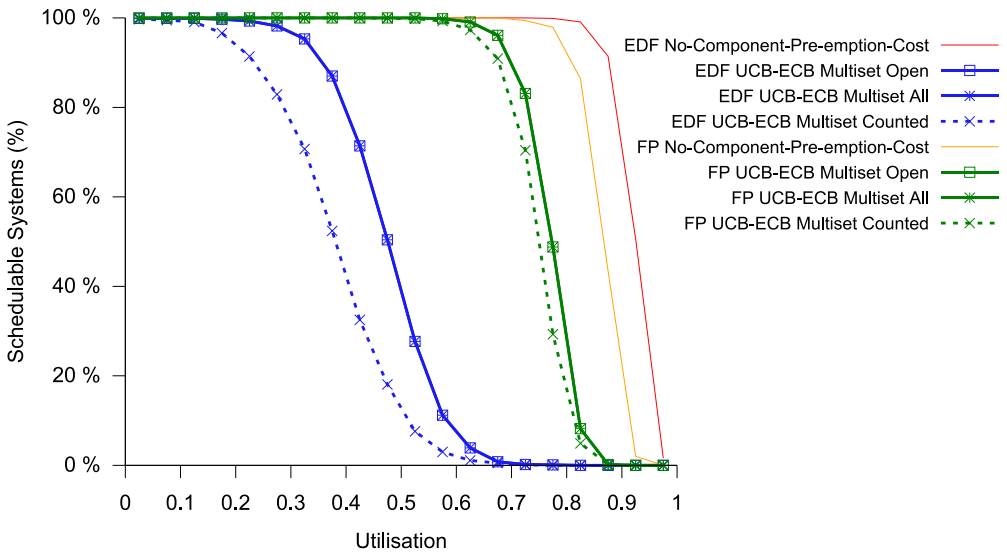



*Figure 7.8 - Percentage of schedulable tasksets at each utilisation level for the synthetic tasksets directly comparing the analysis for local FP and EDF schedulers*

### 7.6.3 Detailed Evaluation

Evaluating all combinations of different task parameters is not possible. Therefore, the majority of our evaluations focused on varying one parameter at a time. To present these results, weighted schedulability measures [21] are used, which are described in Section 4.5.2.

We used 100 systems and varied the utilisation level from 0.025 to 1.0 in steps of 0.025 for the detailed evaluation.

#### Number of Components

To investigate the effects of splitting the overall set of tasks into components, we fixed the total number of tasks in the system at 24, and then varied the number of components from 1, with 24 tasks in one component, to 24, with 1 task per component, see Figure 7.9. Components were allocated an equal number of tasks where possible, otherwise tasks were allocated to each component in turn until all tasks where allocated. We note that with one component, the UCB-Only and UCB-ECB-Multiset-Open approaches calculate a non-zero inter-component CRPD. This is because they assume that every time a component is suspended its UCBs are evicted, even though there is only one component running in the system. With two components the ECB-Only-All and ECB-Only-Counted approaches are equal. Above two components the ECB-Only-All, ECB-Only-Counted and UCB-ECB-Counted approaches get rapidly worse as they over-count blocks. Under a local FP scheduler, all other

approaches improve as the number of components is increased above 2 up to 8 components.

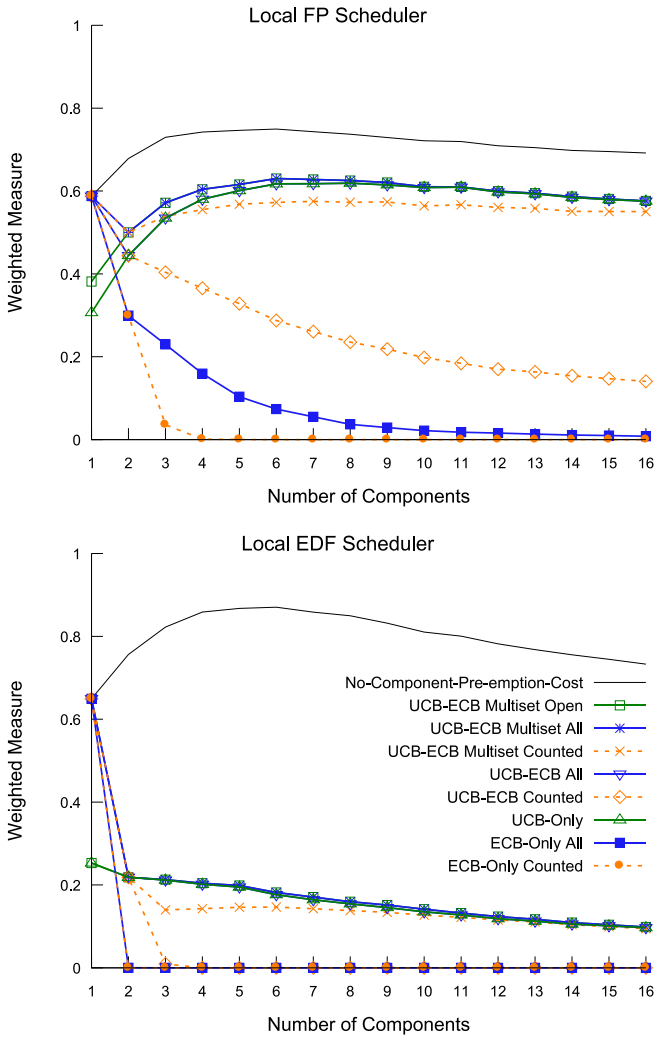



*Figure 7.9 - Weighted measure for varying the number of components from 1 to 16, while keeping the number of tasks in the system fixed*

Under a local EDF scheduler, all approaches that consider inter-component CRPD show a decrease in schedulability as the number of components increases above 2. The No-Component-Pre-emption-Cost case shows an increase in schedulability up to approximately 6-7 components before decreasing. This is because as the number of components increases, the amount of intra-component CRPD from tasks in the same component decreases. This is then balanced against an increased delay in capacity from the components' servers. As the number of components is increased, and therefore the number of servers, $Q^G$ is reduced leading to an increase in $P^G - Q^G$ which increases the maximum time between a server supplying capacity to its component. We also note that at two components, UCB-Only, UCB-ECB-All and UCB-ECB-Counted perform the

same; as do the Multiset approaches. This is because the '-All' and '-Counted' variations are equivalent when there is only one other component.

### System Size

We investigated the effects of introducing components into a system by varying the system size from 1 to 10, see Figure 7.10, where each increase introduces a new component which brings along with it 5 tasks taking up approximately twice the size of the cache.

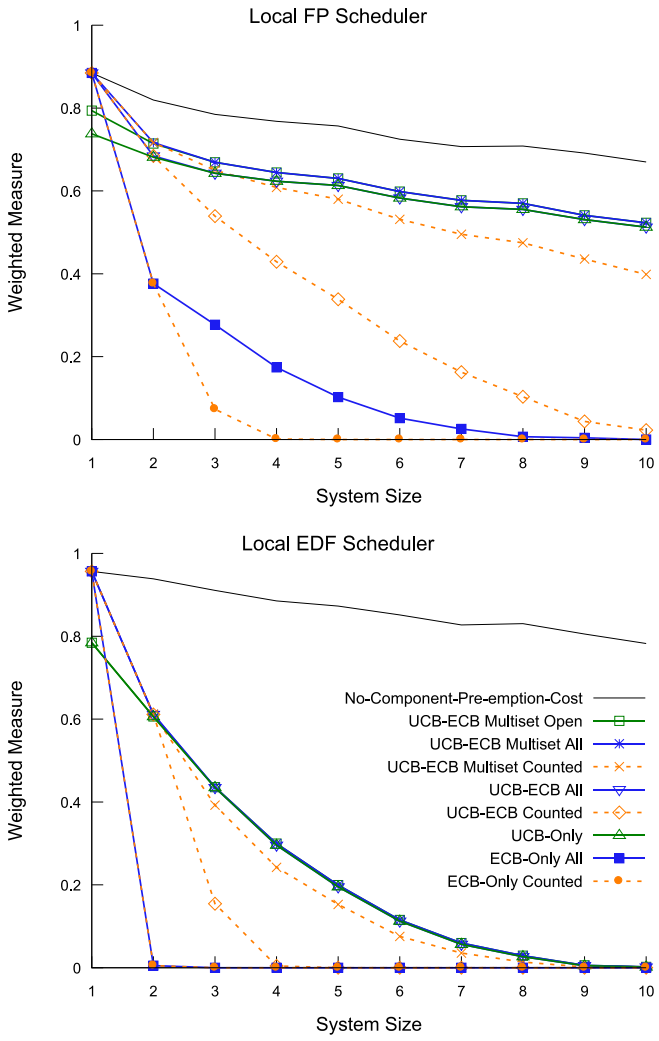



*Figure 7.10 - Weighted measure for varying the system size from 1 to 10 where an increase of 1 in the system size relates to introducing another component that brings along with it another 5 tasks and an increase in the cache utilisation of 2*

