Evaluation of Cache Partitioning for Hard Real-Time Systems

Sebastian Altmeyer* Roeland Douma*
*University of Amsterdam
Amsterdam, The Netherlands
{altmeyer, r.j.douma}@uva.nl

Will Lunniss† Robert I. Davis†
†University of York
York, UK
{wl510, rob.davis}@york.ac.uk

***Abstract*—In hard real-time systems, cache partitioning is often suggested as a means of increasing the predictability of caches in pre-emptively scheduled systems: when a task is assigned its own cache partition, inter-task cache eviction is avoided, and timing verification is reduced to the standard worst-case execution time (WCET) analysis used in non-pre-emptive systems. The downside of cache partitioning is the potential increase in execution times.**

In this paper, we evaluate cache partitioning for hard real-time systems in terms of overall schedulability. To this end, we examine the sensitivity of task execution times to the size of the cache partition allocated and present a cache partitioning algorithm that is optimal with respect to taskset schedulability. We then evaluate the performance of cache partitioning compared to state-of-the-art pre-emption cost analysis based on benchmark code and on a large number of synthetic tasksets. This allows us to derive general conclusions about the usability of cache partitioning and identify taskset and system parameters that influence the relative effectiveness of cache partitioning.

I. Introduction

Cache partitioning is often suggested as a means of increasing the predictability of caches in pre-emptively scheduled hard real-time systems. The rationale behind this argument is that when a task is assigned its own cache partition, inter-task cache eviction is avoided, and timing verification is reduced to the standard worst-case execution time (WCET) analysis used in non-pre-emptive systems. Cache partitioning comes at a cost. The reduced amount of cache available to each task potentially increases intra-task cache conflicts, trading an increase in (non-pre-emptive) execution times for reduced cache related pre-emption delays (CRPD).

Despite the wealth of publications on cache partitioning for real-time systems, little work has been done on the effectiveness of cache partitioning compared to systems where tasks make unconstrained use of the cache. Pre-emptive multi-tasking systems with unconstrained caches were considered unpredictable. Given recent advances in the analysis of cache related pre-emption delays, we consider this view outdated.

In this paper, we evaluate cache partitioning for hard real-time systems in terms of overall schedulability. To this end, we first determine the sensitivity of task execution times to the size of the available cache partition using application code from real-time benchmarks. Contrary to the implicit assumptions in prior work, the worst-case execution time of a task is not necessarily monotonic in the partition size. We show how the monotonicity property can be re-established using a monotonic upper bound function for the execution times. We then present a cache partitioning algorithm that aims at optimizing taskset schedulability. Under the assumption of monotonic execution times, the algorithm is optimal in the sense that it finds a schedulable cache partitioning whenever one exists. The algorithm is based on a branch-and-bound approach and is agnostic with respect to the schedulability test used, i.e., it is valid for any, *sustainable* schedulability test [6] and scheduling algorithm.

We evaluate the performance of cache partitioning vs. a non-partitioned cache, using state-of-the-art pre-emption cost aware schedulability analysis, based on two different benchmark sets (PapaBench and Mälardalen Benchmark Suite) and on a large number of synthetic tasksets. The evaluation using synthetic tasksets enables us to derive results that are valid in general, and not just for a small selection of use-cases. In addition, we identify how different parameter settings affect the relative performance of the partitioned vs. non-partitioned approaches. Finally, we quantify the error margin introduced by the assumption of monotonic execution times.

We focus on a completely analytical approach, where we compare the schedulability of real-time systems assuming fixed priority pre-emptive scheduling and a direct mapped cache. In both cases, partitioned and non-partitioned cache, we rely on bounds on the execution times obtained via WCET analysis, and in the non-partitioned case, also on analytical bounds on the CRPD. We concentrate on direct-mapped caches and fixed priority pre-emptive scheduling.

The paper is structured as follows: In Section II, we introduce the required terminology and notation. In Section III, we review existing approaches to cache partitioning. Section IV explains the sensitivity of the worst-case execution times of tasks with respect to the size of their allocated cache partitions. The optimal cache partitioning algorithm is presented in Section V, the results of the case study in Section VI and the evaluation based on synthetic tasksets in Section VI-C. Section VII concludes with a summary and discussion of future work.

II. System Model, Terminology and Notation

We consider the fixed priority scheduling of a set of sporadic tasks (or taskset) on a single processor. Each taskset Γ comprises n tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, where n is a positive integer. We assume that the index i of task τ_i represents its priority, hence τ_1 has the highest priority, and τ_n the lowest. We assume a discrete time model, where all task parameters are positive integers. We use the notation $hp(i)$ (and $lp(i)$) to mean the set of tasks with priorities higher than (and lower

than) i, and the notation $hep(i)$ (and $lep(i)$) to mean the set of tasks with priorities higher than or equal to (lower than or equal to) i.

Each task τ_i is characterized by its bounded worst-case execution time C_i obtained assuming no pre-emption (i.e. not including any cache related pre-emption delays), minimum inter-arrival time or *period* T_i, and relative *deadline* D_i. Each task τ_i therefore gives rise to a potentially unbounded sequence of invocations or *jobs*, each of which has an execution time upper bounded by C_i, an arrival time at least T_i after the arrival of its previous job, and an absolute deadline that is D_i after its arrival. Further, each task has a bounded release jitter of J_i, representing the maximum time from its notional arrival to it being ready to execute on the processor. In an *implicit-deadline* taskset, all tasks have $D_i = T_i$, in a *constrained-deadline* taskset, all tasks have $D_i \leq T_i$ while in an *arbitrary-deadline* taskset, task deadlines are independent of their periods. In this paper, we assume constrained deadline tasksets. The tasks are assumed to be independent and so cannot block each other from executing by accessing mutually exclusive shared resources, with the exception of the processor. (We note that this restriction is only made to simplify comparisons between the different approaches, resource sharing can be accounted for by schedulability analysis that incorporates CRPD as shown in [2, 3]).

The worst-case response time R_i of a task τ_i is given by the longest possible time from release of a job of the task until it completes execution. Thus task τ_i is schedulable if and only if $R_i \leq D_i - J_i$, and a taskset is schedulable if and only if all of its tasks are schedulable. The utilization U_i, of a task is given by its execution time divided by its period ($U_i = C_i/T_i$). The total utilization U of a taskset is the sum of the utilizations of all of its tasks.

A. Pre-emption Costs

We now extend the sporadic task model to include pre-emption costs. To this end, we need to explain how pre-emption costs can be derived. To simplify the following explanation and examples, we assume direct-mapped caches.

The additional execution time due to pre-emption is mainly caused by cache eviction: the pre-empting task evicts cache blocks of the pre-empted task that have to be reloaded after the pre-empted task resumes. The additional context switch costs due to the scheduler invocation and a possible pipeline-flush can be upper-bounded by a constant. We assume that these *constant* costs are already included in C_i. Hence, from here on, we use *pre-emption cost* to refer only to the cost of additional cache reloads due to pre-emption. This cache-related pre-emption delay (CRPD) is bounded by $g \times \mathrm{BRT}$ where g is an upper bound on the number of cache block reloads due to pre-emption and BRT is an upper-bound on the time necessary to reload a memory block in the cache (block reload time).

To analyse the effect of pre-emption on a pre-empted task, Lee et al. [19] introduced the concept of a useful cache block: A memory block m is called a useful cache block (*UCB*) at program point $\mathcal{P}$, if (i) m may be cached at $\mathcal{P}$ and (ii) m may be reused at program point $\mathcal{Q}$ that may be reached from $\mathcal{P}$ without eviction of m on this path. In the case of pre-emption at program point $\mathcal{P}$, only the memory blocks that (i) are cached and (ii) will be reused, may cause additional reloads. Hence, the number of UCBs at program point $\mathcal{P}$ gives an upper bound on the number of additional reloads due to a pre-emption at $\mathcal{P}$. The maximum possible pre-emption cost for a task is determined by the program point with the highest number of UCBs. Note that for each subsequent pre-emption, the program point with the next smaller number of UCBs can be considered. Thus, the j-th highest number of UCBs can be counted for the j-th pre-emption. A tighter definition is presented in [1]; however, in this paper we need only the basic concept.

The worst-case impact of a pre-empting task is given by the number of cache blocks that the task may evict during its execution. Recall that we consider direct-mapped caches: in this case, loading one block into the cache may result in the eviction of at most one cache block. A memory block accessed during the execution of a pre-empting task is referred to as an evicting cache block (*ECB*). Accessing an ECB may evict a cache block of a pre-empted task.

