Optimising Task Layout to Increase Schedulability via Reduced Cache Related Pre-emption Delays

Will Lunniss[1,3], Sebastian Altmeyer[2], Robert I. Davis[1]

[1]Department of Computer Science
University of York
York, UK
{wl510,rob.davis}@york.ac.uk

[2] Department of Computer Science
Saarland University
Saarbrücken, Germany
altmeyer@cs.uni-sb.de

[3] Rapita Systems Ltd.
IT Center
York Science Park
York, UK
wlunniss@rapitasystems.com

ABSTRACT

Cache memories have been introduced into embedded systems to prevent memory access times from becoming an unacceptable performance bottleneck. For hard real-time systems, it is vital that an accurate estimate of the worst-case response time for each task can be determined. Memory and cache are split into blocks containing instructions and data. During a pre-emption, blocks from the pre-empting task can evict those of the pre-empted task. When the pre-empted task is resumed, if it then has to re-load the evicited blocks, cache related pre-emption delays (CRPD) are introduced which then affect the worst-case response times of the task. Because the position of code in memory determines where the code will be placed in cache, different layouts result in different CRPD and worst-case response times for tasks. We introduce an approach that uses simulated annealing to find layouts that minimise the CRPD incurred due to a pre-emption. This in turn reduces the worst-case response times of tasks, which increases the schedulability of the taskset. We use schedulability analysis that captures whether a block will have to be re-loaded after a pre-emption, to drive the algorithm towards a near optimal solution. After explaining our approach, we present a number of experiments which demonstrate its effectiveness for a number of different system, task and cache configurations.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems - *Real-time and embedded systems*

Keywords

Cache Related Pre-emption Delay (CRPD), Task layout, Fixed priroity pre-emptive scheduling, Response time analysis

1. INTRODUCTION

Over the past years, processor speeds have increased dramatically leaving memory access times as a major performance bottleneck. To bridge this ever increasing gap, caches have been introduced between the processor and memory; however, they introduce significant complexity when trying to verify timing properties of the system.

Real-time systems, especially hard real-time systems, have very stringent timing requirements and the *worst-case response time* of each task must be known, or safely estimated. The worst-case response time of a task is the longest possible time between the task becoming ready to start and it finishing executing, including the time that the task was unable to execute due to pre-emption or busy hardware resources. In order for a system to be schedulable, the response time of every task must be less than or equal to its deadline.

While memory will contain a mixture of instructions and data, caches can either be instruction only, data only or unified, containing instructions and data. In this paper, we only consider instruction caches. The instructions in memory and cache are stored in blocks. The size of a block is almost always bigger than the size of an instruction, for example, 4 instructions per block. When the CPU executes an instruction, it first tries to fetch the instruction from cache. If the block containing the instruction is in cache, then a cache hit occurs. If the block is not in cache then a cache miss occurs, and the block is fetched from the slower memory and stored in the cache. In modern architectures, a cache miss is an order of magnitude slower than a cache hit, it is therefore highly desirable to minimise cache misses. In pre-emptive multi-tasking systems, caches introduce additional *cache related pre-emption delays* (CRPD) caused by the need to re-fetch blocks belonging to the pre-empted task which were evicted from cache by the pre-empting task.

As the position of code in memory affects where blocks are positioned in cache, and therefore, which blocks they evict, controlling and optimising the layout of code in memory can help to reduce the CRPD. In this paper we present an approach that uses *simulated annealing* (SA) to find new layouts for tasks that helps to reduce the CRPD. We use cache aware schedulability analysis to guide the SA towards an optimal layout. We assume that tasks do not share any code, therefore altering the position of tasks in memory, will not affect the *worst-case execution time* (WCET) [1] of the task, just the worst-case response time due to CRPD. The approach is evaluated using a case study based on real code, and an empirical study based on synthetically generated tasks.

This paper builds on the ideas of Gebhard and Altmeyer [16] who used schedulability analysis to drive laying out tasks in order to minimise cache conflicts. However, the analysis used in [16] was not able to capture whether a block evicted from cache would need to be reloaded, and therefore treated all blocks as equal. This

[1] WCET of the task when executed non-pre-emptively

work uses the concept of *useful cache blocks* (UCBs) and *evicting cache blocks* (ECBs) based on the work by Lee *et al.* [18]. ECBs are blocks that may be loaded into cache by the task during its execution. Out of the ECBs, some of them may also be UCBs, which are blocks that are reused once they have been loaded into cache, before potentially being evicted by the task, but not counting evictions from other pre-empting tasks. If a UCB is evicted by a pre-empting task, additional CRPD may be introduced as the UCB may have to be re-loaded when it otherwise would not have been. This work uses schedulability analysis introduced by Altmeyer *et al.* [2], [3] which is able to use the UCBs and ECBs to more accurately calculate the schedulability of the taskset. This analysis is used to drive *simulated annealing* (SA) towards an optimal task layout that increases the schedulability of a taskset by reducing the CRPD and hence the worst-case response time of tasks that would otherwise miss their deadlines.