When there is one component, all approaches except for UCB-Only and UCB-ECB-Multiset-Open give the same result as No-Component-Pre-emption-Cost. As expected, as more components are introduced into the system, system schedulability decreases for all approaches including No-Component-Pre-emption-Cost. This is because each new component includes additional intra-

component CRPD in addition to the inter-component CRPD that it causes when introduced. Furthermore, each new component that is introduced into the system effectively increases the maximum delay before search server supplies capacity to its components. Under a local FP scheduler, the ECB-Only-All approach outperforms UCB-ECB-Counted above a system size of 2, UCB-Only and UCB-ECB-All outperform UCB-ECB-Multiset-Counted above a system size of 3, highlighting their incomparability. Again we note that the '-All' and '-Counted' variations are the same when there are only two components in the system.

### Server Period

The server period is a critical parameter when composing a hierarchical system. The results for varying the server period from 1ms to 20ms, with a fixed range of task periods from 10 to 1000ms are shown in Figure 7.11. When the component pre-emption costs are ignored, having a small server period ensures that short deadline tasks meet their time constraints. However, switching between components clearly has a cost associated with it making it desirable to switch as infrequently as possible. As the server period increases, schedulability increases due to a smaller number of server context switches, and hence inter-component CRPD, up until approximately 7ms under FP, and 7-8ms under EDF, for the best performance. At this point although the inter-component CRPD continues to decrease, short deadline tasks start to miss their deadlines due to the delay in server capacity being supplied unless server capacities are greatly inflated, and hence the overall schedulability of the system decreases. We note that in the case of EDF, the optimum server period is between 7-8ms for most approaches and 9ms for the UCB-ECB-Counted approach. This increase in optimum server period over FP is due to the increased calculated inter-component CRPD under a local EDF scheduler.

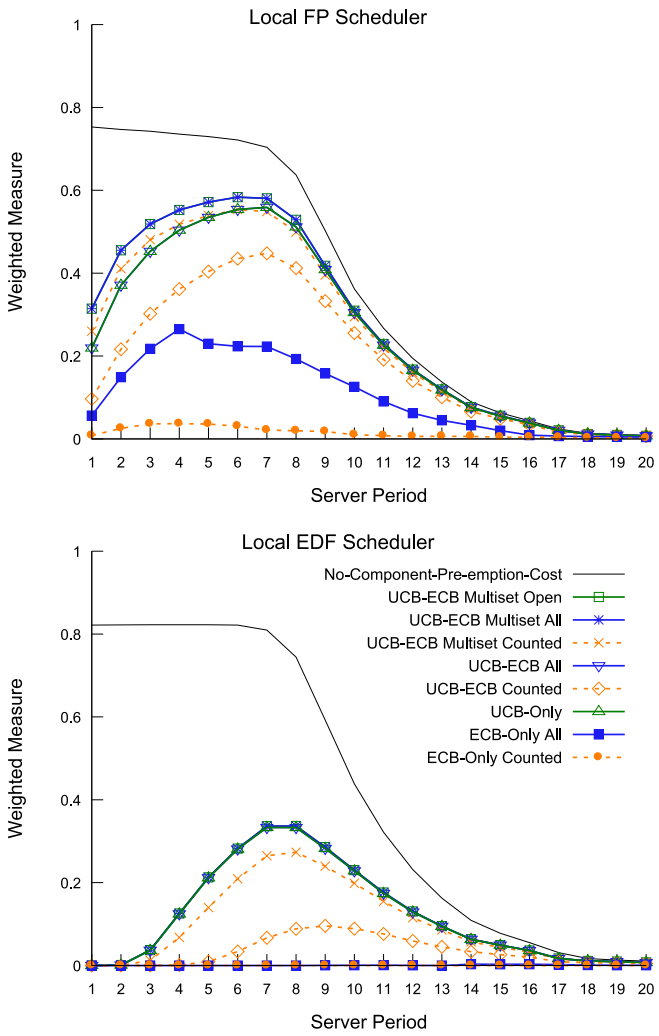



*Figure 7.11 - Weighted measure for varying the server period from 1ms to 20ms (fixed task period range of 10ms to 1000ms)*

### Cache Utilisation

As the cache utilisation increases the likelihood of the other components evicting UCBs belonging to the tasks in the suspended component increases. The results for varying the cache utilisation from 0 to 20 are shown in Figure 7.12. In general, all approaches show a decrease in schedulability as the cache utilisation increases. Up to a cache utilisation of around 2, the UCB-Only and UCB-ECB-Multiset-Open approaches do not perform as well as the more sophisticated approaches, as the other components do not evict all cache blocks when they run. We also observe that up to a cache utilisation of 1 under a local FP scheduler, the ECB-Only-Counted, and the ECB-Only-All approaches perform identically as no ECBs are duplicated.

We note that the weighted measure stays relatively constant for No-Component-Pre-emption-Cost up to a cache utilisation of approximately 2.5.

This is because the average cache utilisation of each component is still less than 1, which leads to relatively small intra-component CRPD between tasks.

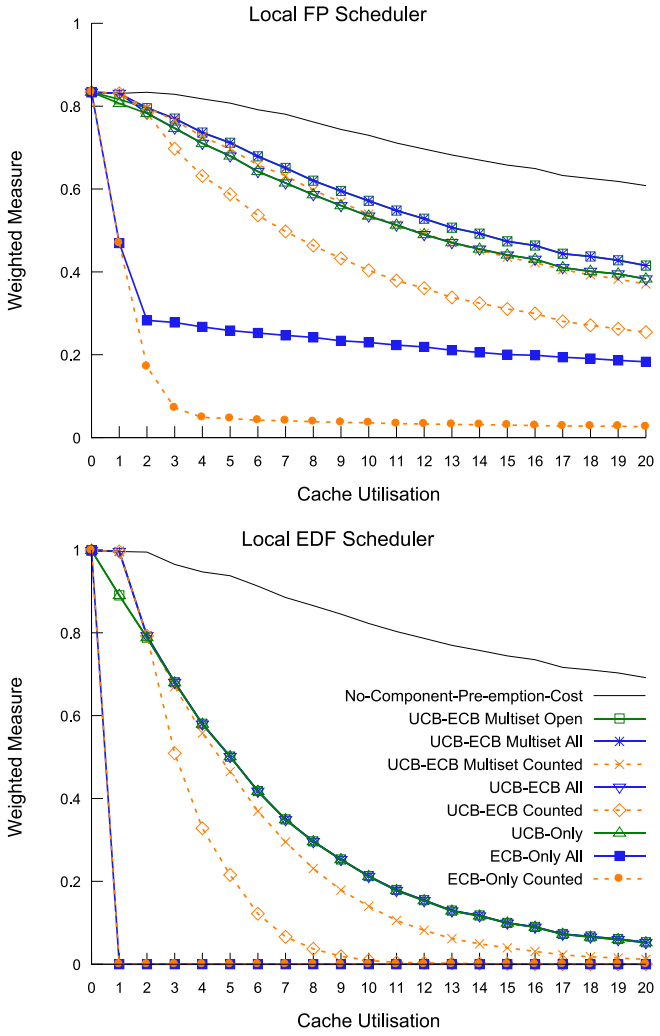



*Figure 7.12 - Weighted measure for varying the cache utilisation from 0 to 20*

### Number of Tasks

We also investigated the effect of varying the number of tasks, while keeping the number of components fixed. As we introduced more tasks, we scaled the cache utilisation in order to keep a constant ratio of tasks to cache utilisation. The results for varying the number of tasks from 3 to 48 are shown in Figure 7.13. As expected, increasing the number of tasks leads to a decrease in schedulability across all approaches that consider inter-component CRPD. However, under a local EDF scheduler, the No-Component-Pre-emption-Cost case actually shows an increase peaking at 12 tasks before decreasing due to the intra-component CRPD. Consider that when there are 3 tasks, there is only one task per component, so there is effectively no local scheduling. Therefore schedulability is based solely on the global scheduling algorithm, which is why

