Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis

Darshit Shah
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
s8dashah@stud.uni-saarland.de

Sebastian Hahn
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
sebastian.hahn@cs.uni-saarland.de

Jan Reineke
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
reineke@cs.uni-saarland.de
https://orcid.org/0000-0002-3459-2214

Abstract

In the presence of caches, preemptive scheduling may incur a significant overhead referred to as *cache-related preemption delay* (CRPD). CRPD is caused by preempting tasks evicting cached memory blocks of preempted tasks, which have to be reloaded when the preempted tasks resume their execution.

In this paper we experimentally evaluate state-of-the-art techniques to account for the CRPD during timing analysis. We find that purely synthetically-generated task sets may yield misleading conclusions regarding the relative precision of different CRPD analysis techniques and the impact of CRPD on schedulability in general. Based on task characterizations obtained by static worst-case execution time (WCET) analysis, we shed new light on the state of the art.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases real-time systems, timing analysis, cache-related preemption delay

Digital Object Identifier 10.4230/OASIcs.WCET.2018.7

Funding This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the project PEP, and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

Acknowledgements We thank Tina Jung who contributed the ECB and UCB analyses to `LLVMTA`.

1 Introduction

In real-time systems, it is often necessary to schedule tasks preemptively in order to meet all tasks' deadlines. Most work on preemptive scheduling is based on the assumption that the overhead incurred by preemptions is negligible and may thus be subsumed into the worst-case execution time of each task. When tasks are executed on complex microarchitectures with caches, this assumption is problematic: Preempting tasks may alter the state of the cache, leading to an increased execution time of preempted tasks once they are resumed, because their data has been evicted from the cache and needs to be reloaded from main memory. The additional execution time due to such reloads is known as *cache-related preemption delay* (CRPD).

As the CRPD depends both on the “low-level” cache aspect and on “higher-level” scheduling decisions, it is tackled by a combination of low-level static analysis, characterizing each task’s “cache footprint”, and CRPD-aware response-time analysis, bounding a task’s response time using the low-level characterizations. Altmeyer et al. [1] provide an overview of the state-of-the-art techniques to account for CRPD during response-time analysis. To experimentally evaluate the different CRPD-aware response-time analyses, Altmeyer et al. [1] use synthetically-generated task sets with synthetically-generated task characteristics.

In this paper, we experimentally evaluate the state of the art concerning CRPD-aware timing analysis to gain further insights into where future research on CRPD may be profitable. To this end, we attempt to answer the following questions:

- Can we reproduce the results obtained in the experimental evaluation of Altmeyer et al. [1] based on synthetic task sets?
- Do we obtain similar per-task characteristics (worst-case execution time (WCET) values, number of Evicting Cache Blocks (ECBs), and number of Useful Cache Blocks (UCBs)) as Altmeyer et al. [1] based on our low-level analysis toolchain?
- If we base the experimental evaluation on task characterizations obtained by static WCET analysis, do we observe similar trends as those observed in [1] based on synthetic task sets? If not, why? Related to the previous question: Are the parameters for the synthetic task set generation meaningful?

Our paper is structured as follows: We summarize the background concerning caches and CRPD-aware timing analysis in Section 2. In Section 3 we discuss relevant details of our analysis implementation. Then, in Section 4 we present the results of our experimental evaluation, partially answering some of the questions listed above. We conclude the paper with a summary of our findings in Section 5.

2 Background and Related Work

2.1 Caches

Caches are small but fast memories that store a subset of the main memory’s contents to bridge the latency gap between processors and DRAM-based main memory.

To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* of a certain size. Blocks are cached as a whole in cache lines of the same size. When accessing a memory block, the cache logic has to determine whether the block is stored in the cache, a *cache hit* or not, a *cache miss*.

To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set is called the *associativity* of the cache. Caches of associativity one are called *direct mapped*. Most work concerning CRPD-aware response time analysis has been conducted in the context of direct-mapped caches. Such caches are also the focus of this work.

2.2 Timing Analysis for Preemptive Scheduling

Timing analysis is traditionally separated into two phases:

1. Worst-case execution time (WCET) analysis, which determines bounds on each task’s execution time. Usually C_i denotes the WCET bound of task τ_i.
2. Response-time analysis (RTA), which determines bounds on each task’s response time under a particular scheduling algorithm; based on the tasks’ WCET bounds, minimum inter-arrival times (also referred to as periods), denoted by T_i, and release jitter, denoted by J_i.