In this paper, we represent the sets of ECBs and UCBs as sets of integers with the following meaning:

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

$$s \in \mathrm{ECB}_i \Leftrightarrow \tau_i \text{ may evict a cache block in cache-set } s$$

Separate computation of the pre-emption cost is restricted to architectures without timing anomalies [20] but is independent of the type of cache used, i.e. data, instruction or unified cache.

In the case of set-associative LRU caches[1], a single cache-set may contain several useful cache blocks. For instance, $\mathrm{UCB}_1 = \{1, 2, 2, 2, 3, 4\}$ means that task τ_1 contains 3 UCBs in cache-set 2 and one UCB in each of the cache sets $1, 3$ and 4. As one ECB suffices to evict all UCBs of the same cache-set [10], multiple accesses to the same set by the pre-empting task does not need to appear in the set of ECBs. Hence, we keep the set of ECBs as used for direct-mapped caches. A bound on the CRPD in the case of LRU caches due to task τ_i directly pre-empting τ_j is thus given by the intersection $\mathrm{UCB}_j \cap' \mathrm{ECB}_i = \{m | m \in \mathrm{UCB}_j : m \in \mathrm{ECB}_i\}$, where the result is also a multiset that contains each element from UCB_j if it is also in ECB_i. A precise computation of the CRPD in the case of LRU caches is given in [4]. In this paper, we assume direct-mapped caches. Note that all equations provided within this paper are for direct-mapped caches, they are also valid for set-associative LRU caches with the above adaptation to the set-intersection.

B. Schedulability Test

We now recapitulate the exact (sufficient and necessary) schedulability test for fixed priority pre-emptive scheduling of constrained-deadline tasksets based on *response time analysis* [5, 17, 13]. Subsequent work on integrating cache related pre-emption delays into schedulability analysis for fixed priority pre-emptive systems is based on this analysis. The basic form given below assumes that pre-emption costs are zero.

The response time R_i of a task necessarily contains its execution time C_i, and in addition, τ_i may suffer interference and be pre-empted by tasks with higher priority than i. Let τ_j be such a task. Within the response time R_i of τ_i, task τ_j executes at most $\left\lceil \frac{R_i + J_j}{T_j} \right\rceil$ times, each time for at most C_j. Hence,

[1] The concept of UCBs and ECBs cannot be applied to FIFO or PLRU replacement policies as shown in [10].

the response time R_i of task τ_i is given by:

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

where $\mathrm{hp}(i)$ denotes the set of tasks with higher priority than i. The response time R_i of task τ_i appears on both the left-hand side and the right-hand side of (1). As the right-hand side is a monotonically non-decreasing function of R_i, then a solution may be found via fixed-point iteration:

$$R_i^{x+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{x} + J_j}{T_j} \right\rceil C_j \tag{2}$$

Iteration starts with an initial value, typically $R_i^0 = C_i$, and ends when either $R_i^{x+1} > D_i - J_i$ in which case the task is unschedulable, or when $R_i^{x+1} = R_i^x$, in which case the task is schedulable, with a worst-case response time R_i^{x+1}. We note that convergence may be speeded up using the techniques described in [13].

A schedulability test is *sustainable* [6] if any taskset that was deemed schedulable by the test remains so if the parameters "improve", e.g., if the runtimes decrease or periods increase. Note that response time analysis as well as the standard schedulability tests for systems with dynamic priorities (e.g. EDF scheduling) are sustainable.

C. Pre-emption Cost aware Schedulability Test

To integrate pre-emption costs into response time analysis, Busquets et al. [11] extended (1) by adding a term $\gamma_{i,j}$ representing the pre-emption cost of a job of task τ_j executing during the response time of task τ_i (with $j \in hp(i)$):

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j + \gamma_{i,j}) \tag{3}$$

An alternative approach was taken by Petters et al. [26] and later Staschulat et al. [29], who based their analyses on the following equation:

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left(\left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j + \hat{\gamma}_{i,j} \right) \tag{4}$$

The value $\hat{\gamma}_{i,j}$ denotes the pre-emption cost of *all* jobs of task τ_j executing during the response time of task τ_i (again with $j \in hp(i)$). It is given by the $\left\lceil \frac{R_i+J_j}{T_j} \right\rceil$-highest pre-emption costs of a job of task τ_j executing during R_i. Although the difference with respect to (3) is subtle, more precise analysis can be obtained by using $\hat{\gamma}_{i,j}$ as a bound on the *overall impact* of all jobs of τ_j on the response time R_i instead of a bound on the impact of *just one* job of τ_j.

We note that when pre-emption costs are considered explicitly, the worst-case scenario is not necessarily given by a synchronous release of all higher priority tasks [23] and hence (3) and (4) provide sufficient, but not exact schedulability tests.

1) Pre-emption Cost Computation: The value $\gamma_{i,j}$ can be computed in a number of different ways, which are described in detail in [3], here, we restrict our explanations to the two dominant approaches: ECB-Union and UCB-Union.

a) UCB-Union: Tan and Mooney [30] analysed the pre-emption cost via an upper bound on the number of useful cache blocks (of all pre-empted tasks) that a pre-empting task τ_j may evict. As it is only the eviction of useful cache blocks belonging to tasks with equal or higher priority than task τ_i that may increase the response time of task τ_i, only tasks with intermediate priorities in the set $\mathrm{aff}(i, j) = \mathrm{hep}(i) \cap lp(j)$, need be considered.

$$\gamma_{i,j}^{\text{UCB-U}} = \mathrm{BRT} \cdot \left| \left(\bigcup_{k \in \mathrm{aff}(i,j)} \mathrm{UCB}_k \right) \cap \mathrm{ECB}_j \right| \tag{5}$$

Here, $\gamma_{i,j}^{\text{UCB-U}}$ represents the worst-case impact a job of task τ_j can have on all (useful cache blocks of) tasks with lower priority than task τ_j down to task τ_i. We refer to this approach as UCB-Union.

b) ECB-Union: Instead of considering the precise set of ECBs of a pre-empting task and bounding all possibly affected UCBs (as UCB-Union does), ECB-Union [2, 3] considers the precise number of UCBs of the pre-empted task. It then assumes that the pre-empting task τ_j has itself already been pre-empted by *all* tasks with higher priority. This nested pre-emption of the pre-empting task is represented by the union of the ECBs of all tasks with higher or equal priority than task τ_j:

$$\gamma_{i,j}^{\text{ECB-U}} = \max_{\forall k \in \mathrm{aff}(i,j)} \left\{ \left| \mathrm{UCB}_k \cap \left(\bigcup_{h \in \mathrm{hep}(j)} \mathrm{ECB}_h \right) \right| \right\} \tag{6}$$

The UCB-Union and ECB-Union approaches are *incomparable* in that there are tasks that may be deemed schedulable using one approach but not the other and vice-versa. More precise analysis can therefore be achieved by using a combination of both approaches, as described in [2]. As schedulability tests based on UCB-Union and ECB-Union are sufficient, then if either test deems a task schedulable then it is proven schedulable, even if it is not deemed schedulable according to the other test.

2) Optimal Task Layout: The precise cache mapping, i.e., the mapping of memory block to cache sets strongly influences the pre-emption costs. Consider for instance the extreme situation where all tasks are aligned to the first cache-set: Each task will definitely evict cache blocks of another task. If tasks' code is instead aligned sequentially in the cache, the pre-emption costs are very likely to be smaller. In [21], Lunniss et al. showed how to optimize the task layout with respect to the taskset schedulability and the pre-emption costs. The technique used determines the order in which the code for each task is placed sequentially in memory, without leaving any gaps. Thus unlike cache partitioning, optimizing the task layout does not require any severe changes to the system. It only affects the position of the code and data in the binary, hence an appropriate layout can only improve performance.

III. Review of Cache Partitioning For Real-Time Systems

Cache partitioning [24, 27] is a technique to reduce or even completely avoid cache-related pre-emption delays, aimed at increasing the predictability of real-time systems. Cache partitioning trades *inter-task* for *intra-task* cache conflicts, i.e. it trades off reduced cache-related pre-emption delays against potentially increased worst-case execution times. Partitioning techniques can be implemented either in hardware [18] or in software [24, 27]. Mueller et al. [24] and later Plazar et

al. [27] proposed a partitioning-aware compiler, asserting that each task only accesses its own cache partition. This comes at the cost of often substantial changes to the code and data layout, which further increases task execution times; however, as no additional hardware is needed, the memory access delays remain unchanged. This is in contrast to hardware-based solutions where an additional mapping layer from code/data to main memory is needed.