the results for No-Component-Pre-emption-Cost are the same for FP and EDF with 3 tasks. As more tasks are introduced the execution time of individual tasks is reduced, making it less likely that a task will miss a deadline due to its components' server not running. This increases schedulability until the effect of the intra-component CRPD outweighs it.

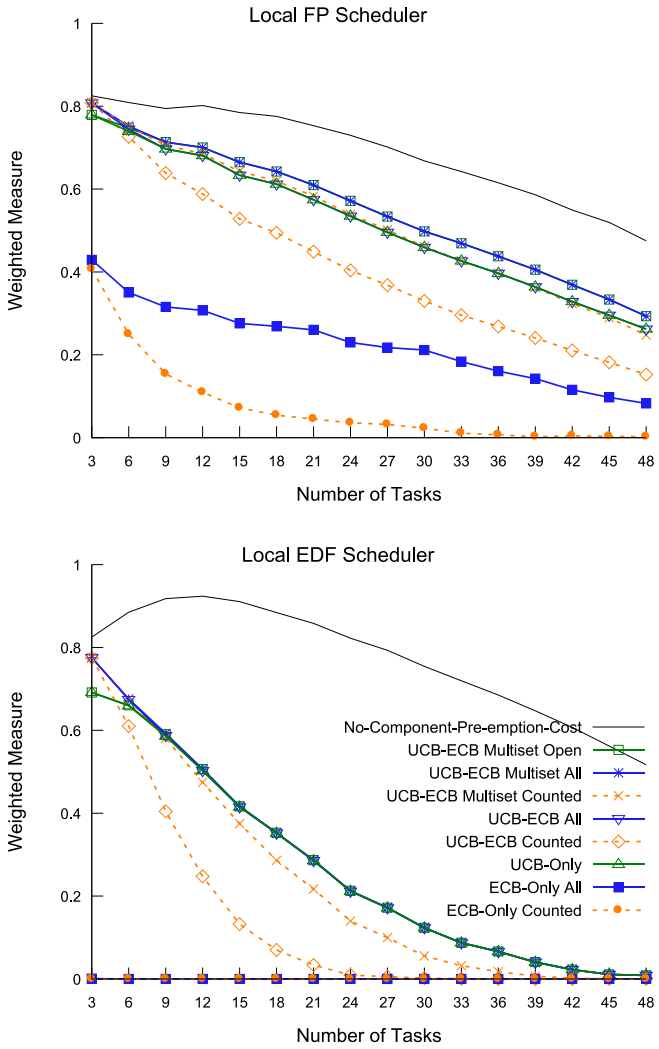



*Figure 7.13 - Weighted measure for varying the total number of tasks from 3 to 48 (1 to 16 tasks per component)*

### Task Period Range

We varied the range of task periods from [1, 100]ms to [20, 2000]ms, while fixing the server period at 5ms. The results are shown in Figure 7.14, as expected, the results show an increase in schedulability across all approaches as the task period range is increased.

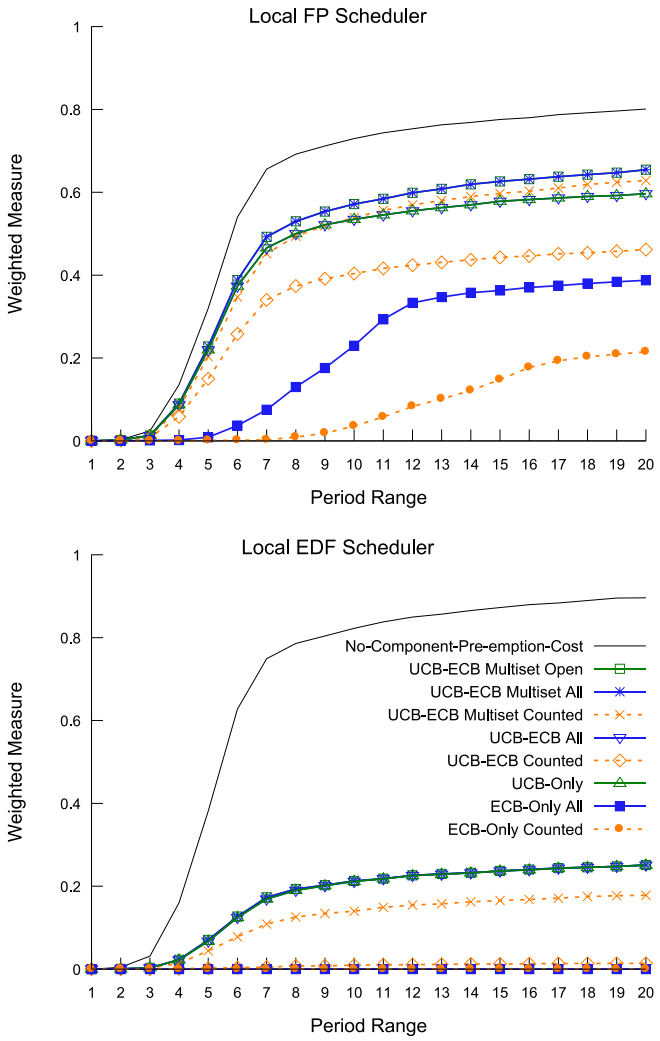



*Figure 7.14 - Weighted measure for varying the period range of tasks from [1, 100]ms to [20, 2000]ms (while fixing the server period at 5ms)*

### 7.6.4 EDF Analysis Investigation

The results for varying the system size, Figure 7.10, and varying the cache utilisation, Figure 7.12, suggest that the inter-component CRPD analysis for a local EDF scheduler has a significant reduction in performance when CRPD costs are increased. In this section we present the results for varying the BRT, which impacts the cost of a pre-emption, and for varying the deadlines of tasks. These results give further insight into this behaviour.

#### Block Reload Time (BRT)

We investigated the effects of varying the BRT, effectively adjusting the costs of a pre-emption in Figure 7.15. With a BRT of 0 there is effectively no CRPD, so all approaches achieve the same weighted measure. Once the BRT increases, the results show that the performance of the approaches that consider inter-

component CRPD under a local EDF scheduler are significantly reduced. This indicates that the analysis for a local EDF scheduler is particularly susceptible to higher pre-emption costs.