1.1 Related Work

There are a number of approaches for dealing with caches in pre-emptive multitasking systems. One of these approaches is cache locking. Cache locking is used to lock the cache which prevents blocks from being evicted. The aim of the algorithms used is to find an optimal selection of blocks to be locked into cache. There are two main approaches, static cache locking, and dynamic cache locking. Static cache locking loads some initial content into the cache, and then locks it for the remainder of the systems' execution. For both approaches, the challenge is deciding what to lock into cache, and when. Campoy *et al.* in 2001 [12] used a genetic algorithm while Puaut *et al.* in 2002 [22] used a greedy algorithm to select cache contents. Both algorithms performed similarly [13] with the genetic algorithm performing slightly better. Static cache locking is however only really suitable for systems with a small number of frequently called procedures that can all be locked into cache at once. An alternative is dynamic cache locking which locks a defined cache contents into the cache at a number of different points, usually at the start of each task, and after pre-emption. This increases the predictability of the system, and facilitates more accurate WCET and worst-case response time estimation. Campoy *et al.* in 2002 [11] adapted their genetic algorithm from [12] to work with dynamic cache locking as did Arnaud and Puaut in 2006 [4] for their greedy algorithm from [22]. In more recent work, Liu *et al.* used execution flow graphs and trees that they solved using ILP [20] for static and dynamic cache locking. They again found that dynamic cache locking performs better than static cache locking when the cache is relatively small compared to the size of the code; however, dynamic cache locking is still not optimal. If for example a task is pre-empted when it has nearly finished executing, analysis such as [11] assumes that the pre-empted tasks' entire cache contents must be reloaded as it does not account for that fact that the task may at that stage only need to access a small percentage of the blocks.

A different approach is to limit pre-emption. Bertogna in 2011 [7] used *fixed pre-emption points* (FPP) based on Burns' work in 1994 [9] to limit pre-emption to known points in a task, facilitating the calculation of the CRPD at these known points; however, the possible pre-emption points need to be defined which makes the approach somewhat manual. Additionally, because the analysis assumes the entire cache is invalidated, the estimated CRPD can still be overly pessimistic.

Code positioning techniques rearrange the code in memory to improve cache performance. Lokuciejewski *et al.* in 2008 [21] applied procedure positioning to reduce the WCET for systems with cache. They placed procedures on the *worst-case path* that call each other frequently close together to reduce conflicts. Falk *et al.* [15] recently used block and procedure positioning based on cache conflict graphs to reduce the WCET estimate. Unlike previous work, they accounted for the cache configuration including the associativity, size and replacement policy. In contrast to the work in this paper, all of these techniques only consider single tasks without pre-emption.

Gebhard and Altmeyer [16] in 2007 used schedulability analysis to evaluate different layouts. They performed their analysis on a pre-emptive multi-tasking system with the goal of preventing pre-empting tasks from evicting the pre-empted task's blocks from cache by positioning whole tasks contiguously in memory. The layouts were 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, and hence whether there is a potential for conflict. It also takes into account the lifespan of blocks due to the replacement policy. However, all of the tasks' ECBs are effectively treated as UCBs. Therefore, the cost is proportional to the number of blocks belonging to the pre-empted task that reside in the same locations as the pre-empting tasks' blocks. 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 returned by the cost function. This was because no consideration was given to the actual code inside the tasks and all the tasks' ECBs were treated as UCBs. If the actual UCBs are positioned so that they are safe from eviction, then the overall number of misses can potentially be reduced significantly more than when the blocks are positioned to minimise the total number of evictions. This is the aim of the approach presented in this paper.

1.2 Organisation

The remainder of this paper is organised as follows. Section 2 introduces the system model, terminology and notation. Section 3 details CRPD aware schedulability analysis while section 4 explains why different layouts cause different numbers of cache conflicts. Section 5 details our approach to finding improved task layouts. Section 6 presents the case study and section 7 presents the experiments based on synthetically generated tasksets. Finally, section 8 concludes with a summary and directions for future work.

2. SYSTEM MODEL, TERMINOLOGY AND NOTATION

In this section we describe the system model terminology and notation used in the remainder of the paper.

It is assumed that *fixed priority pre-emptive scheduling* is used on a single processor. Each taskset contains a fixed number of tasks ($\tau_1..\tau_n$) with unique fixed priorities. The priority of task τ_i, is i, where a priority of 1 is the highest and n is the lowest. Each task, τ_i, has a deadline D_i, WCET C_i, minimum inter-arrival time or period T_i and release jitter J_i. Tasks have constrained deadlines, i.e. the deadline of each task is less than or equal to its period. Each task has a utilisation U_i, where $U_i = C_i / T_i$. Each task can also have a blocking time B_i which is the time that the task is blocked from executing because it is waiting for access to a shared resource other than the processor. To determine which tasks can pre-empt each other, the following sets are used. *hp*(*i*) and *lp*(*i*) are the sets of tasks with higher and lower priorities than task τ_i, and *hep*(*i*) and *lep*(*i*) are the sets containing tasks with higher or equal and lower or equal priorities to task τ_i. Additionally, $aff(i,j) = hep(i) \cap lp(j)$ is used to represents all of the tasks that may be pre-empted by task τ_j and have at least the priority of task τ_i. Finally, each task τ_i has a set UCB_i of UCBs, and a set ECB_i of ECBs, represented by a set of integers. If for example, task τ_1 contains 4 ECBs located in cache sets 1 to 4 and cache sets 2 and 4 are also UCBs, this would be represented using $ECB_1 = \{1,2,3,4\}$ and $UCB_1 = \{2,4\}$.