If no task's response time may exceed its relative deadline D_i, then the task set is determined to be schedulable.

Traditional response-time analysis assumes that preemptions are free, i.e., context switches are performed instantaneously and the execution times of tasks are not affected by the execution of preempting tasks. For fixed-priority preemptive scheduling, the least solution of the following recursive equation [3, 12] then determines a task's response time:

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

where $\mathrm{hp}(i)$ denotes the indices of those tasks that have a higher priority than task τ_i.

Unfortunately, context switches cannot be performed instantaneously for several reasons: 1. The scheduler takes some time to select the next task to execute. 2. Upon a context switch, the hardware needs to save the contents of registers and restore the contents of the task whose execution is to be resumed. This process also results in flushing the pipeline. It is commonly assumed that the cost of these two actions can be bounded by a constant, which can then be taken into account by appropriately inflating each task's WCET bound.

In the presence of caches, however, preemptive scheduling may incur an additional overhead, called *cache-related preemption delay* (CRPD): The execution time of preempted tasks may be prolonged due to additional cache misses caused by cache evictions of preempting tasks.

To account for CRPD, Equation (1) can be extended by $\gamma_{i,j}$ representing the preemption cost due to each job of a higher-priority preempting task τ_j executing within the worst-case response time of task τ_i [6]:

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

Note, that the response time calculation inherently relies on timing compositionality [10] of the cache-related preemption effects. Recent work in the area of multi-core timing analysis [9] has shown that the compositionality assumption is often violated even on simple hardware platforms. However, the analysis techniques of [9] could be adopted to still allow for a compositional reasoning.

2.3 Characterizing a Task's Cache Footprint

To bound $\gamma_{i,j}$ one needs to bound the number of additional cache misses in preempted tasks τ_i due to the execution of preempting tasks. The number of such cache misses, depends on the memory-access behavior of both the preempted task and its preempting tasks.

To characterize preempting tasks, Busquets-Mataix et al. [6] introduced the notion of ECBs: A memory block is an ECB of task τ_j if it may be accessed during τ_j's execution. For the computation of cache-related preemption delays the precise identity of a memory block is irrelevant. What is important is which cache set an ECB maps to, which is where it may potentially evict cached memory blocks of a preempted task. Further, in case of direct-mapped caches, if two or more ECBs map to the same cache set, they may not do a greater damage than an individual ECB mapping to this cache set. Thus, in case of direct-mapped caches – which we focus on in this paper – one may characterize the set of ECBs of a task by a set $ECB_j \subseteq \{0, \ldots, N-1\}$, capturing the subset of cache sets that a task's evicting cache blocks map to.

To characterize preempted tasks, Lee et al. [13] introduced the notion of UCBs: A memory block m is a UCB of task τ_i if there is a program point P within τ_i, such that m may be cached at P and m may be reused at a later program point P', which may be reached from P without eviction of memory block m along the execution from P to P'. Intuitively, only useful cache blocks may result in additional cache misses due to preemptions. As there may be at most one useful cache block in each cache set of a direct-mapped cache at any point in time, the set of UCBs of a task may be represented by a set $UCB_i \subseteq \{0, \dots, N-1\}$, capturing the cache sets that a task's useful cache blocks map to.

Later, Altmeyer and Maiza [2] introduced the notion of *definitely-cached useful cache blocks* (DC-UCBs): A memory block m is a DC-UCB of task τ_i if there is a program point P within τ_i, such that m must be cached at P and m may be reused at a later program point P', which may be reached from P without eviction of memory block m along the execution from P to P', and, crucially, m is considered to be a cache hit at P' by the WCET analysis. The set of DC-UCBs is always a subset of the set of UCBs. The observation in the definition of DC-UCBs is that CRPD analysis needs to only account for preemption-induced cache misses that are not already conservatively accounted for during WCET analysis.

2.4 CRPD-Aware Response-Time Analysis for Fixed-priority Scheduling

Based on the sets of ECBs and UCBs of all tasks, there are different ways of defining $\gamma_{i,j}$, such that the response times of tasks are correctly bounded by solutions of (2). In the following, we briefly summarize the six state-of-the-art approaches from Altmeyer et al. [1] that apply to direct-mapped caches. More details can be found in [1]. The methods can be extended to be applied to set-associative caches with LRU replacement, but not to caches with FIFO or PLRU replacement [5]. Those six approaches follow from two different interpretations of $\gamma_{i,j}$ that differ in case of nested preemptions:

1. "Effect of the preempting task": In this case $\gamma_{i,j}$ bounds the cost of additional misses in the preempted tasks due to execution of the preempting task τ_j.
2. "Effect on the immediately preempted task": In this case $\gamma_{i,j}$ bounds the cost of additional misses in the task immediately (i.e. not in a nested fashion) preempted by τ_j, due to τ_j's execution *and* the execution of higher-priority tasks which may in turn have preempted τ_j.