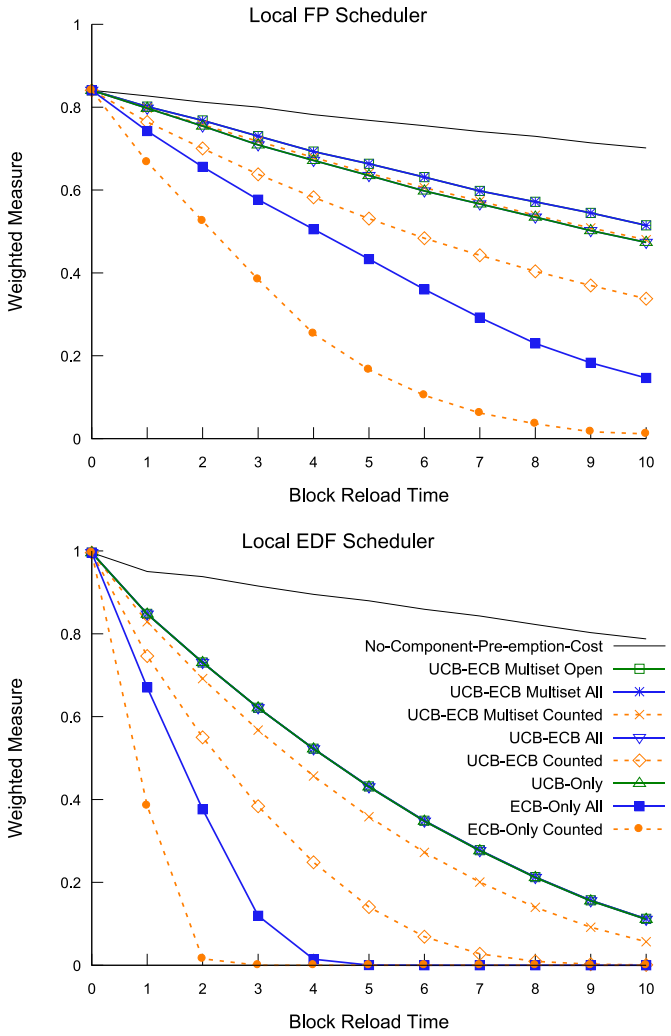



*Figure 7.15 - Weighted measure for varying the block reload time (BRT) from 0 to 10 in steps of 1*

### Deadline Factor

We also varied the task deadlines via $D_i = xT_i$ by varying $x$ from 0.1 to 1 in steps of 0.1. The results are shown in Figure 7.16. Under a local FP scheduler, all approaches showed an increase in the weighted measure as the deadlines are increased. Under a local EDF scheduler, the No-Component-Pre-emption-Cost case performs as expected, showing an increase in schedulability as the deadlines are increased. Additionally, the non UCB-ECB-Multiset approaches also show an increase in the number of schedulable systems. However, the UCB-ECB-Multiset approaches show an increase in the number of systems deemed schedulable, and hence the weighted measure, up to a deadline factor

of 0.8. After this point it shows a reduction in schedulability. This reduction is because although tasks deadlines are relaxed, and thus tasks are less likely to miss them, the number of times that the inter-component CRPD is accounted for is also increased as $E^G(D_k)E_k(t)$ will increase with longer deadlines.

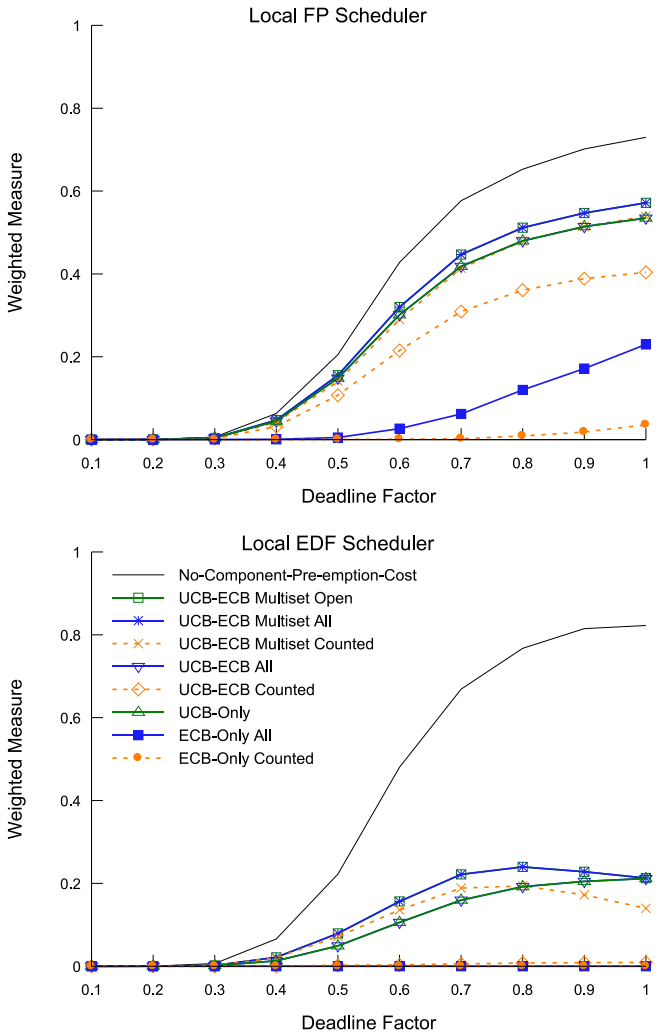



*Figure 7.16 - Weighted measure for varying the task deadlines via $D_i = xT_i$ by varying* x *from 0.1 to 1 in steps of 0.1*

# 7.7 Summary

Hierarchical scheduling provides a means of composing multiple real-time applications onto a single processor, such that the temporal requirements of each application are met. The main contribution of this chapter is a number of approaches for calculating CRPD in hierarchical systems with a global non-pre-emptive scheduler and a local pre-emptive FP or EDF scheduler. This is important because hierarchical scheduling has proved popular in industry as a way of composing applications from multiple vendors as well as re-using

legacy code. However, unless the cache is partitioned these isolated applications can interfere with each other, and so inter-component CRPD must be accounted for.

In this chapter we presented a number of approaches to calculate inter-component CRPD in a hierarchical system with varying levels of sophistication. We showed that when taking inter-component CRPD into account, minimising server periods does not maximise schedulability. Instead, the server period must be carefully selected to minimise inter-component CRPD while still ensuring short deadline tasks meet their time constraints.

We found the analysis for determining inter-component CRPD under a local EDF scheduler deemed a lower number of systems schedulable than the equivalent analysis for a local FP scheduler. This is due to pessimism in the analysis for EDF, and the difficulty in tightly bounding the number of server suspensions that result in inter-component CRPD. Specifically, the analysis considers the number of server suspensions that result in inter-component CRPD based on a task's deadline. In contrast for a local FP scheduler, the analysis can calculate a bound based on a task's response time.

While it was not the best approach in all cases we found the UCB-ECB-Multiset-Open approach, which does not require any information about the other components in the system, to be highly effective. This is a useful result as the approach does not require a closed system. Therefore it can be used when no knowledge of the other components is available and/or cache flushing is used between the execution of different components to ensure isolation and composability.

The UCB-ECB-Multiset-All approach dominates the UCB-ECB-Multiset-Open approach. Therefore, if information about other components is available, it can be used to calculate tighter bounds in cases where not all cache blocks will be evicted by the other components. However, this requires a small enough cache utilisation such that the union of the other components ECBs is less than the size of the cache.

Finally, we note that the presented analysis is not dependent on the scheduling policies of the other components, and hence can be applied to a system where some components are scheduled locally a FP scheduler while others use an EDF scheduler.

# CHAPTER 8. CONCLUSIONS

Accurate analysis of *cache related pre-emption delays* (CRPD) is essential for resource efficient scheduling of complex embedded real-time systems

## 8.1 Summary of Contributions

This thesis set out with the view that CRPD can be a significant factor affecting the schedulability of multi-tasking systems with cache. This is not a new idea. Existing research has recognised this and developed advanced analysis for FP scheduling and some basic analysis for EDF scheduling. However, the focus has been mainly been on FP, with the analysis for EDF being overly pessimistic, and the effects of CRPD have not been previously compared across scheduling algorithms. Furthermore, up until now it has not been possible to account for the effects of CRPD when analysing systems that utilise hierarchical scheduling.