It is assumed that the cache is direct mapped and that tasks do not share any code. Finally it is assumed that there is no intermediate virtual memory layer, i.e. the position of tasks in the linked executable fully determines their position in memory and cache.

3. SCHEDULABILITY ANALYSIS WITH CRPD

In this section, we summarise schedulability analysis for fixed priority pre-emptive systems with CRPD, described in detail in [2], [3].

Schedulability tests are used to determine if a taskset is schedulable, i.e. all the tasks will meet their deadlines given the worst-case pattern of arrivals and execution. For a given taskset, the response time R_i for each task τ_i, can be calculated and compared against the tasks' deadline, D_i. If every task in the taskset meets its deadline, then the taskset is schedulable. The equation used to calculate R_i is defined as [5]:

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

Equation (1) can be solved using fixed point iteration. Iteration continues until either $R_i^{\alpha} > D_i - J_i$ in which case the task is unschedulable, or until $R_i^{\alpha} = R_i^{\alpha}$ in which case the task is schedulable and has a worst-case response time of R_i^{α}.

Note the convergence of (1) may be speeded up using the techniques described in [14].

To account for the CRPD, a component $\gamma_{i\,j}$ is introduced which is the cost associated with a pre-emption by task τ_j during the response time of task τ_i. This is found by using the cost incurred when reloading a block, the *block reload time* (BRT), multiplied by the number of blocks which may need to be reloaded after a pre-emption. Incorporating $\gamma_{i\,j}$ into (1) gives a revised equation for R_i as [10]:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_i}{T_j} \right\rceil \left(C_j + \gamma_{i,j}\right) \tag{2}$$

A number of different methods can be used to compute $\gamma_{i\,j}$ as described by Altmeyer et al in [2], which we will now summarise. The UCB Union method [23] accounts for the effects of nested pre-emption by assuming that the UCBs of task τ_i, as well as the UCBs of tasks with priorities higher than that of τ_i but lower than that of τ_j could all be evicted by τ_j:

$$\gamma_{i,j}^{UCB-U} = BRT \bullet \left| \left(\bigcup_{\forall k \in aff(i,j)} UCB_k \right) \cap ECB_j \right| \tag{3}$$

The alternative ECB-Union method [2] accounts for nested pre-emptions by assuming that when task τ_j pre-empts some task τ_k within the response time of task τ_i, task τ_j may already have itself been pre-empted by all higher priority tasks:

$$\gamma_{i,j}^{ECB-U} = BRT \bullet \max_{\forall k \in aff(i,j)} \left\{ \left| UCB_k \cap \left(\bigcup_{h \in hp(j) \cup \{j\}} ECB_h \right) \right| \right\} \tag{4}$$

As the two methods are incomparable, the smallest response time can be taken. This is because if at least one of the approaches deems a taskset schedulable, then it is schedulable. Giving the definition of the combined approach [2] as:

$$R_i = \min\left(R_i^{UCB-U}, R_i^{ECB-U}\right) \tag{5}$$

The approach used in this paper is Altmeyer *et al.* combined multiset approach [3]. It is similar to the combined approach described above and a full derivation and descriptions is presented in [3]. This approach combines the UCB-Union multiset method with the ECB-Union multiset method. These methods build upon and remove some of the pessimism found in the ECB-Union and UCB-Union methods due to nested pre-emptions.

4. CACHE CONFLICTS AND IMPROVED LAYOUTS

In this section, we discuss cache conflicts, how they can be reduced by appropriate task layouts, and how the effectiveness of different layouts can be compared.

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.

4.1 Example Layouts

Figure 1 shows how five tasks ordered by priority could be laid out in cache. Task τ_1 has the highest priority, so its UCBs can never be evicted as it cannot be pre-empted. Task τ_2 and τ_3's UCBs are safe from eviction as they are not mapped to the same location in cache as higher priority task's ECBs. However, task τ_4's UCBs could be evicted by task τ_1, and τ_5's UCBs could be evicted by task τ_1, τ_2 or τ_4.

4.2 Comparing Layouts

The aim of this work 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 is implemented by controlling the linker.