Effect of the Preempting Task

Busquets-Mataix [6] and later Tomiyama and Dutt [16] used the number of ECBs of the preempting task to bound the preemption cost, in what Altmeyer et al. [1] termed the *ECB-Only* approach:

$$\gamma_{i,j}^{ECB} = \text{BRT} \cdot |ECB_j|, \tag{3}$$

where BRT denotes the *block reload time*, i.e., the time to fetch one memory block from main memory into the cache.

Tan and Mooney [15] improved upon *ECB-Only* by also considering the set of UCBs of all tasks that may be affected by the preemption by τ_j. This approach is termed *UCB-Union* in [1]. Altmeyer et al. [1] introduced the *UCB-Union-Multiset* approach, which improves upon *UCB-Union* by taking into account how often different tasks may preempt each other based on their minimum inter-arrival times.

Effect on the Immediately-Preempted Task

Lee et al. [13] introduced the *UCB-Only* approach, which bounds cost of a preemption in the immediately-preempted task by considering its UCBs:

$$\gamma_{i,j}^{UCB} = \mathrm{BRT} \cdot \max_{\forall k \in \mathrm{aff(i,j)}} \{|UCB_k|\}, \tag{4}$$

where $\mathrm{aff}(i,j) = \mathrm{hep}(i) \cap \mathrm{lp}(j)$ is the set of tasks that have a lower priority than task τ_j but a higher-or-equal priority than task τ_i, the task under analysis. Thus $\mathrm{aff}(i,j)$ is the set of tasks that task τ_j may immediately preempt during task τ_i's response time.

Altmeyer et al. [1] later introduced the *ECB-Union* and the *ECB-Union-Multiset* approaches, which additionally take into account the ECBs of the preempting tasks, and the number of times tasks may preempt each other based on their minimum inter-arrival times.

As the *UCB-Union-Multiset* and the *ECB-Union-Multiset* approaches are incomparable, they may be combined by taking the minimum response time of the two approaches for each task, to yield an approach, coined *Combined-Multiset*, that dominates the two [1].

3 Implementation

In this section we describe the tools used for the experiments presented later in this paper. This includes the tools for analysis of tasks to determine task characteristics, for generation of synthetic task sets and for running the schedulability analysis.

3.1 Obtaining Task Characteristics

We use our low-level timing analysis tool `LLVMTA` [9] to compute not only the worst-case execution time bound for a given task, but also its preemption-related characteristics: ECBs, UCBs, and DC-UCBs. `LLVMTA` supports the detailed microarchitectural analysis of different processors. In this paper, we use the model of a conventional in-order pipeline with five stages [11], static branch prediction, and native support for floating-point instructions. The main memory is accessed via separate instruction cache and data scratchpad. We employ standard must- and may-analyses [7] to predict the instruction cache behavior. Similar to [2], our analyses are context sensitive: they (virtually) peel the first iteration of loops and distinguish the different call sites for each function. This is important to achieve precise analysis results [7].

The microarchitectural analysis results in a *microarchitectural execution graph* [9] whose nodes correspond to abstract microarchitectural states and whose edges represent the possible execution flow. To obtain the worst-case execution time bound, we use an integer linear program to calculate the longest path through the microarchitectural execution graph – also known as implicit path enumeration [14].

This microarchitectural execution graph also contains detailed information about the memory accesses initiated in the pipeline. Unlike the control-flow graph of a program, it explicitly includes speculative accesses triggered by branch prediction or even access reorderings in out-of-order pipelines. To ensure soundness, we thus perform the ECB and DC-UCB analysis on this microarchitectural execution graph rather than on the control-flow graph of the program. Our implementation follows the data-flow description of the analysis in [2].

The set of useful cache blocks is a property of a particular program point. To be able to use the sets in an efficient way in the schedulability analysis, we need to combine the program-point-sensitive results. In accordance with [1], we calculate the set of those cache sets that exhibit a useful cache block at any program point.