Despite the wealth of publications on cache partitioning for real-time systems, little work has been done on evaluating the effects of cache partitioning, and in particular, its effectiveness compared to systems where tasks make unconstrained use of the cache. The previously cited papers either focus on the implementation of cache partitioning [24, 27, 28], or compare partitioned systems with systems without cache [31]. The rationale behind this limited evaluation is the belief that pre-emptive systems that make unconstrained use of cache are unpredictable. Given recent advances in the analysis of cache related pre-emption delays, this view can now be considered somewhat outdated.

Studies on general usability of cache partitioning have been conducted by Busquets-Mataix and Wellings [12] (to a limited extent), and more recently by Bui et al. [9]. Busquets-Mataix and Wellings based their evaluation on simplistic models of task execution times and pre-emption costs. The execution time variation was modelled according to [16], favouring efficiency over precision, and only delivers rough estimates. The authors also assume that each evicting cache block causes an additional pre-emption cost, which is a very pessimistic assumption [3].

Bui et al. [9] based their evaluation on high-level execution time models [32] and simulation. This is in stark contrast to our work. We rely on the results of static timing analysis (both for the WCET bounds and the pre-emption costs) as used in safety-critical hard real-time systems.

Since finding an optimal cache partitioning is NP-hard [9], previous approaches employed heuristics either to minimize the number of cache misses, or to minimize the utilization [18, 12, 9, 27].

The research that we present in this paper differs in the following aspects: As schedulability is the key criterion in verifying the temporal correctness of hard real-time systems, we focus on taskset schedulability as opposed to utilization. A cache partitioning may be schedulable even though the task utilization is not the minimum that could be obtained. Similarly, minimizing the utilization does not necessarily optimize schedulability. We present a partitioning algorithm which is optimal under the assumption that the worst-case execution time of each task is monotonic in the size of the partition allocated to that task. We aim at deriving general statements about the usability and efficiency of cache partitioning compared to a non-partitioned cache analysed using state-of-the-art pre-emption cost analyses.

IV. Partition-Size Sensitivity

In this section, we evaluate the sensitivity of the worst-case execution times of tasks with respect to the size of their allocated cache partitions. The aim of this sensitivity analysis is to form simple yet accurate execution time functions that are parametric in the size of the cache partition allocated to the task. These functions provide the information required by the optimal partitioning algorithm described in Section V.

We perform sensitivity analysis by computing WCET bounds for varying cache partition sizes using static analysis. Based on these values, we can deduce typical variations in execution time depending on the code size of the task and the size of the cache partition allocated to it. The rationale behind this empirical evaluation is twofold: First, we are interested in the behaviour of a set of real examples, and second, we want to use realistic models of execution-time as a function of cache partition size to determine an effective partitioning of the cache between tasks. We note that with hardware support for cache partitioning, partitions are typically restricted to being a power of 2 in size e.g. 8,16,32 cache sets etc.; whereas software methods [27] can support cache partitions of any arbitrary number of sets. In the remainder of the paper, we assume that the number of cache sets in a partition may take any arbitrary value; however, we note that the techniques introduced are easily adapted to the case were partition sizes come from a restricted set of hardware-supported values.

The target architecture is an ARM7 processor[2] with direct-mapped cache of size 4kB with a line size of 16 Bytes (and thus, 256 cache sets), a block reload time of $8\mu s$ and a clock rate of 100 MHz. As benchmarks, we used PapaBench [25] and the Mälardalen benchmark suite [15]. We used the aiT Timing analyzer [14] to compute WCET bounds, and evaluate the sensitivity of execution time with respect to cache partition size.

Figures 1 and 2 show the normalized WCET bounds for the benchmark tasks with varying cache partition sizes and cache types. A perfect data (or instruction) cache means that all data (or instruction) accesses are served instantaneously. Even though this assumption is unrealistic, it removes possible noise and and allows us to fully concentrate on the effects of pre-emption and partitioning. The bounds are only shown for a selection of the benchmark tasks to avoid cluttering the graphs, (which are best viewed online in colour). We have also performed experiments with *instruction cache but without data cache* and also with *data cache but without instruction cache*. The results are very similar to the evaluation shown for perfect caches, but less accentuated.

In figures 1 and 2, each line denotes the execution time for one benchmark. The y-axis depicts the normalized execution time with the value 1 representing the largest WCET bound (which typically corresponds to the smallest cache partition size i.e. zero). The x-axis depicts the normalized cache partition size with the value 1 representing the code-size/maximum memory usage of the task. Increasing the size of the cache partition beyond the code size/memory footprint does not improve the execution time any further. We can see that variation in the execution times is stronger in the case of instruction cache compared to data cache. This behaviour is as expected since each instruction results in an instruction cache access, but not necessarily in a data cache access. Similarly, the variation in the execution times is amplified by the assumption of a perfect data/instruction cache.

Note we do not assume any implementation cost for cache partitioning. In reality, the additional hardware or software support needed to limit accesses to the allocated cache partition would affect task execution times. Hardware supported cache partitioning requires additional circuits that may increase the

[2] http://www.arm.com/products/processors/classic/arm7

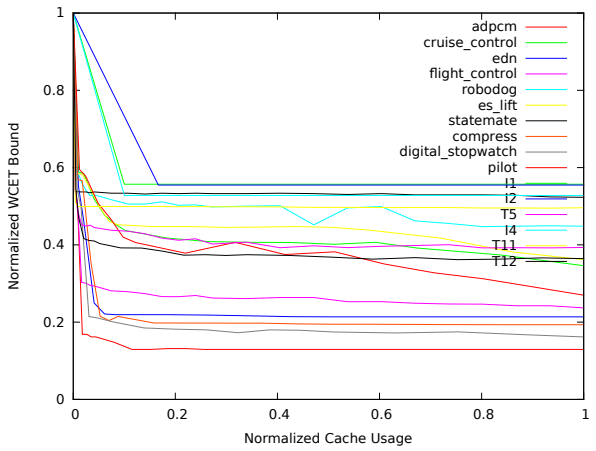


Fig. 1: WCETs depending on the cache partition size (direct mapped instruction cache, perfect data cache) for a representative selection of tasks from Table III and Table I.

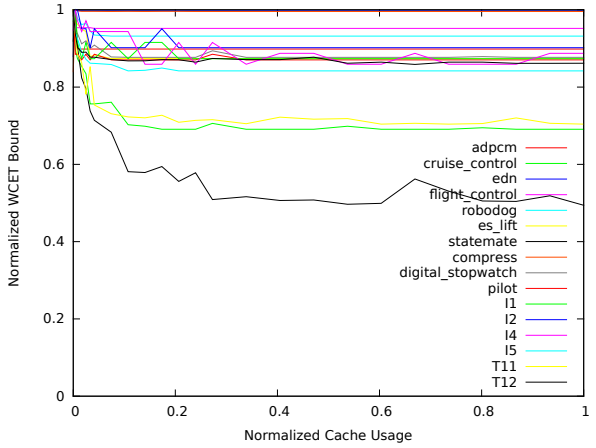


Fig. 2: WCETs depending on the cache partition size (direct mapped data cache, perfect instruction cache) for a representative selection of tasks from Table IV and Table II.

critical path or reduce the available cache size [18]. Software supported cache partitioning requires additional instructions that may increase the code-size and the execution time [27]. Ignoring these costs slightly favours cache partitioning over sharing the cache which incurs no such costs.

Monotonicity

We observe from Figures 1 and 2 that the execution time bounds are not necessarily monotonic with respect to the cache partition size.

This counter-intuitive behaviour can be explained by differences in the mapping of memory blocks to the cache sets. Assuming a direct-mapped cache with a line size of 16 bytes and a task that exhibits the following access sequence

$$0x00030 \rightarrow 0x00080 \rightarrow 0x00030$$

If we assign this task a cache partition of size 4, memory block $0x00030$ maps to set 3 of this partition and $0x00080$ maps to set 0. The last access to $0x00030$ therefore results in a cache

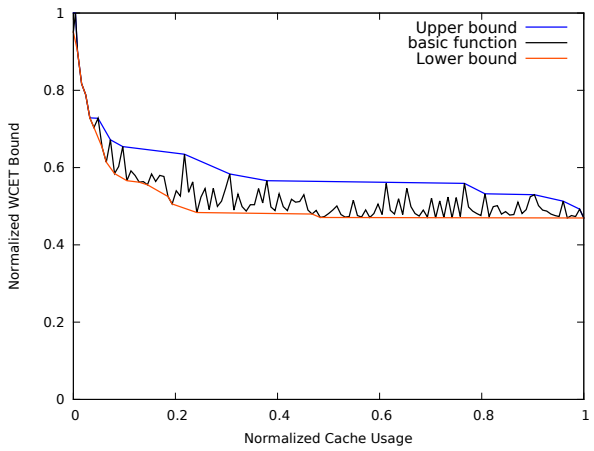


Fig. 3: Over-/Underapproximations of the WCET function.

hit. In contrast, in a larger cache partition of size 5, memory blocks $0x00030$ and $0x00080$ both map to cache set 3 and the last access to $0x00030$ is a cache miss. Hence, for this trivial example, the performance with 5 cache sets is worse than that for 4 cache sets.