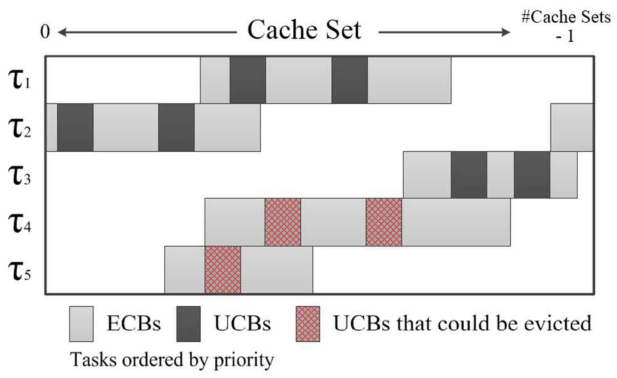

Figure 1 - Example layout showing how the position of tasks in cache affects whether their UCBs could be evicted during pre-emption

In order to evaluate different layouts for a taskset, a schedulability test can be used. A taskset has a fixed utilisation defined by the execution times and periods of the tasks, so 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* [19] 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. OPTIMISING TASK LAYOUT USING SIMULATED ANEALING

In this section we describe the changes that we made to the task layout during each iteration of the SA algorithm, the number of iterations that the algorithm when through before terminating, and the criteria that we used when deciding whether to accept a layout.

We used a SA algorithm to find improved task layouts as it allows a close to optimal solution to be found in a reasonable number of iterations. Given an initial layout, changes are made and evaluated over a number of iterations. In the initial layout the tasks are laid out one after another with no gaps in-between them. The tasks are in priority order, with the highest priority task first.

During each iteration of the SA algorithm, one of the following changes to the current layout is chosen at random and then evaluated:

- *Swap near* – swaps two neighbouring tasks by picking a random task *x* from tasks *1->X-1* where *X* is the number of task. This is based on the order of tasks in memory, rather than their priorities, i.e. task 1 is first in memory, followed by task 2. Once task *x* is picked, it is swapped with task *x+1*.
- *Swap far* – swaps 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* – 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, i.e. the gap is moved with the task.

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 and 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, it 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, $P_{accpt\ neg\ \Delta}$ is defined as:

$$P_{\text{accept neg }\Delta} = e^{\frac{\Delta\,BU}{temp}} \qquad (6)$$

5.1 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 for the experiment, then the new layout will be rejected and will not be evaluated by the schedulability test.

6. CASE STUDY

In this section we describe the results of a case study used to evaluate the task layouts produced by the SA algorithm. This case study is the same one used in Altmeyer *et al* [2] to evaluate methods for analysing CRPD in [2] and later used in [3] for the same purpose. The case study comprises a number of tasks from the Mälardalen benchmark suite[2] [17]. While these tasks do not represent a real taskset, they do represent typical code found in real-time systems. For each task, the derived WCET, ECBs and UCBs are taken from [1] and are shown in Table 1. The system was then setup to model an ARM7 processor[3] clocked at 10MHz 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.

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% (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 BU 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.

Table 1. WCET and number of UCBs and ECBs for a selection of tasks from the Mälardalen benchmark suite

	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

The results showing the breakdown utilisation for each layout are given in Table 2. Here, the layout obtained via SA provides a significant increase in the breakdown utilisation over that obtained by SeqPO. Trying 1000 random task orderings did give a slightly better result than the SA algorithm in this case, but at the expense of trying over twice as many layouts.

[2] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[3] http://www.arm.com/products/processors/classic/arm7/index.php

Table 2. Breakdown utilisation for the taskset in Table 1

	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	

The actual layout selected by the SA for the case study is shown in Figure 2. The tasks are ordered by priority.

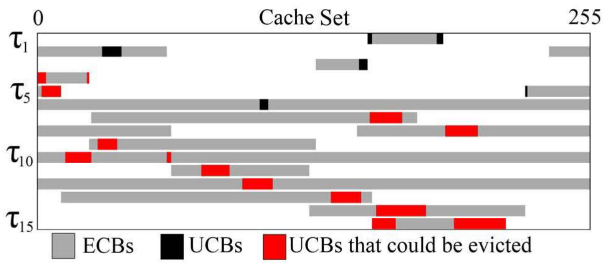

Figure 2. Layout of the tasks in cache chosen by the SA for the task set in Table 1

Note that because task τ_6 (loop3) is bigger than the number of cache sets, its ECBs can evict the UCBs of all the lower priority tasks so evictions were inevitable. Nevertheless, the SA algorithm still improved upon the SeqPO which is shown in Figure 3. The layout generated by the SA algorithm has a larger number of UCBs in conflict compared to the SeqPO layout; however, despite this it improves taskset schedulability. This is because of how the UCBs are organised. In the layout generated by the SA algorithm, the UCBs of lower priority tasks are evicted less often than they are in the SeqPO layout. This is due to the fact that high priority tasks, especially, tasks $\tau1$ to $\tau5$, have much shorter periods than the lowest priority tasks and therefore can pre-empt them many times.