In Chapter 4 we presented a number of new methods for analysing CRPD under EDF scheduling. While there was an existing approach for analysing CRPD under EDF scheduling, we identified the potential for significant pessimism in it which we demonstrated during our evaluation. In particular, we found that the approach was especially vulnerable to high numbers of tasks, high cache utilisation and high UCB percentages, giving pessimistic results in these cases. Our new analysis, specifically the Combined Multiset approach, both dominates and significantly outperformed the existing analysis for EDF.

CRPD is dependent on how tasks are positioned in cache, which is controlled by their layout in memory. In Chapter 5 we presented a technique for optimising task layout in memory so as to increase system schedulability via reduced CRPD. This approach uses *simulated annealing* (SA) driven by schedulability analysis which can account for CRPD in order to evaluate task layouts. By making a series of changes to the layout, the approach can discover a layout that maximises system schedulability. We built functionality into our algorithm to add gaps between tasks in memory, but found that this had little

effect on the schedulability of tasksets for all but the most trivial cases. The fact that adding gaps made little difference is beneficial for a number of reasons. Firstly, the search space is significantly reduced when just considering the order of tasks. Secondly, it is easier to setup a linker to layout tasks with no gaps between them. This is also an important practical point, in that it means that no additional memory space is required.

In Chapter 6, using the new CRPD analysis for EDF presented in Chapter 4, we performed a detailed comparison between FP and EDF scheduling accounting for CRPD. This comparison allowed us to explore the relative impact of CRPD on these two popular scheduling algorithms across a large range of taskset and system parameters in order to gain a better understanding for how CRPD affects system schedulability. We found that when CRPD is considered, the performance gains offered by EDF over FP, while still significant, are somewhat diminished. This is most likely due to greater pessimism in the CRPD analysis for EDF than FP. We also discovered that in configurations that cause relatively high CRPD, optimising task layout can be just as effective as changing the scheduling algorithm from FP to EDF.

Hierarchical scheduling provides a means of running multiple applications or components on a single processor, as found in a partitioned architecture. It is motivated by the need to run multiple components independently of each other without allowing them to impact the functional or temporal behaviour of each other. However, as caches are shared there is the potential for inter-component CRPD to significantly impact schedulability. In Chapter 7, we presented new analysis with varying levels of sophistication that bound CRPD in hierarchical systems. We showed that when taking inter-component CRPD into account, minimising server periods does not maximise schedulability. Instead, the server period must be carefully selected to minimise inter-component CRPD while still ensuring short deadline tasks meet their time constraints. The analysis works for both local FP and EDF schedulers, although the analysis was somewhat pessimistic in the case of EDF. However, the analysis is not dependent on the scheduling policies of the other components, and hence can be used in a system where components are scheduled using different local schedulers. We also noted that in most practical systems components' tasks will occupy an area of memory equal or larger than the size of the cache. We therefore presented an approach which does not require any information about the other components in the system, and found it to be highly effective. This is a useful result as the approach does not require a closed system. It can therefore be used when no knowledge of the other components is available and/or cache flushing is used

between the execution of different components to ensure isolation and composability.

## 8.2 Future Work

The author was recently involved in an investigation led by Altmeyer *et al.* [8] comparing a fully partitioned cache, with one task per partition, against a shared cache without partitions. We found that the gain due to no CRPD did not compensate for the increase in task WCET due to increased inter-task interference. It may be that a hybrid approach, of partitioning groups of tasks with similar periods into their own partition, and then applying layout optimisation, could increase system schedulability further.

Assigning priorities under FP using Deadline Monotonic is not optimal when considering CRPD [53]. Furthermore, schedulability tests that account for CRPD violate some of the conditions that are required for Audsley's OPA algorithm [14]. Therefore, optimal priority assignment for FP with CRPD, without performing an exhaustive search through all possible priority orders which would be intractable for moderately sized tasksets, remains an open problem.

This thesis has focused on techniques for calculating CRPD when performing schedulability analysis on a single core processor. The next major advance would be to extend the work to multi core processors. This brings with it an additional factor to consider, *cache related migration delays*, due to a task being migrated to a different core and losing its private cache. Some work has been conducted which focuses on determining a lower bound via measurements [21] and then utilising those bounds for analysis purposes [128].

# List of Abbreviations

| | |
|---|---|
| ACET | *Average Case Execution Time* |
| BCET | *Best Case Execution Time* |
| BRT | *Block Reload Time* |
| BU | *Breakdown Utilisation* |
| CAC | *Cache Access Classification* |
| CFG | *Control Flow Graph* |
| CRPD | *Cache Related Pre-emption Delays* |
| CSC | *Context Switch Cost* |
| DC-UCB | *Definitely Cached UCB* |
| ECB | *Evicting Cache Block* |
| EDF | *Earliest Deadline First* |
| ETPs | *Execution Time Profiles* |
| FIFO | *First In First Out* |
| FP | *Fixed Priority* |
| GC | *Call Graph* |
| ILP | *Integer Linear Programming* |
| IPET | *Implicit Path Enumeration Technique* |
| LRU | *Least Recently Used* |
| MRU | *Most Recently Used* |
| OPA | *Optimal Priority Assignment* |
| QPA | *Quick convergence Processor-demand Analysis* |
| SA | *Simulated Annealing* |
| SeqPO | *Sequential Priority Order* |
| SRP | *Stack Resource Policy* |

| | |
|---|---|
| TDM | *Time Division Multiplexing* |
| UCB | *Useful Cache Block* |
| VIVU | *Virtual Inlining Virtual Unrolling* |
| WC path | *Worst Case path* |
| WCET | *Worst Case Execution Time* |

# List of Notations

| | |
|---|---|
| $C_i$ | Worst case execution time (determined for non-pre-emptive execution) of task $\tau_i$ |
| $D_i$ | Relative deadline of task $\tau_i$ |
| $D_{max}$ | The largest relative deadline of any task in the taskset |
| $ECB_i$ | Set of ECBs of task $\tau_i$ |
| $ECB^G$ | Set of ECBs of all tasks in $\Gamma^G$ |
| $G$ | A component in a hierarchical system |
| $I^G$ | Maximum execution time of all interrupts in an interval of length $Q^G$ |
| $J_i$ | Release jitter of task $\tau_i$ |
| $L$ | Minimum interval in which it can be guaranteed that an unschedulable taskset will be shown to be unschedulable when determining the processor demand under EDF |
| $m$ | Memory block |
| $M^{description}_{restriction}$ | A multiset of cache blocks, specific to the description of the approach, with an optional restriction. E.g. $M^{ucb}_{i,j}$ is the multiset of UCBs that could be affected by task $\tau_j$ pre-empting task $\tau_i$ |
| $P^G$ | Server period for component $G$ |
| $Q^G$ | Server capacity for component $G$ |
| $R_i$ | Response time of task $\tau_i$ |
| $S^G$ | The server for component $G$ |
| $\tau_i$ | Task $i$ from the taskset $\Gamma$ |
| $T_i$ | Minimum inter-arrival time or period of task $\tau_i$ |
| $T_{max}$ | The largest period of any task in the taskset |
| $U$ | Utilisation of the taskset |
| $U_i$ | Utilisation ($U_i = C_i / T_i$) of task $\tau_i$ |