We note that the assumption of monotonic execution time bounds is both common and often not explicitly stated in work on cache partitioning for real-time systems [9, 12, 18, 24, 27].

The impact of these effects is however limited, and so we can replace the actual execution time function with monotonic over/under-approximations without significant loss of precision, as shown in Figure 3. Here, the basic function (black line) is non-monotonic, while the upper bound (blue line) and the lower bound (red line) are monotonically non-increasing functions of cache partition size. We thus establish monotonicity of the WCET with respect to the cache partition size and can use this property in our approach to partitioning the cache. In Section VI-D, we quantify the error introduced by this approximation.

V. Optimal Cache Partitioning

In this section, we derive an optimal cache partitioning algorithm, which makes use of the monotonic upper bound execution time functions of cache partition size described in the previous section. We assume a direct-mapped cache of size S. A cache partitioning P is a tuple of non-negative integers describing for each task τ_i, the size p_i of its allocated cache partition:

$$P = (p_1, p_2 \ldots, p_n) : \underbrace{\mathbb{N} \times \ldots \times \mathbb{N}}_{n} \tag{7}$$

We assume that each task has a dedicated cache partition which is not shared with any other tasks (we return to this point in Section VII). A cache partitioning is valid, if the total size of the cache partitions does not exceed the overall size S of the cache (i.e. if $\sum_i p_i \leq S$).

We are interested in the schedulability of a taskset, as this is the main optimization criterion for hard real-time systems. We therefore say that a cache partitioning algorithm is optimal, iff it finds a cache partitioning whereby the tasks are schedulable, whenever such a partitioning exists. Note that this is different from minimizing the utilization of a taskset, since taskset

utilization is only a rough indicator of system schedulability.

To compute an optimal cache partitioning, we use a branch-and-bound approach (see Algorithm 1) which is certain, under the assumption of monotonic execution time functions, to find a feasible cache partitioning if one exists. To this end, we exploit the sustainability of the schedulability test with respect to execution times and the monotonicity of the execution time function with respect to the cache partition size to prune the search space.

The algorithm is implemented using a recursive function *checkPartition*. This function takes as its input the current task index i, a partially defined partitioning P and the remaining cache size s. The partitioning is defined up to index i and the remaining cache size s is given by S minus the sum of the sizes of the first i partitions i.e. $s = S - \sum_{j=1}^{i} p_i$.

The initial input to the function is the first task index 1, an arbitrary partitioning P and the overall cache size S. If the last task index is reached, the partitioning is fully defined and the result is determined by the function *isSchedulable*, which checks the schedulability of the taskset for the defined partitioning. Note, here we employ the basic schedulability test without pre-emption costs (see Section II-B) given by (1), as the cache partitioning prevents any cache-related pre-emption delays.

Algorithm 1 Optimal Cache Partitioning (Schedulability)

```
1: function checkPartition(int i, partition P, int s)
2:     if i = n then
3:         return isSchedulable(P)
4:     end if
5:     let P = (p_1, .., p_n)
6:     if not isSchedulable((p_1, .., p_{i-1}, s, .., s)) then
7:         return false
8:     end if
9:     if isSchedulable((p_1, .., p_{i-1}, ⌊s/(n-i+1)⌋, .., ⌊s/(n-i+1)⌋)) then
10:        return true
11:    end if
12:    p_i = 0
13:    while p_i ≤ s do
14:        if checkPartition(i + 1, (p_1, .., p_i, .., p_n), s − p_i) then
15:            return true
16:        else
17:            p_i = nextStep(i,p_i)
18:        end if
19:    end while
20:    return false
21: end function
```

In the next step, the algorithm checks taskset schedulability under (a) the optimistic assumption that each not yet specified task partition is of size s and (b) under the pessimistic assumption that each not yet specified task partition is given an equal share of the remaining cache size, i.e., $\lfloor s/(n-i+1) \rfloor$. This enables effective pruning of the search in the case where (a) schedulability is disproved for any extensions to the current partial partitioning, and early exit in the case (b) schedulability is proven assuming that all further tasks are schedulable with a cache partition of equal size.

The last construct of the algorithm, the while loop, implements the branching. The partition size of cache partition p_i is varied from 0 up to the remaining cache size s and each possible partitioning is evaluated using a recursive function call. This is done using the function *nextStep* which computes the next partition size for task τ_i. Due to the monotonicity of the execution time functions with respect to cache partition size, *nextStep* jumps directly to the next partition size where the execution time changes. All intermediate partition sizes with the same execution time can be safely ignored. In the worst-case, up to n^S different cache partitionings must be evaluated, where n is the number of tasks and S the number of cache sets. In practice, the runtime is substantially lower due to early exits and the reduced number of partition sizes which give different execution times. We return to this point in the following section. Further, in the case where hardware support is provided for a limited number of partition sizes, the runtime is further reduced due to the restricted number of partition sizes supported.

VI. Case Study

In this section, we evaluate the partitioning algorithm based on PapaBench, the Mälardalen benchmark suite and a set of SCADE[3] tasks (partially provided by SCADE, partially from our own SCADE models). Besides the effectiveness of the cache partitioning algorithm, we are interested in the precision of the simplified execution time model and the runtime performance of the algorithm.

For the case study, the target architecture is an ARM7 processor (with a 4kB direct-mapped cache, line size of 16 Bytes, 256 cache sets, block reload time $8\mu s$, clock rate of 100 MHz). The execution times bounds were derived using the aiT Timing analyzer [14].

Papabench provides two different tasksets (*fbw* and *autopilot*) with deadlines and periods (see Table I and II). With the initial processor frequency of 100MHz, both tasksets are schedulable both with and without cache partitioning. The other benchmarks only provide code and do not form a meaningful taskset. We therefore randomly selected tasks from (i) Table I and II, and (ii) Table III and IV (together with execution times, the execution time variations, codes size and UCBs/ECBs).

The remaining task and taskset parameters used in our experiments were randomly generated as follows:

- The default taskset size was 10.
- Task utilizations were generated using the UUni-fast [8] algorithm.
- Task periods were set based on the utilization and execution times: $C_i = U_i \cdot T_i$.
- Task deadlines were implicit[4], i.e., $D_i = T_i$.
- Priorities were assigned in Deadline Monotonic order.
- The jitter of all tasks was assumed to be zero, i.e. $J_i = 0$.

In each experiment the taskset utilization not including pre-emption cost was varied from 0.025 to 0.975 in steps of 0.025. For each utilization value, 1000 tasksets were generated and the schedulability of those tasksets was determined using the cache partitioning algorithm or pre-emption cost aware

[3]Esterel SCADE http://www.esterel-technologies.com/

[4]Evaluation for constrained deadlines, i.e., $D_i \in [2C_i; T_I]$ gave broadly similar results although fewer tasksets were deemed schedulable.