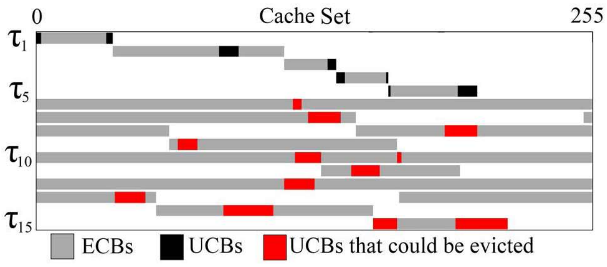

Figure 3. Layout of the tasks in using SeqPO for the task set in Table 1

Figure 4 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. It can be seen that the SA algorithm minimises the CRPD for the low priority tasks which are close to missing their deadlines, at the expense of the higher priority tasks which are not.

[4] The no pre-emption cost value differs from [2] due to a minor error in the computation of this value in [2]

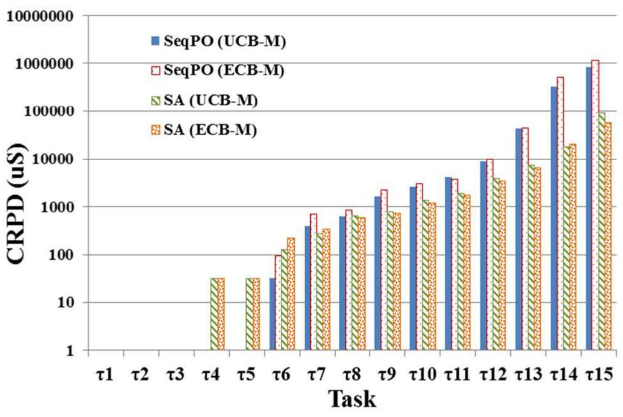

Figure 4. Graph showing the total CRPD/task for the task set in Table 1

7. EXPERIMENTAL EVALUTATION

In addition to the case study, in this section we describe the results of experiments aimed at evaluating 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.

In these experiments, the UUnifast algorithm [8] was used 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 as:

$$C_i = U_i T_i \tag{7}$$

As implicit deadlines were used, $D_i = T_i$.

UCBs were distributed through each task. Figure 5 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.

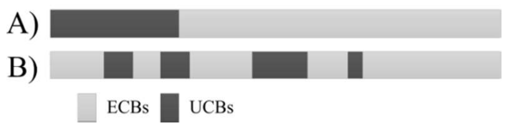

Figure 5. Two different distributions of UCBs throughout a task.

Distribution A is not representative of real code, therefore the majority of the experiments were done 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 experiment.

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 run again to generate the gaps between the groups of UCBs, at which point the UCBs were then laid out using a random starting position.

Finding an improved layout for a taskset with 10 tasks took roughly 1-2 minutes using a single thread on a processor running at 2.3GHz. As we wanted to evaluate our algorithm against a large number of tasksets, we split the tasksets up and ran them in parallel over four 8 core 2.3GHz processors.

7.1 Baseline Experiments

A number of experiments were run in order to investigate the quality of the task layouts produced by the SA for different cache and task configurations. These experiments 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 experiments:

- Allowed memory overhead was fixed to 0% (adding a random gap between tasks was not allowed)
- 10 tasks per taskset
- 1000 tasksets per experiment.
- 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 and therefore, 1000 random layouts were tried and averaged out. As the synthetically generated experiments 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 experiment investigates the quality of the task layouts produced by the SA algorithm compared to the other layouts. Figure 6 shows results for distribution B. This graphs shows the number of schedulable tasksets vs. utilisation for no pre-emption cost, SA, SeqPO, random and CS[i]=0. Note that the lines on the graphs are in the same order as they are in the key. The graphs are best viewed online in colour.

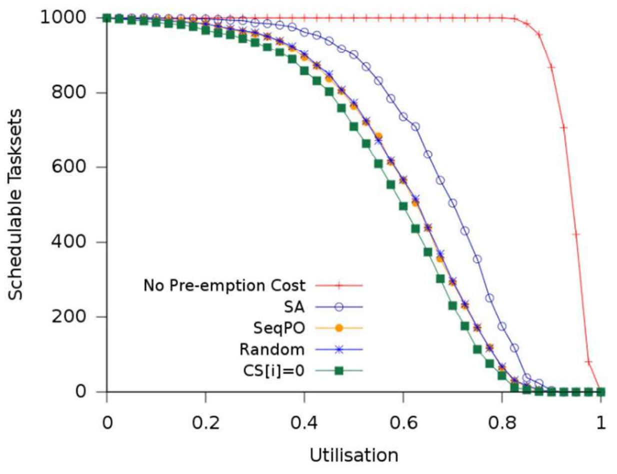

Figure 6. Schedulable tasksets vs Utilisation for UCB distribution B with a maximum of 5 groups of UCBs.

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. Table 3 shows the weighted schedulability measures, described next in subsection 7.2, for the baseline experiment 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 6.