| | |
|---|---|
| $U^{\gamma}$ | Utilisation due to CRPD incurred by tasks |
| $U^{\gamma G}$ | Utilisation due to inter-component CRPD incurred by tasks in component $G$ |
| $UCB_i$ | Set of UCBs of task $\tau_i$ |
| $UCB^G$ | Set of UCBs of all tasks in $\Gamma^G$ |
| $\gamma_{i,j}$ | CRPD due to a single pre-emption of task $\tau_i$ by task $\tau_j$ under FP scheduling |
| $\gamma'_{i,j}$ | CRPD due to all jobs of task $\tau_j$ executing within the response time of task $\tau_i$ under FP scheduling |
| $\gamma_{t,j}$ | CRPD associated with a pre-emption by a single job of task $\tau_j$ on jobs of other tasks that are both released and have their deadlines in an interval of length $t$ under EDF scheduling |
| $\gamma'_{t,j}$ | CRPD due to the maximum number $E_j(t)$ of pre-emptions by jobs of task $\tau_j$ that have their release times and absolute deadlines in an interval of length $t$ under EDF scheduling |
| $\gamma_i'^G$ | CRPD incurred by task $\tau_i$ in component $G$ caused by tasks in the other components running while the server ($S^G$) for component $G$ is suspended with a local FP scheduler |
| $\gamma_t'^G$ | CRPD incurred by tasks in component $G$ due to tasks in the other components running while the server ($S^G$) for component $G$ is suspended with a local EDF scheduler |
| $\Gamma$ | Taskset made up of a fixed number of tasks ($\tau_1..\tau_n$) where $n$ is a positive integer |
| $\Gamma^G$ | Set of tasks in component $G$ from the taskset $\Gamma$ |
| $\Psi$ | Set of components in a hierarchical system |
| aff($i$,$j$) | The set of tasks that may be pre-empted by task $\tau_j$ and have at least the priority of task $\tau_i$. aff($i$,$j$) = hep($i$) ∩ lp($j$) under FP scheduling |
| aff($t$,$j$) | The set of tasks that can be pre-empted by jobs of task $\tau_j$ in an interval of length $t$ under EDF scheduling |

| | |
|---|---|
| aff($G$,$t$) | Set of tasks in component $G$ whose relative deadlines are less than or equal to $t$ with a local EDF scheduler |
| $E_k(R_i)$ | Maximum number of jobs of task $\tau_k$ that can execute during the response time of task $\tau_i$ |
| $E_j(t)$ | The maximum number of jobs of task $\tau_j$ that can have both their release times and their deadlines in an interval of length $t$ |
| $E^G(t)$ | The maximum number of times server $S^G$ can be both suspended and resumed within an internal of length $t$ |
| $h(t)$ | Processor demand bound function used to determine demand on the processor within an interval of length $t$ under EDF scheduling |
| hp($i$) | Set of tasks that may have a higher priority, and can pre-empt task $\tau_{i,}$ |
| hp($G$, $i$) | Sets of tasks in component $G$ with higher priorities than task $\tau_i$ with a local FP scheduler |
| hep($i$) | Sets of tasks with higher or equal priorities to task $\tau_i$ under FP scheduling |
| hep($G$, $i$) | Sets of tasks in component $G$ with higher or equal priorities to task $\tau_i$ with a local FP scheduler |
| $isbf^G(c)$ | Inverse supply bound function for component $G$. Used to determine the maximum amount of time needed by a specific server to supply some capacity $c$ |
| lp($i$) | Sets of tasks with lower priorities than task $\tau_i$ under FP scheduling |
| lep($i$) | Sets of tasks with lower or equal priorities to task $\tau_i$ under FP scheduling |
| $P_j(D_i)$ | The maximum number of times that jobs of task $\tau_j$ can pre-empt a single job of task $\tau_i$ under EDF scheduling |
| $X^Z(S^G,t)$ | The number of executions of server $S^Z$ while server $S^G$ is suspended that results in tasks from component $Z$ evicting cache blocks that tasks in component $G$ might have loaded and need to reload |

# References


[1] AbsInt. aiT Worst-Case Execution Time Analyzers. http://www.absint.com/aiT_WCET.pdf (Last accessed September 2014)

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache Behavior Prediction by Abstract Interpretation," in *Proceedings of the 3rd International Symposium on Static Analysis*, 1996, pp. 52-66.

[3] S. Altmeyer and C. Burguiere, "A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, Dublin, Ireland, 2009, pp. 109-118.

[4] S. Altmeyer and C. Burguière, "Cache-related Preemption Delay via Useful Cache Blocks: Survey and Redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707-719, August 2011.

[5] S. Altmeyer and R. I. Davis, "On the Correctness, Optimality and Precision of Static Probabilistic Timing Analysis," in *Proceedings Design Automation and Test Europe (DATE)*, Dresden, Germany, 2014.

[6] S. Altmeyer, R.I. Davis, and C. Maiza, "Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011, pp. 261-271.

[7] S. Altmeyer, R.I. Davis, and C. Maiza, "Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499-512, September 2012.

[8] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Evaluation of Cache Partitioning for Hard Real-Time Systems," in *Proceedings 26th Euromicro Conference on Real-Time Systems (ECRTS)*, Madrid, Spain, 2014, pp. 15-26.

[9] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience Analysis: Tightening the CRPD Bound for Set-Associative Caches," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Stockholm, Sweden, 2010, pp. 153-162.

[10] ARINC, "ARINC 651: Design Guidance for Integrated Modular Avionics," Airlines Electronic Engineering Committee (AEEC), 1991.

[11] ARINC, "ARINC 653: Avionics Application Software Standard Interface (Draft 15)," Airlines Electronic Engineering Committee (AEEC), 1996.

[12] ARM. CoreSight Trace Macrocells. http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php (Last accessed September 2014)

[13] M. Åsberg, M. Behnam, and T. Nolte, "An Experimental Evaluation of Synchronization Protocal Mechanisms in the Domain of Hierarchical Fixed-Priority Scheduling," in *Proceedings of the 21st International Conference*

*on Real-Time and Network Systems (RTNS)*, Sophia Antipolis, France, 2013.

[14] N. C. Audsley, "On Priority Asignment in Fixed Priority Scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39-44, May 2001.

[15] N. C. Audsley, A. Burns, M. Richardson, and A.J Wellings, "Applying new Scheduling Theory to Static Priority Preemptive Scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284-292, 1993.

[16] T. P. Baker, "Stack-Based Scheduling for Realtime Processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67-99, 1991.

[17] S. Baruah, "The Limited-Preemption Uniprocessor Scheduling of Sporadic Task Systems," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, Palma de Mallorca, Balearic Islands, Spain, 2005, pp. 137-144.

[18] S. Baruah and A. Burns, "Sustainable Scheduling Analysis," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, 2006, pp. 159-168.

[19] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptive Scheduling Hard-Real-Time Sporadic Tasks on One Processor," in *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*, Lake Buena Vista, Florida, USA, 1990, pp. 182-190.

[20] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301-324, 1990.

[21] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proceedings of Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, Brussels, Belgum, 2010, pp. 33-44.

[22] A. Bastoni, B. Brandenburg, and J. Anderson, "Is Semi-Partitioned Scheduling Practical?," in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, Porto, Portugal, 2011, pp. 125-135. [Online]. Extended version: http://www.cs.unc.edu/~anderson/papers/ecrts11-long.pdf

[23] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing Real-Time Open Systems," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, 2007, pp. 279-288.

[24] C. Berg, "PLRU Cache Domino Effects," in *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis, in conjunction with the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, Dresden, Germany, 2006.

[25] G. Bernat, A. Colin, and S. Patters, "pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems," University of York, York, UK, Technical Report YCS-2003-353, 2003.

[26] G. Bernat, A. Colin, and S. Petters, "WCET Analysis of Probabilistic Hard Real-Time Systems," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002, pp. 279-288.

[27] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption Points Placement for Sporadic Task Sets," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Bruxelles, Belgium, 2010, pp. 251-260.

[28] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal Selection of Preemption Points to Minimize Preemption Overhead," in *Proceedings of 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, Porto, Portugal, 2011, pp. 217-227.

[29] A. Betts and G. Bernat, "Issues using the Nexus Interface for Measurement-Based WCET Analysis," in *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis*, Palma de Mallorca, Balearic Islands, Spain, 2007.

[30] A. Betts and G. Bernat, "Tree-Based WCET Analysis on Instrumentation Point Graphs," in *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, Gyeongju, Korea, 2006, pp. 558-565.

[31] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel, "WCET Coverage for Pipelines," University of York, York, Technical Report for the ARTIST2 Network of Excellence 2006.