TABLE I: Execution times and number of UCBs and ECBs for the PapaBench Benchmarks. Instruction cache with perfect data cache ($WCET^1$) and without data cache ($WCET^2$)

	Description	UCBs	ECBs	WCET[1]	WCET[2]	Period
I4	interrupt-modem	2	10	$303\mu s$	$520\mu s$	-
I5	interrupt-spi-1	1	10	$251\mu s$	$447\mu s$	-
I6	interrupt-spi-2	1	4	$151\mu s$	$228\mu s$	-
I7	interrupt-gps	3	26	$283\mu s$	$493\mu s$	-
T5	altitude-control	20	66	$1478\mu s$	$1660\mu s$	$250ms$
T6	climb-control	1	210	$5429\mu s$	$6241\mu s$	$250ms$
T7	link-fbw-send	1	10	$233\mu s$	$471\mu s$	$250ms$
T8	navigation	1	256	$44,42ms$	$54,35ms$	$50ms$
T9	radio-control	0	256	$15,6ms$	$21,1ms$	$50ms$
T10	receive-gps-data	22	194	$5987\mu s$	$6659\mu s$	$25ms$
T11	reporting	2	256	$12,22ms$	$5ms$	$100ms$
T12	stabilization	11	194	$5681\mu s$	$6654\mu s$	$50ms$

TABLE II: Data cache with perfect instruction cache ($WCET^1$) and without instruction cache ($WCET^2$)

	Description	UCBs	ECBs	WCET[1]	WCET[2]	Period
I4	interrupt-modem	3	10	$335\mu s$	$790\mu s$	-
I5	interrupt-spi-1	2	10	$287\mu s$	$644\mu s$	-
I6	interrupt-spi-2	1	4	$135\mu s$	$338\mu s$	-
I7	interrupt-gps	3	26	$278\mu s$	$712\mu s$	-
T5	altitude-control	2	66	$654\mu s$	$3860\mu s$	$250ms$
T6	climb-control	5	210	$2375\mu s$	$14,21mss$	$250ms$
T7	link-fbw-send	2	10	$298\mu s$	$634\mu s$	$250ms$
T8	navigation	10	256	$23,38ms$	$138ms$	$50ms$
T9	radio-control	14	256	$10,2ms$	$51ms$	$50ms$
T10	receive-gps-data	4	194	$3058\mu s$	$20,5mss$	$25ms$
T11	reporting	6	242	$12,8ms$	$32ms$	$100ms$
T12	stabilization	6	194	$2711\mu s$	$16,1mss$	$50ms$

response time analysis with either sequential or optimal task layout [21]. We thus compared the results for cache partitioning against those for (i) no partitioning with a sequential task layout, (ii) no partitioning with an optimized task layout, (iii) analysis ignoring pre-emption costs, but assuming that all the tasks shared the cache; (iv) naive cache partitioning with all tasks allocated the same size partition S/n; (v) no cache. The sequential task layout reflects the basic un-optimized cache mapping, i.e., where the code for each task is placed consecutively in memory.

Despite the fact that the worst-case behaviour of the cache partitioning algorithm is exponential, we were able to compute the schedulability of the $42,000$ tasksets of both case studies in less than 10 minutes on a 2.6-GHz Quadcore processor.

A. PapaBench

Most tasks from Table I and II have rather short execution times, leading to relatively high pre-emption costs. These tasks are simple, short control tasks with limited computations and data accesses. Figures 4a and 4c show the *success ratio*; the number of tasksets based on Papabench that were schedulable at the various levels of utilization. In the case of instruction caches (Figure 4b), optimal partitioning has similar performance to sequential task layout with no partitioning, while optimal task layout with no partitioning results in improved performance. Optimal cache partitioning was only able to improve performance over sequential task layout with no partitioning in a few cases. In the case of data caches (Figure 4d), optimal partitioning outperforms optimal task layout with no partitioning. The variation of the execution times in this

TABLE III: Mälardalen Benchmark Suite (M) and SCADE Benchmarks (S). Instruction cache with perfect data cache ($WCET^1$) and without data cache ($WCET^2$)

	Description	UCBs	ECBs	WCET[1]	WCET[2]
M	adpcm	24	226	$5,541s$	$6,521s$
M	compress	25	114	$3,664s$	$8,426s$
M	edn	56	98	$244,8ms$	$458,2ms$
M	fir	28	50	$21,52ms$	$497ms$
M	jfdctinit	40	162	$13,89ms$	$32,98ms$
M	ns	17	26	$73,38ms$	$168ms$
M	nsichneu	53	256	$77,96ms$	$163ms$
M	statemate	3	256	$9,757s$	$20,07s$
S	cruise control system	25	107	$1,959s$	$3,548s$
S	flight control system	70	256	$2,138s$	$4,083s$
S	navigation system	45	82	$1,409s$	$3,712s$
S	stopwatch	58	130	$3,786s$	$5,533s$
S	elevator simulation	40	114	$1,586s$	$2,917s$
S	robotics systems	68	256	$4,311s$	$6,377s$

TABLE IV: Data cache with perfect instruction cache ($WCET^1$) and without instruction cache ($WCET^2$)

	Description	UCBs	ECBs	WCET[1]	WCET[2]
M	adpcm	7	242	$5,856s$	$43,17s$
M	compress	6	242	$9,740s$	$25,26s$
M	edn	5	98	$518,9ms$	$1,422s$
M	fir	5	50	$42,65ms$	$121ms$
M	jfdctinit	8	242	$23,2ms$	$73,63ms$
M	ns	3	26	$133,7ms$	$466,9ms$
M	nsichneu	8	242	$66,74ms$	$178,3ms$
M	statemate	30	242	$8,143s$	$22,45s$
S	cruise control system	15	98	$1,77s$	$6,207s$
S	flight control system	12	242	$3,24s$	$11,02s$
S	navigation system	3	82	$2,96s$	$7,566s$
S	stopwatch	9	130	$4,417s$	$25,03s$
S	elevator simulation	4	114	$1,863s$	$5,432s$
S	robotics systems	5	242	$3,427s$	$22,45s$

case is rather low, while the number of UCBs is comparably high. We thus note that the two approaches are *incomparable*.

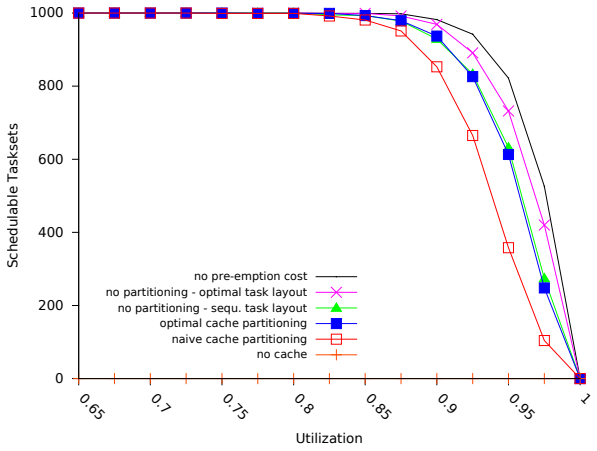
Almost no tasksets were schedulable with no cache, except for the case of data cache with perfect instruction cache as the impact of the data cache alone is limited.

B. Mälardalen and SCADE Benchmarks

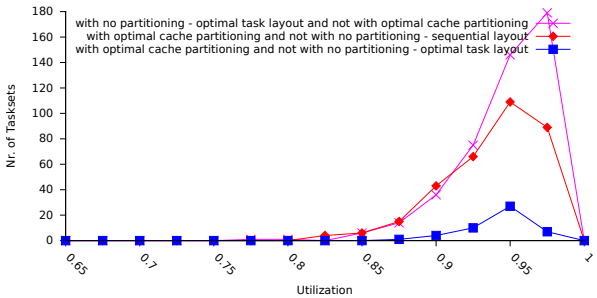
In contrast to the first case study, the execution times of the tasks from Table III and Table IV for the Mälardalen and SCADE Benchmarks are comparably high, and thus the pre-emption costs relatively low. These tasks exhibit a low locality of memory accesses but high amounts of computation. In this case, the low cache related pre-emption delays result in significantly better performance if the cache is not partitioned. Here, cache partitioning was unable to improve performance over the simple sequence task layout with no partitioning, as illustrated in Figure 5a. Note that in this case there are no major differences for data and instruction caches, the results of the different approaches are just more (instruction caches) or less (data caches) accentuated.