Table 3. Weighted schedulability measures for the baseline experiments

	Distribution A	Distribution B
No pre-emption cost	0.859	
SA	0.665	
SeqPO	0.581	
Random	0.578	
CS[i]=0		

7.2 Weighted Schedulability

Evaluating all combinations of different task parameters is not possible. Therefore, the majority of our experiments focused on varying one parameter at a time. To present these results, weighted schedulability measures [6] 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 taskset utilisation is based on no pre-emption cost. 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:

$$W_l(p) = \left(\sum_{\forall\tau} u(\tau) \bullet S_l(\tau,p)\right) / \sum_{\forall T} u(\tau) \qquad (8)$$

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 being able to schedule higher utilisation tasksets.

7.3 Weighted Schedulability Experiments

For these weighted schedulability experiments, we used 100 tasksets, rather than 1000 tasksets at each utilisation level.

The second experiment varies the maximum number of UCB groups. As explained in section 7, the actual number of UCB groups is chosen at random between 1 and the maximum. Figure 7 show the impact on the schedulability of the tasksets. 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, and so the CRPD becomes dependent only on how the ECBs are laid out.

The third experiment investigates the effect the maximum UCB percentage has on schedulability. The maximum UCB percentage was varied from 0% to 100%, and the results are shown in Figure 8. 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, which leads to 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.

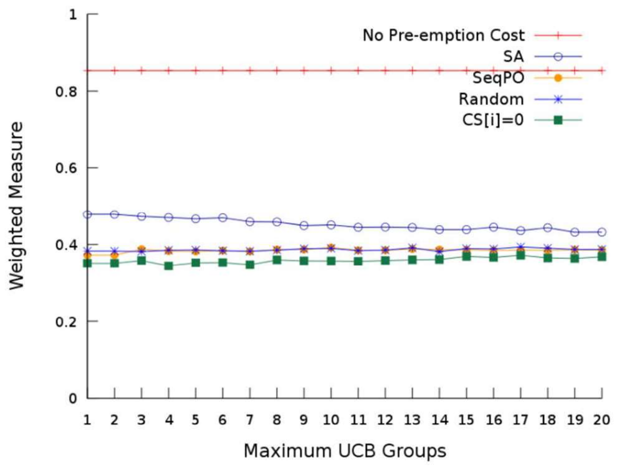

Figure 7. Varying the number of maximum number of UCB groups from 1 to 20

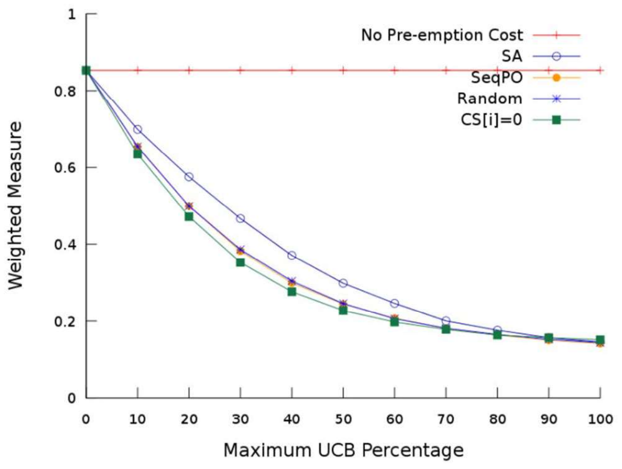

Figure 8. Varying the maximum UCB percentage from 0% to 100%

The fourth experiment investigates varying the cache utilisation, the results of which are shown in Figure 9. A cache utilisation of 1 represents all the tasks fitting into the cache, therefore any layout which does not include gaps between tasks is the best layout. This is why CS[i]=0 does not have the same weighted measure, as it introduces gaps. As the cache utilisation increases, the weighted measure decreases for all layouts with the layouts generated by the SA algorithm giving improved results up until a cache utilisation of 10.

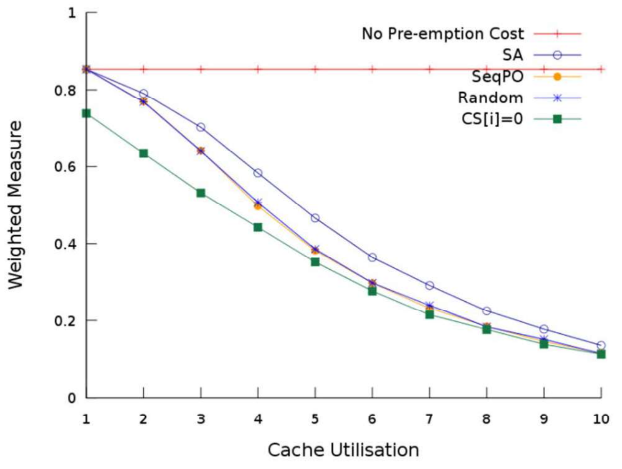

Figure 9. Varying the cache utilisation from 1 to 10

The fifth experiment investigates varying the number of cache sets, as shown in Figure 10. When varying the number of cache sets, the layouts generated by the SA algorithm performed well as the number of cache sets increased. 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.