[32] E. Bini and G. Buttazzo, "Measuring the Performance of Schedulability Tests ," *Real-Time Systems*, vol. 30, no. 1, pp. 129-154, 2005.

[33] P. Bo, N. Fisher, and M. Bertogna, "Explicit Preemption Placement for Real-Time Conditional Code," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, Madrid, Spain, 2014.

[34] B. B. Brandenburg, H. Leontyev, and J. H. Anderson, "Accounting for Interrupts in Multiprocessor Real-Time Systems," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Beijing, 2009, pp. 273 - 283.

[35] C. Burguière, J. Reineke, and S. Altmeyer, "Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions," in *Proceeding of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET)*, Dublin, Ireland, 2009.

[36] A. Burns, "Premptive Priority-Based Scheduling: An Appropriate Engineering Approach," in *Advances in Real-Time Systems*.: Prentice Hall, 1994, pp. 225-248.

[37] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems," in *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1996, pp. 204-212.

[38] G. C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems*, vol. 29, no. 1, pp. 5-26, January 2005.

[39] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited Preemptive Scheduling for Real-Time Systems: A Survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3-15, March 2013.

[40] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying Static WCET Analysis to Automotive Communication Software," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Palma de Mallorca, Balearic Islands, Spain, 2005, pp. 249-258.

[41] A. M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix, "Static Use of Locking Caches in Multitask Preemptive Real-Time Systems," in *Proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop*, 2001.

[42] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "PROARTIS: Probabilistically Analysable Real-Time Systems," *ACM Transactions on Embedded Computing Systems - Special section on Probabilistic Embedded Computing*, 2013.

[43] Certification Authorities Software Team (CAST), "CAST-20 Addressing Cache in Airborne Systems and Equiptment," Position Paper 2003.

[44] A. Colin and S. M. Petters, "Experimental Evaluation of Code Properties for WCET Analysis," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, 2003, pp. 190-199.

[45] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238-252.

[46] R. I. Davis, "A Review of Fixed Priority and EDF Scheduling for Hard Real-Time Uniprocessor Systems," *ACM SIGBED Review - Special Issue on the 3rd Embedded Operating Systems Workshop (Ewili 2013)*, vol. 11, no. 1, pp. 8-19, 2014.

[47] R. I. Davis and M. Bertogna, "Optimal Fixed Priority Scheduling with Deferred Pre-emption," in *Proceedings 33rd IEEE Real-Time Systems Symposium (RTSS*, San Jan, Puerto Rico, 2012, pp. 39-50.

[48] R. I. Davis and M. Bertogna, "Optimal Fixed Priority Scheduling with Deferred Pre-emption," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS'12)*, San Juan, Puerto Rico, 2012, pp. 39 - 50.

[49] R. I. Davis and A. Burns, "An Investigation into Server Parameter Selection for Hierarchical Fixed Priority Pre-emptive Systems," in *Proceedings 16th International Conference on Real-Time and Network Systems (RTNS)*, Renne, France, 2008, pp. 19-28.

[50] R. I. Davis and A. Burns, "Hierarchical Fixed Priority Pre-emptive Scheduling," in *Proceedings of the 26th IEEE Real-Time Systems Symposium*

*(RTSS)*, 2005.

[51] R. I. Davis and A. Burns, "Resource Sharing in Hierarchical Fixed Priority Pre-Emptive Systems," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, Rio de Janeiro, Brazil, 2006, pp. 257-270.

[52] R. I. Davis and A. Burns, "Robust Priority Assignment for Fixed Priority Real-Time Systems," in *Proceedings 28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, Arizona, USA, 2009, pp. 3-14.

[53] R. I. Davis, L. Cucu-Grosjean, and A. Burns, "Getting One's Priorities Right: A Review of Priority Assignment in Fixed Priority Real-Time Systems," *Real-Time Systems (Under Submission)*.

[54] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of Probabilistic Cache Related Pre-emption Delays," in *Proceedings 25th Euromicro Conference on Real-Time Systems (ECRTS)*, Paris, France, 2013, pp. 168-179.

[55] R. I. Davis, A. Zabos, and A. Burns, "Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1261-1276, September 2008.

[56] Z. Deng and J. W. S. Liu, "Scheduling Real-Time Applications in Open Environment," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, San Francisco, USA, 1997.

[57] M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," in *Proceedings of the International Federation for Information Processing (IFIP) Congress*, 1974, pp. 807-813.

[58] R. Ernst and W. Ye, "Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, USA, 1997, pp. 598-604.

[59] H. Falk and H. Kotthaus, "WCET-driven Cache-aware Code Positioning," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Taipei, Taiwan, 2011, pp. 145-154.

[60] X. Feng and A. K. Mok, "A Model of Hierarchical Real-Time Virtual Resources," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, Austin, TX, USA, 2002, pp. 26-35.

[61] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, Theiling. H., S. Thesing, and R. Wilhelm, "Reliable and Precise WCET Determination for a Real-Life Processor," *Lecture Notes in Computer Science*, vol. 2211, pp. 469-485, 2001.

[62] C. Ferdinand and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems*, vol. 17, no. 2, pp. 131-181, November 1999.

[63] C. Ferdinand and R. Wilhelm, "On Predicting Data Cache Behavior for

Real-Time Systems," in *Proceedings of the ACM SIGPLAN Workshop on Languages Compilers, and Tools for Embedded Systems (LCETS)*, Montreal, Canada, 1998, pp. 16-30.

[64] N. Fisher and F. Dewan, "A Bandwidth Allocation Scheme for Compositional Real-time Systems with Periodic Resources," *Real-Time Systems*, vol. 48, no. 3, pp. 223-263, 2012.

[65] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems," in *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS)*, Pisa, Itally, 1995, pp. 152-161.

[66] G. Gebhard and S. Altmeyer, "Optimal Task Placement to Improve Cache Performance," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, Salzburg, Austria, 2007, pp. 259-268.

[67] L. George, N. Rivierre, and M. Spuri, "Preemptive and Non-Preemptive Real-Time Uniprocessor Scheduling," INRIA, Technical Report 2966, 1996.

[68] J. Gustafsson, A. Betts, A. Ermedah, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, Brussels, Belgium, September 2010, pp. 137-147.

[69] D. Hardy and I. Puaut, "WCET Analysis of Instruction Cache Hierarchies," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 677-694, August 2011.

[70] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *BCS Computer Journal*, vol. 29, no. 5, pp. 390-395, 1986.

[71] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for Cache-Related Preemption Delay in Dynamic Priority Schedulability Analysis," in *Design, Automation and Test in Europe Conference and Exposition (DATE)*, Nice, France, 2007, pp. 1623-1628.

[72] S. Kim, S. Min, and R. Ha, "Efficient Worst Case Timing Analysis of Data Caching," in *Proceedings of the 2nd IEEE Real-Time Technology and Application Symposium (RTAS)*, Boston, MA, USA, 1996, pp. 230-240.

[73] D. B. Kirk and J. K. Strosnider, "SMART (Strategic Memory Allocation for Real-Time) Cache Design," in *Proceedings 11th Real-Time Systems Symposium (RTSS)*, Lake Buena Vista, FL , USA, 1990, pp. 322-330.

[74] R. Kirner and P. Puschner, "Transformation of Path Information for WCET Analysis During Compilation," in *Proceedings of the 13th Euromicro Conference of Real-Time Systems (ECRTS)*, 2001, pp. 29-36.

[75] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using Measurements as a Complement to Static Worst-Case Execution Time Analysis," *Intelligent Systems at the Service of Mankind*, vol. 2, December 2005.

[76] T-W. Kuo and C-H. Li, "A Fixed Priority Driven Open Environment for Real-Time Applications," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.

[77] C. Lee, J. Hahn, Y. Seo, S. Min, H. Ha, S. Hong, C. Park, M. Lee, and C. Kim, "Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700-713, June 1998.