C. Synthetic Tasksets

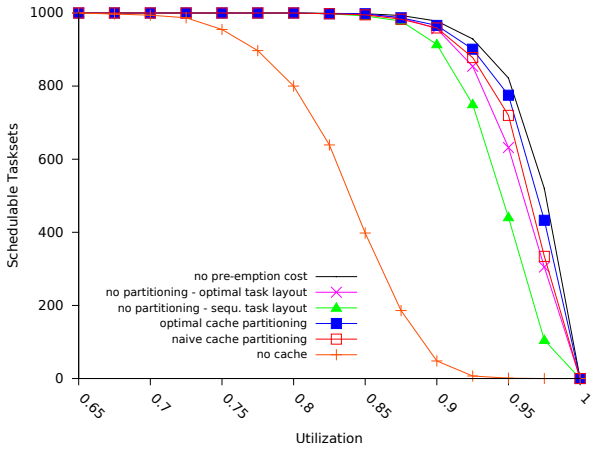
We also evaluated the effectiveness of cache partitioning on a large number of synthetic tasksets with varying cache configurations and varying task parameters. Our aim here was to identify those parameters that have a significant influence on the relative effectiveness of cache partitioning versus a non-



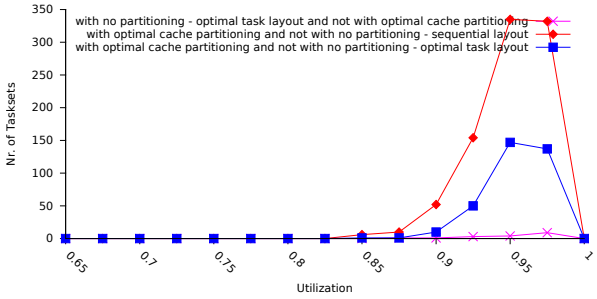
(a) Number of tasksets deemed schedulable at the different total utilizations (instruction cache with perfect data cache).



(b) Number of tasksets deemed schedulable with one approach and not another (instruction cache with perfect data cache).

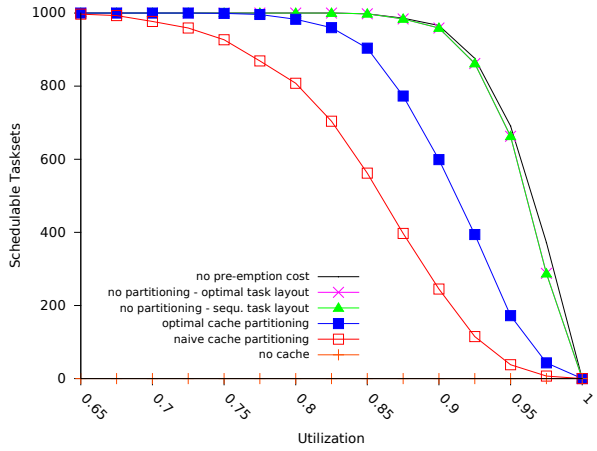


(c) Number of tasksets deemed schedulable at the different total utilizations (data cache with perfect instruction cache).

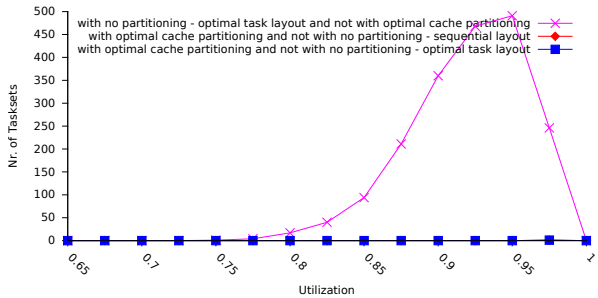


(d) Number of tasksets deemed schedulable with one approach and not another (data cache with perfect instruction cache).

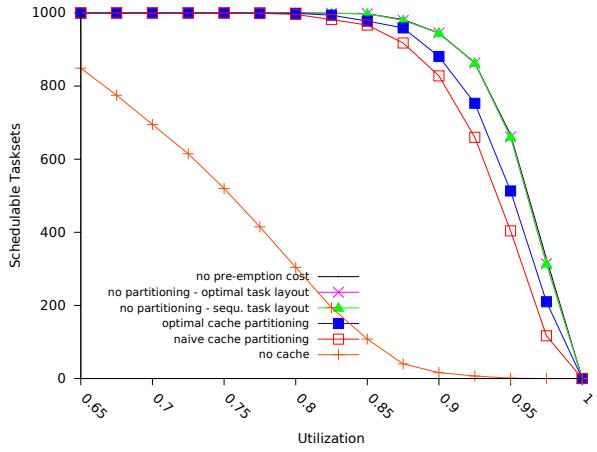
Fig. 4: Evaluation of PapaBench Benchmarks.



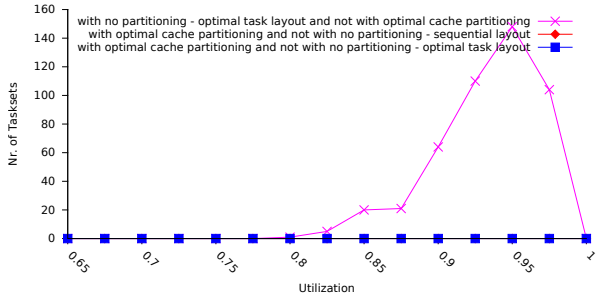
(a) Number of tasksets deemed schedulable at the different total utilizations (instruction cache with perfect data cache).



(b) Number of tasksets deemed schedulable with one approach and not another (instruction cache with perfect data cache).



(c) Number of tasksets deemed schedulable at the different total utilizations (data cache with perfect instruction cache).



(d) Number of tasksets deemed schedulable with one approach and not another (data cache with perfect instruction cache).

Fig. 5: Evaluation of Mälardalen Benchmarks.

partitioned cache. The evaluation using randomly generated tasksets enables us to fully control all relevant parameters, which is not possible using the benchmark tasks directly.

The task parameters used in our experiments were randomly generated as follows:

- The default taskset size was 10.
- Task utilizations were generated using the UUnifast [8] algorithm.
- Task periods were generated according to a log-uniform distribution with a factor of 1000 difference between the minimum and maximum possible task period and a minimum period of 5ms. This represents a spread of task periods from 5ms to 5s, thus providing reasonable correspondence with real systems.
- Task execution times were set based on the utilization and period selected: $C_i = U_i \cdot T_i$.
- Task deadlines were implicit
- Priorities were assigned in deadline monotonic order.
- The jitter of all tasks was assumed to be zero, i.e. $J_i = 0$.

To model the variation in the execution time, we randomly selected one of the execution time functions from our benchmarks (see Table I, Table III and Figure 1). Note that this only affects the variation of the execution time for different partition-sizes and C_i refers to the base execution time when τ_i can use the complete cache.

The following parameters affecting pre-emption costs were also varied, with default values given in parentheses:

- The number of cache-sets ($CS = 256$).
- The block-reload time ($BRT = 8\mu s$)
- The cache usage of each task, and thus, the number of ECBs, were generated using the UUnifast [8] algorithm (for a total cache utilization $CU = \sum_i |\text{ECB}|/CS = 4$). UUnifast may produce values larger than 1 which means a task fills the whole cache.
- For each task, the UCBs were generated according to a uniform distribution ranging from 0 to the number of ECBs times a reuse factor: $[0, RF \cdot |ECB|]$. The factor RF was used to adapt the assumed reuse of cache-sets to account for different types of real-time applications, for example, from data processing applications with little reuse up to control-based applications with heavy reuse (default $RF = 0.3$).

The parameters of the base configuration were chosen according to the actual values observed in the case studies of the PapaBench benchmarks VI-A and the Mälardalen benchmarks VI-B. The results (Figures 6 and 7) lie between those of the case studies (Figures 4a and 5a).

Overall, cache partitioning and pre-emption cost analysis with a sequential, un-optimized task layout have similar performance; however, we note that there are also a large number of tasksets that can only be scheduled with one of the two approaches, but not with the other. This shows that cache partitioning is a viable alternative in some scenarios and detrimental in others. However, we also observe that the optimal task layout with no partitioning has a clear advantage over optimal partitioning in terms of the number of schedulable tasksets (see Figure 7).

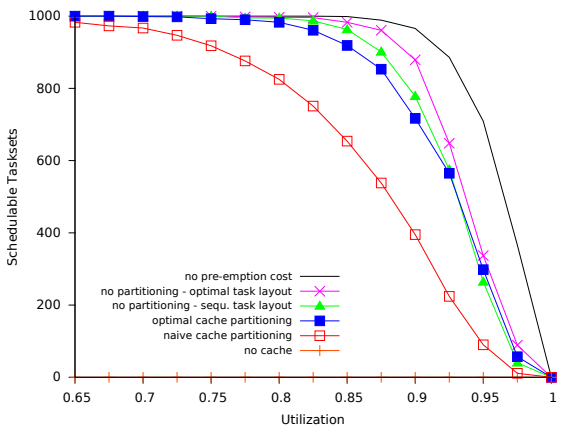


Fig. 6: Evaluation for the base configuration. Number of tasksets deemed schedulable at the different total utilizations.