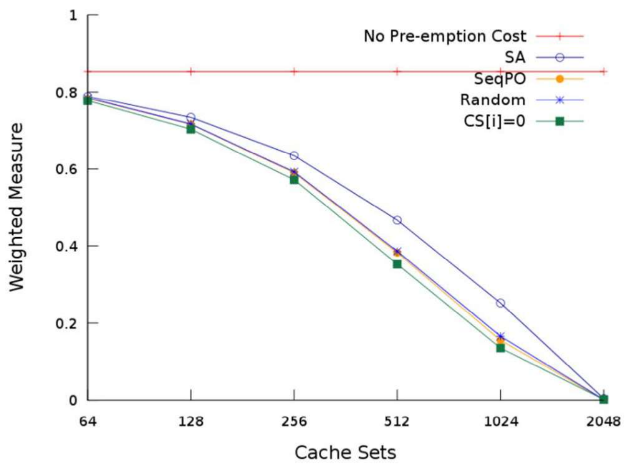

Figure 10. Varying the number of cache sets from 64 to 2048

The sixth experiment investigates the impact of the number of tasks on the schedulability of the taskset. The results can be seen in Figure 11. As the number of tasks increases, the number of schedulable tasksets decreases as expected because of the increased number of pre-emptions. After about 20 tasks, this started to level 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 is lining all of the tasks up on top of each other in the cache. The counter-intuitive result of the weighted measure levelling off for SA, SeqPO and random 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.

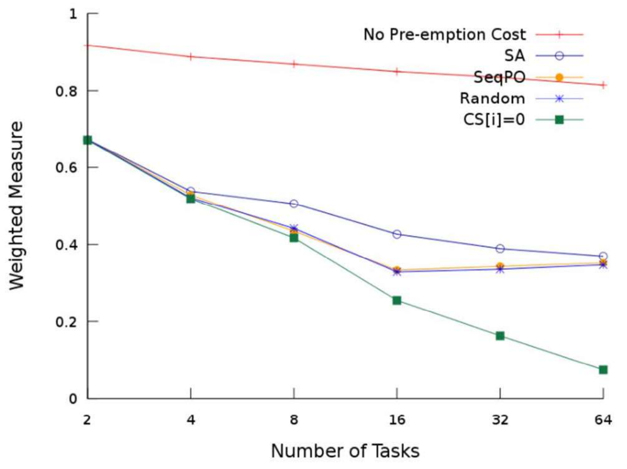

Figure 11. Varying the number of tasks from 2-64 in powers of 2

Finally, we investigated the distribution of CRPD per task for our default values for different layouts. We found that it followed a very similar pattern to the case study shown in section 6.

All of the experiments were run with three different memory restrictions on the SA algorithm, (0%, 10% and 100%), but have been presented with just 0%. This is because for the majority of our results, letting the SA algorithm add gaps between tasks had little effect. When changing the allowed memory overhead from

0% to 100%, the weighted measure for the baseline experiment 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.

We therefore decided to compare 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 tries. Trying every permutation results in 5040 (7!) different layouts, for 7 tasks, compared to 377 layouts[5] for the SA algorithm. While this is reasonable for 7 tasksets, as the number of tasks, n, increases it becomes increasingly prohibitive as there are n! different permutations of task ordering.

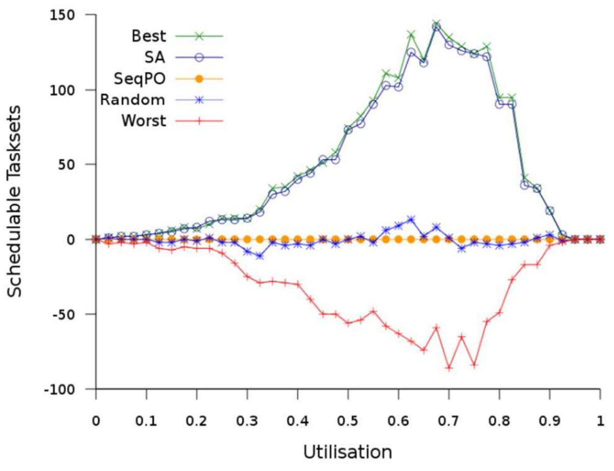

Figure 12. Comparing the SA algorithm at swapping tasks against a brute force approach of trying every permutation.

Figure 12 shows the results for 1000 tasksets normalised against the starting SeqPO layout. The graph shows that while the SA algorithm does not always find the best layout, it gets very close in significantly less time.

8. SUMMARY AND CONCLUSIONS

The major contribution of this paper is using CRPD aware schedulability analysis to drive a simple *simulated annealing* (SA) algorithm towards a layout that increases the schedulability of a taskset. 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 to layout tasks with not gaps in 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.

Through a number of experiments, we showed that our approach was able to find layouts that allowed the tasksets to be schedulable at a higher utilisation level than other layouts, specifically, the 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 6.

This work is useful for a number of reasons. 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.