[78] J. Lehoczky, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings of the 10th Real Time Systems Symposium (RTSS)*, Santa Monica, California, USA, 1989, pp. 166-171.

[79] J. Y.-T. Leung and M. L. Merrill, "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks," *Information Processing Letters*, vol. 11, no. 3, pp. 115-118, 1980.

[80] J. Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, vol. 2, no. 2, pp. 237-250, 1982.

[81] Y-T. S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," in *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference* , San Francisco, USA, 1995, pp. 456-461.

[82] G. Lipari and S. K. Baruah, "Efficient Scheduling of Real-time Multi-task Applications in Dynamic Systems," in *Proceddings Real-Time Technology and Applications Symposium (RTAS)*, 2000, pp. 166-175.

[83] G. Lipari and E. Bini, "A Methodology for Designing Hierarchical Scheduling Systems," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257-269, December 2005.

[84] G. Lipari, J. Carpenter, and S. Baruah, "A Framework for Achieving Inter-application Isolation in Multiprogrammed, Hard Real-time Environments," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, FL, USA, 2000, pp. 217-226.

[85] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, January 1973.

[86] P. Lokuciejewski, H. Falk, and P. Marwedel, "WCET-driven Cache-based Procedure Positioning Optimizations," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, Prague, Czech Republic, 2008, pp. 321-330.

[87] P. Lokuciejewski, H. Falk, P. Marwedel, and H. Theiling, "WCET-Driven, Code-Size Critical Procedure Cloning," in *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, Munich, Germany, 2008, pp. 21-30.

[88] T. Lundqvist and P. Stenstrom, "Timing Anomalies in Dynamically Scheduled Microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, Phoenix, AZ , USA, 1999, pp. 12-21.

[89] F. Mueller, "Compiler Support for Software-based Cache," *SIGPLAN Not.*, vol. 30, no. 11, pp. 125-133, 1995.

[90] F. Mueller, "Static Cache Simulation and its Applications," PhD Thesis 1994.

[91] F. Mueller, "Timing Analysis for Instrction Caches," *Real-Time Systems*, vol. 18, no. 2-3, pp. 217-247, May 2000.

[92] F. Mueller and J. Wegener, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," in *Proceedings of the 4th Real-Time Technology and Application Symposium (RTAS)*, Denver, Colorado, USA, 1998, pp. 144-154.

[93] Nexus. Nexus 5001 Forum. http://www.nexus5001.org (Last accessed September 2014)

[94] C. Y. Park, "Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths," *Real-Time Systems*, vol. 5, no. 1, pp. 31-62, 1993.

[95] C. Y. Park and A. C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Computer*, vol. 24, no. 5, pp. 48-57, May 1991.

[96] S. M. Petters, "Bounding the Execution Time of Real-Time Tasks on Modern Processors," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, 2000, pp. 498-502.

[97] K. Pettis and R. Hansen, "Profile Guided Code Positioning," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 16-27.

[98] S. Plazar, P. Lokuciejewski, and P. Marwedel, "WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems," in *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time Analysis (WCET)*, Dublin, Ireland, 2009.

[99] P. Puschner and A. Burns, "Guest Editorial: A Review of Worst-Case Execution-Time Analysis," *Real-Time Systems*, vol. 18, no. 2, pp. 115-128, May 2000.

[100] P. Puschner and C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," *The Journal of Real-Time Systems*, vol. 1, no. 2, pp. 159-176, 1989.

[101] P. P. Puschner and A. V. Schedl, "Computing Maximum Task Execution Times — A Graph-Based Approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67-91, 1997.

[102] E. Quinones, E. D. Berger, G. Bernat, and F. J. Cazorla, "Using Randomized Caches in Probabilistic Real-Time Systems," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009, pp. 129-138.

[103] Rapita Systems Ltd. RapiTime. http://www.rapitasystems.com/products/RapiTime (Last accessed September 2014)

[104] J. Reineke, "Caches in WCET Analysis," Universität des Saarlandes, Saarbrücken, PhD Thesis 2008.

[105] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing Predictability for Cache Replacement Polcies," *Real-Time Systems*, vol. 37, no. 2, pp. 99-122, November 2007.

[106] K. Richter, "Compositional Scheduling Analysis Using Standard Event Models," Technical University Carolo-Wilhelmina of Braunschweig, PhD Dissertation 2005.

[107] I. Ripoll, A. Crespo, and A. K. Mok, "Improvement in Feasibility Testing for Real-Time Tasks," *Real-Time Systems*, vol. 11, no. 1, pp. 19-39, 1996.

[108] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein, "Analysis of Hierarchical Fixed Priority Scheduling," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, Vienna, Austria, 2002, pp. 173-181.

[109] J. Schneider, "Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-time Systems," in *Proceddings of the 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, USA, 2000, pp. 195-204.

[110] F. Sebek, "Measuring Cache Related Pre-emption Delay on a Multiprocessor Real-Time System," Dept. of Computer Engineering, Mälardalen University, Västerås, Sweden, 2001.

[111] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, 2003, pp. 2-13.

[112] M. Spuri, "Analysis of Deadline Schedule Real-Time Systems," INRIA, Technical Report 2772, 1996.

[113] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay," in *In Proceedings 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Balearic Islands, Spain, 2005, pp. 41-48.

[114] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, Miami, Florida, USA, 2005, pp. 223-232.

[115] Y. Tan and V. Mooney, "Timing Analysis for Preemptive Multitasking Real-Time Systems with Caches," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 1, February 2007.

[116] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and Precise WCET Prediction by Separated Cache and Path Analyses," *Real-Time Systems*, vol. 18, no. 2, pp. 157-179, 2000.

[117] L. Thiele and R. Wilhelm, "Design for Timing Predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157-177, November 2004.

[118] Y. Wang and M. Saksena, "Scheduling Fixed-Priority Tasks with Preemption," in *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Hong Kong, China, 1999, pp. 328-335.

[119] C. B. Watkins and R. Walter, "Transitioning from Federated Avionics Architectures to Integrated Modular Avionics," in *Proceedings of the 26th IEE/AIAA Digital Avionics Systems Conference (DASC)*, 2007.

[120] L. Wehmeyer and P. Marwedel, "Influence of Onchip Scratchpad Memories on WCET Prediction," in *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[121] J. Whitham and N. Audsley, "Investigating Average versus Worst-Case Timing Behavior of Data Caches and Data Scratchpads," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, 2010, pp. 165-174.

[122] J. Whitham and N. Audsley, "Studying the Applicability of the Scratchpad Memory Management Unit," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Stockholm, Sweden, 2010, pp. 205-214.

[123] J. Whitham and N. Audsley, "The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study," University of York, York, UK, Technical Report YCS-2009-439, 2009.

[124] J. Whitham, N. C. Audsley, and R. I. Davis, "Explicit Reservation of Cache Memory in a Predictable, Preemptive Multitasking Real-time System," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, 2014.

[125] J. Whitham, R. I. Davis, N. C. Audsley, S. Altmeyer, and C. Maiza, "Investigation of Scratchpad Memory for Preemptive Multitasking," in *Poceedings 33rd IEEE Real-Time Systems Symposium (RTSS)*, San Juan, Puerto Rico, 2012, pp. 3-13.

[126] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, April 2008.

[127] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems," *IEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966-978, July 2009.

[128] M. Xu, L. T.X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill, "Cache-Aware Compositional Analysis of Real-Time Multicore Virtualization Platforms," in *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS)*, Vancouver, Canada, 2013.

[129] F. Zhang and A. Burns, "Schedulability Analysis for Real-Time Systems with EDF Scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1250-1258, September 2009.

[130] W. Zhao, D. Whalley, C. Healy, and F. Mueller, "Improving WCET by Applying a WC Code Positioning Optimization," *ACM Transcations on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 335-365, December 2005.

[131] W. Zhao, D. Whalley, C. Healy, and F. Mueller, "WCET Code Positioning," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, 2004, pp. 81-91.