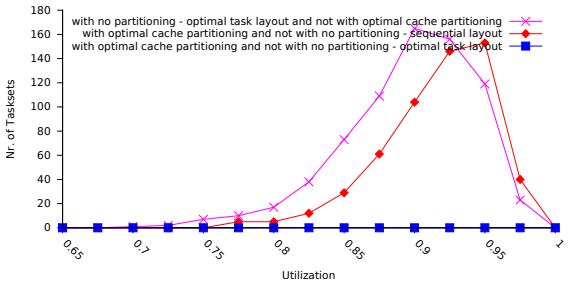


Fig. 7: Evaluation of the base configuration. Number of tasksets deemed schedulable with one approach and not another.

Exhaustive evaluation of all combinations of cache and taskset configuration parameters is not possible. We therefore fixed all parameters except one and varied the remaining parameter in order to see how performance depends on this value. The parameters we examined were: (i) the pre-emption cost as determined by the block reload time (BRT) and a scaling factor applied to task periods; (ii) the cache utilization, (iii) the number of tasks, and (iv) the cache size.

The graphs below show the weighted schedulability measure $W_y(q)$ [7] for schedulability test y as a function of parameter q. For each value of q, this measure combines data for all of the tasksets τ generated for all of a set of equally spaced utilization levels. Let $S_y(\tau, q)$ be the binary result (1 or 0) of schedulability test y for a taskset τ and parameter value q then:

$$W_y(q) = (\sum_{\forall \tau} u(\tau) \cdot S_y(\tau, q)) / \sum_{\forall \tau} u(\tau) \qquad (8)$$

where $u(\tau)$ is the utilization of taskset τ. This weighted schedulability measure reduces what would otherwise be a 3-dimensional plot to 2 dimensions [7]. Weighting the individual schedulability results by taskset utilization reflects the higher value placed on being able to schedule higher utilization tasksets.

1) Pre-emption Costs: Pre-emption costs are determined by several parameters. Among those, the dominant factors are the block reload time (BRT) and the range of task execution times. Figure 8 shows the weighted schedulability measure for different block reload times. In our setting, the break-

even point is at a block reload time of about $10\mu s$. For larger block reload times cache partitioning becomes the more effective approach, while for smaller block reload times a non-partitioned cache is more effective. In Figure 9, we varied the

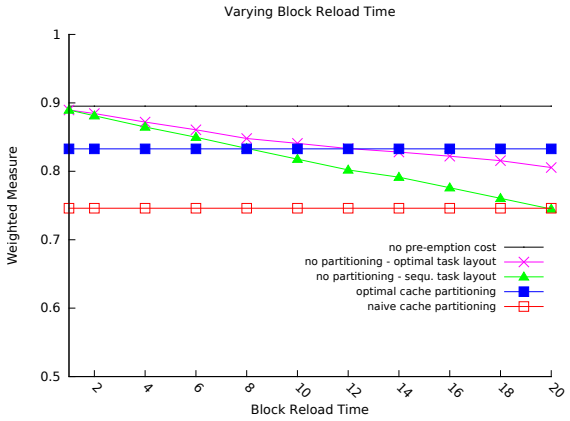


Fig. 8: Weighted schedulability measure; varying block reload time from $1\mu s$ to $20\mu s$

scaling factor w from 0.5 to 10 and hence the range of task periods given by $w[1, 100]$ms. Given that the block reload time is constant in this experiment, the ratio of pre-emption costs to taskset utilization decreases as the task periods, deadlines and execution times are all scaled up. A lower scaling factor resembles tasks with shorter execution times (as in Table I and Table II), a higher scaling factor resembles tasks with higher execution times and commensurately longer periods (as in Table III and Table IV). The results indicate that cache

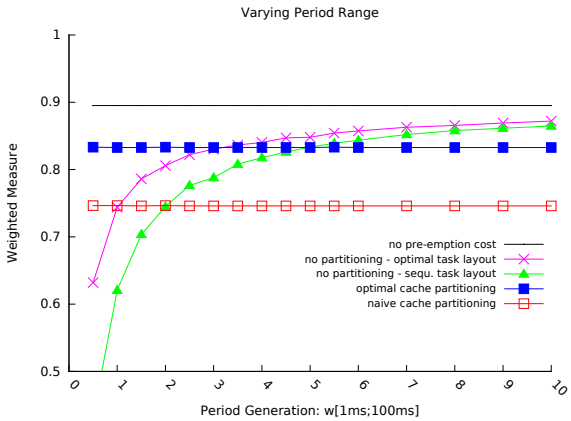


Fig. 9: Weighted schedulability measure; varying the scale of task periods $w[1, 100]$ from $w = 0.5$ to $w = 10$

partitioning is useful for control-oriented tasks with short execution times and very short periods and thus relatively high preemption costs compared to their WCET. When the pre-emption costs are low compared to the WCET, cache partitioning typically does not pay off.

Note that increasing the block reload time typically also leads to increased (non-preemptive) execution times. In these experiments, we have fixed the execution times to vary only the relation between pre-emption costs and execution time bounds.

2) Cache Utilization: The cache utilization determines the ratio between the total code size of all the tasks and the overall cache size. Increasing the cache utilization leads to higher pre-emption costs, and higher execution times in the case of cache partitioning. Cache partitioning; however, suffers less from increased cache utilization as can be seen in Figure 10. The results for the non-partitioned system suffer somewhat

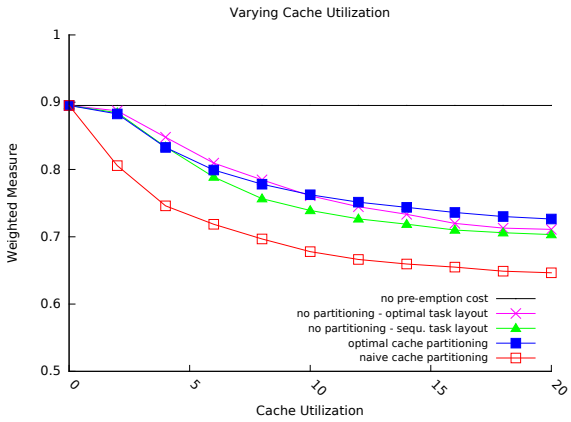


Fig. 10: Weighted schedulability measure; varying cache utilization from 0 to 20

from the over-approximation of the UCB/ECB analysis and the pre-emption cost aware response time analysis: This assumes additional cache misses due to pre-emption even though the misses have already been accounted for by a prior pre-emption, providing more pessimistic results at high cache utilization levels.

3) Number of Tasks: We also conducted experiments varying the number of tasks. Note that it is an unrealistic assumption to change the number of tasks without also changing the cache utilization. This would mean the cache usage of each individual task decreasing as more tasks are added to the system. Realistically, cache utilization increases with the number of tasks. Figure 11 shows the results of the evaluation if we increase the number of tasks and the cache utilization, while keeping the per task cache utilization constant. Here, we see

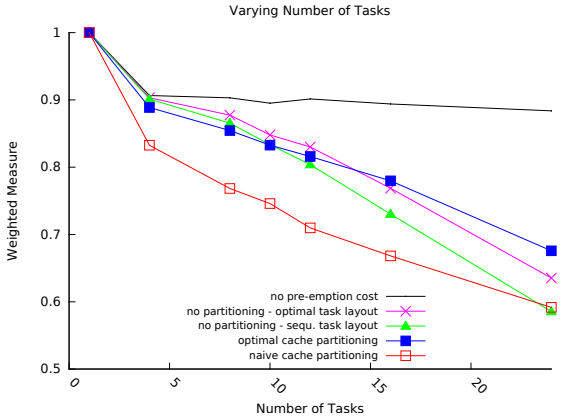


Fig. 11: Weighted schedulability measure; varying the number of tasks from 2 to 24 with constant ratio of number of tasks to cache usage

that the performance of the non-partitioned approach gradually degrades with increasing taskset size due to pessimism in the analysis of a large number of pre-emption levels.