The work presented assumed a direct mapped cache, future work could include extending it to N-way set associative caches with the LRU replacement policy. Our work was performed at the task level and an extension to it could be to split tasks up into procedures and perform the same layout optimisations on the individual procedures. This finer control of the code layout in memory should help to further reduce CRPD.

We would also like to perform a more comprehensive case study using real code from a multitasking application. This would allow us to evaluate the algorithm on more realistically positioned UCBs and ECBS. Finally, it would also be interesting to further investigate whether adding spaces between tasks gives any benefit for real code from large systems. If it can be proved that only swapping the tasks is enough, then future algorithms can be simplified to exploit that fact.

9. ACKNOWLEDGEMENTS

We would like to thank Antoine Colin for his help with the implementation and experiment design.

This work was partially funded by the UK EPSRC through the Engineering Doctorate Centre in Large-Scale Complex IT Systems (EP/F501374/1), the UK EPSRC funded Tempo project (EP/G055548/1), the Transregional Collaborative Research Center AVACS of the German Research Council (DFG), and the European Community's ARTEMIS Programme and UK Technology Strategy Board, under ARTEMIS grant agreement n° 295371-2 CRAFTERS.

10. REFERENCES

[1] Altmeyer, S. and Burguière, C. Cache-related Preemption Delay via Useful Cache Blocks: Survey and Redefinition. *Journal of Systems Architecture* (2010).

[2] Altmeyer, S., Davis, R.I., and Maiza, C. Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive

[5] See section 5 for an explanation of the SA algorithm, how many iterations it goes through, and why.

Systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)* (Vienna, Austria 2011), 261-271.

[3] Altmeyer, S., Davis, R.I., and Maiza, C. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *Real-Time Systems*, 48, 5 (September 2012), 499-512.

[4] Arnaud, A. and Puaut, I. Dynamic Instruction Cache Locking in Hard Real-Time Systems. In *The 14th International Conference on Real-Time and Network Systems* (2006).

[5] Audsley, N. C., Burns, A., Richardson, M., and Wellings, A.J. Applying new Scheduling Theory to Static Priority Preemptive Scheduling. *Software Engineering Journal*, 8, 5 (1993), 284-292.

[6] Bastoni, A., Brandenburg, B., and Anderson, J. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In *Proceedings of OSPERT* (Brussels, Belgum 2010), 33-44.

[7] Bertogna, M., Xhani, O., Marinoni, M., Esposito, F., and Buttazzo, G. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In *Proceedings of 23rd Euromicro Conference on Real-Time Systems (ECRTS)* (Porto, Portugal 2011), 217-227.

[8] Bini, E. and Buttazzo, G. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30, 1 (2005), 129-154.

[9] Burns, A. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In *Advances in Real-Time Systems*. 1994.

[10] Busquets-Mataix, J. V., Serrano, J. J., Ors, R., Gil, P., and Wellings, A. 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), 204-212.

[11] Campoy, A. M., Ivars, A. P., and Busquets-Mataix, J. V. Dynamic Use of Locking Caches in Multitask, Preemptive Real-Time Systems. In *Proceedings of the 15th Triennial World Congress of the International Federation of Automatic Control* (Barcelona 2002).

[12] Campoy, A. M., Ivars, A. P., and Busquets-Mataix, J. V. Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. In *Proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop* (2001).

[13] Campoy, A. M., Puaut, I., Ivars, A. P., and Busquets-Mataix, J. V. Cache Contents Selection for Statically-locked Instrction Caches: An Algorithm Comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)* (Palma de Mallorca, Balearic Islands, Spain 2005), 49-56.

[14] Davis, R. I., Zabos, A., and Burns, A. Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems. *IEEE Transactions on Computers*, 57, 9 (September 2008), 1261-1276.

[15] Falk, H. and Kotthaus, H. WCET-driven Cache-aware Code Positioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (Taipei, Taiwan 2011), 145-154.

[16] Gebhard, G. and Altmeyer, S. Optimal Task Placement to Improve Cache Performance. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT)* (Salzburg, Austria 2007), 259-268.

[17] Gustafsson, J., Betts, A., Ermedah, A., and Lisper, B. 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), 137-147.

[18] Lee, C., Hahn, J., Seo, Y., Min, S., Ha, H., Hong, S., Park, C., Lee, M., and Kim, C. Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling. *IEEE Transactions on Computers*, 47, 6 (June 1998), 700-713.

[19] Lehoczky, J. 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), 166-171.

[20] Liu, T., Li, M., and Xue, C.J. Instruction Cache Locking for Multi-task Real-Time Embedded Systems. *Real-Time Systems*, 48, 2 (2011), 166-197.

[21] Lokuciejewski, P., Falk, H., and Marwedel, P. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)* (Prague, Czech Republic 2008), 321-330.

[22] Puaut, I. and Decotigny, D. Low-complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)* (2002), 114-123.

[23] Tan, Y. and Mooney, V. Timing Analysis for Preemptive Multitasking Real-Time Systems with Caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6, 1 (February 2007).