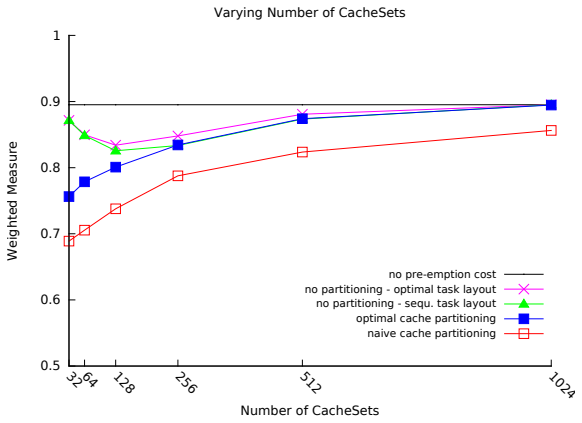


Fig. 12: Weighted schedulability measure; varying the number of cache sets 64 to 1024 with constant ratio (CU/CS)

4) Cache Size: If we only adapt the cache size without changing the relation between the execution time and the pre-emption costs (or the cache utilization), we would penalise the pre-emption cost computation: if there are more cache sets, there are also more UCBs and thus, higher pre-emption costs. The results of the cache partitioning, though, does not change. To avoid this discrimination, we have increased the number of sets, while keeping the relation of cache utilization to cache size (CU/CS) constant. The results are shown in Figure 12. For small cache, the partition sizes are very small which leads to high execution times and thus low schedulability for the paritioning approaches. For larger caches, the performance of partitioned and non-partitioned systems converge as the cache utilization decreases.

We note that small caches also lead to a reduced pre-emption overhead as the number of UCBs is upper bounded by the number of sets: The delay of additional cache reloads that would otherwise contribute to the pre-emption overhead is included in the non-pre-emptive execution time bound. The performance of the non-partitioned approaches thus declines from 32 to 128 sets (where the pre-emption overhead is maximal) as we use the task utilization (without pre-emption costs) as the baseline for each experiment.

D. Precision of the simplified Execution-Time Model

To evaluate the precision of the simplified execution time model, and so obtain a measure of the pessimism introduced in order to obtain monotonicity of execution times, we computed for each taskset an optimal cache partitioning (using Algorithm 1) (i) assuming upper bounds (Figure 3 blue upper line) and (ii) optimistic lower bounds on the execution times (Figure 3 red lower line). The difference in the results—the number of tasksets that were deemed schedulable using the lower but not the upper bounds—provides a measure of the imprecision of the simplified execution time model. In the first case study (PapaBench) 0.21% of all tasksets were deemed schedulable only using lower bounds, and 1.21% (Mälardalen and SCADE) for the second case study. Note that these percentages refer to the uncertainty due to the assumed monotonicity and not due to the cache partitioning algorithm. Also note that this does not necessarily mean that 0.21%, resp. 1.21%, of the tasksets have been falsely deemed not schedulable, rather these are upper bounds on the imprecision.

VII. Conclusions & Future Work

In this paper, we evaluated the relative performance, in terms of taskset schedulability, of partitioning the cache on a per task basis versus allowing all tasks to share the entire cache. Our research contrasts with previous work in this area, in that we used system schedulability as the performance metric, effective techniques for analysis of cache related pre-emption delays, and code from real benchmarks as the foundation of our empirical evaluation.

The main contributions of this paper are as follow:

- Sensitivity analysis of WCET with respect to partition size, showing how the precise WCET bound as a function of the size of the partition can be effectively upper and lower bounded by monotonic functions.
- The introduction of an optimal algorithm for cache partitioning, which makes use of the monotonic WCET functions.
- A thorough evaluation of the relative performance of optimal per task cache partitioning versus no partitioning.

Our results showed that for simple, short control tasks such as those from Papabench, where the pre-emption costs are relatively high compared to the WCET, the performance of partitioned and non-partitioned approaches were similar, with the use of an optimal task layout providing the non-partitioned approach with a small performance advantage. By contrast, tasks from the Mälardalen benchmark suite exhibited lower locality of memory accesses and higher amounts of computation, with larger WCETs compared to the associated cache related pre-emption delays. For tasksets based on this benchmark, the non-partitioned approach (with and without cache layout optimization) outperformed optimal partitioning. These results indicate that in most cases, the increased predictability of a partitioned cache, in terms of eliminating cache related pre-emption delays, does not compensate for the performance degradation in the WCETs.

Our extended evaluation using synthetic benchmark tasksets showed that the key parameters affecting the relative effectiveness of cache partitioning versus no partitioning are: (i) The ratio of pre-emption costs to the overall WCET (partitioning does not pay off when this ratio is small). (ii) The Block Reload Time (partitioning is most effective when the BRT is large increasing pre-emption costs). (iii) Cache utilization (the non-partitioned approach suffers from pessimism at high values of cache utilization). (iv) The number of tasks (with no partitioning the analysis suffers from increasing pessimism in the computation of pre-emption costs as the number of tasks increases). Further, we found that the relative performance of the two approaches was largely unaffected by the number of cache sets.

Our evaluation—and thus our findings—are restricted to direct-mapped L1 caches. Static cache and CRPD analyses are sufficiently precise to justify unconstrained cache usage; Cache partitioning to increase the predictability is often not required but instead detrimental to the provable system performance. It remains an open research topic if this observation also holds for more complex cache architectures where the analyses are less precise in general.

A. Future Work

There are a number of ways in which we aim to extend this line of research. Firstly, this paper considers fixed priority pre-emptive scheduling; however, the optimal partitioning algorithm derived in Section V is agnostic to the schedulability test used. It is therefore equally applicable to schedulability tests for EDF. As effective cache related pre-emption delay analysis has recently been integrated into schedulability analysis for EDF [22], we aim to make comparisons between systems using partitioned and non-partitioned cache assuming pre-emptive EDF scheduling, and to compare those results with the results given in this paper for fixed priority pre-emptive scheduling. Secondly, this paper compares two extremes, either all of the tasks share the entire cache, or every task has an individual cache partition. It is clear that between these two extremes, there is an approach which subsumes and dominates both. This intermediate approach involves allocating groups of tasks to appropriately sized cache partitions, and then controlling the layout of those tasks in memory [21] to enhance schedulability through a reduction in cache related pre-emption delays within each partition.

Last but not least, we aim to evaluate cache partitioning for more complex cache architectures.

Acknowledgements

This work was partially funded by the UK EPSRC through the MCC project (EP/K011626/1), the Engineering Doctorate Centre in Large-Scale Complex IT Systems (EP/F501374/1) and the COST Action IC1202: Timing Analysis On Code-Level (TACLe).

References

[1] S. Altmeyer and C. Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*, pages 109–118, July 2009.

[2] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In *RTSS*, pages 261–271, December 2011.

[3] 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*, 48(5):499–526, 2012.

[4] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: Tightening the crpd bound for set-associative caches. In *LCTES*, pages 153–162, April 2010.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[6] S. Baruah and A. Burns. Sustainable scheduling analysis. In *RTSS*, pages 159–168, December 2006.

[7] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *OSPERT*, pages 33–44, July 2010.

[8] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[9] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA*, pages 101–110, August 2008.

[10] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related pre-emption delay computation for set-associative caches—pitfalls and solutions. In *WCET*, July 2009.

[11] 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 *RTAS*, pages 204–212, June 1996.

[12] J. V. Busquets-Mataix and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *RTS*, June 1997.

[13] R.I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Trans. Comput.*, 57:1261–1276, September 2008.

[14] C. Ferdinand and R. Heckmann. aiT: worst case execution time prediction by static program analysis. In *IFIP*, pages 377–384, August 2004.

[15] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET*, pages 137–147, July 2010.

[16] L. Higbee. Quick and easy cache performance analysis. *SIGARCH Comput. Archit. News*, 18(2):33–44, May 1990.

[17] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986.

[18] D. B. Kirk and J. K. Strosnider. Smart (strategic memory allocation for real-time) cache design. In *RTSS*, pages 322–330, December 1990.

[19] C.-G. Lee, J. Hahn, Y.-M. Seo, S.L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[20] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, pages 12–21, 1999.

[21] W. Lunniss, S. Altmeyer, and R. I. Davis. Optimising task layout to increase schedulability via reduced cache related pre-emption delays. In *RTNS*, pages 161–170, November 2012.

[22] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *RTAS*, pages 75–84, April 2013.

[23] P. Meumeu Yomsi and Y. Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *ECRTS*, pages 280–290, July 2007.

[24] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, 1995.

[25] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *WCET*, July 2006.

[26] S. M. Petters and G. Farber. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems*, 2001.

[27] S. Plazar, P. Lokuciejewski, and P. Marwedel. Wcet-aware software based cache partitioning for multi-task real-time systems. In *WCET*, July 2009.

[28] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS*, pages 114–124, December 2002.

[29] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, pages 41–48, July 2005.

[30] Y. Tan and V. Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *Trans. on Embedded Computing Sys.*, 6(1), 2007.

[31] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems*, 7(1):4:1–4:38, December 2007.

[32] J. L. Wolf, H. S. Stone, and D. Thiébaut. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Comput.*, 41(4):388–410, April 1992.