Instructions that share a single cache line are usually executed consecutively in straight-line code fragments (spatial locality). As a consequence, the cache line is useful at the program point between two instruction of that cache line. In the program-point-insensitive result, almost every cache line in the program is (definitely-cached) useful due to spatial locality if the line size exceeds the word size. Similar to [2], blocks that are useful only due to immediate spatial locality can be ignored in the low-level analysis, if they are compensated for by one additional cache reload per preemption in the schedulability analysis.

3.2 Task Set Generation and Schedulability Analysis

The generation of synthetic task sets and the CRPD-aware schedulability analysis is done using a new tool that we developed using the Rust programming language. This tool performs CRPD-aware schedulability analyses of [2] on synthetically generated task sets using either the task characteristics provided by `LLVMTA` or by synthetically generating them.

4 Evaluation

In this section we experimentally evaluate the differences between the various approaches discussed in Section 2.4 using both synthetically-generated task characteristics and task characteristics derived through our WCET analysis tool, `LLVMTA`. In the latter case, we use the programs from the Mälardalen test bench [8] as tasks.

As in most previous work on cache-related preemption delay, including [1, 2], we assume a direct-mapped instruction cache and a data scratchpad. The size of the data scratchpad is sufficient to not cause preemption-induced reloads. To compare our results to [1], we use a cache with a line size of 8 bytes and 256 sets, i.e. a total size of 2 kB.

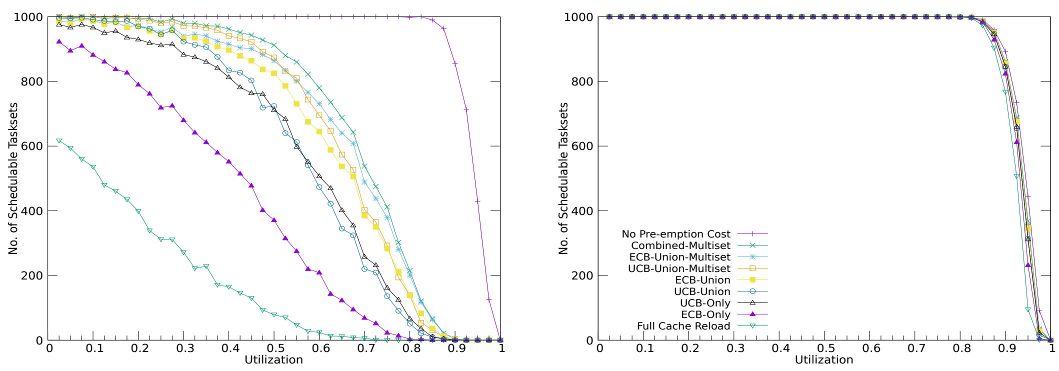
In each of our evaluations, we have compared the six CRPD approaches listed in Section 2.4, along with the optimistic *No Preemption Cost* case where preemptions are considered to the completely free and the extremely pessimistic, *Full Cache Reload* case where each preemption causes the entire cache of the preempted task to be reloaded.

4.1 Timing Analysis With Synthetically-Generated Task Characteristics

In order to replicate the results of [1], we used our tool to generate synthetic task sets with the same parameters as in the original experiment. The task characteristics such as WCET, UCB, and ECB values are also synthetically generated following the methodology used in [1]. Since, the clock frequency is not mentioned in the original study, we assumed a frequency of 100 Mhz[1].

For the experimental evaluation, 1000 task sets with 10 tasks each were generated for every utilization value from 0.025 to 1.000 in steps of 0.025 using the UUnifast [4] algorithm. The task periods were generated according to a log-uniform distribution with the minimum period as 500,000 cycles (5 ms) and a maximum period of 50,000,000 cycles (500 ms). The cost of a cache miss, also known as the Block Reload Time, is considered to be 800 cycles (8 μs). The cache utilization (CU) of a task is the ratio of the number of ECBs of a task to the total number of cache sets available. The cache utilizations for the tasks in a task

[1] As all task parameters are given in terms of wall-clock time, the particular processor frequency has a negligible effect on the experimental results.



(a) Block reload time, BRT = 800. **(b)** Block reload time, BRT = 20.

Figure 1 Results using synthetic task sets with synthetically-generated task characteristics.

set were generating using UUnifast[2], considering a total CU = 10. The number of ECBs of a task are then computed using the generated cache utilization values. The ratio of the number of UCBs of a task to the number of ECBs is defined as the reuse factor (RF). For this experiment, the reuse factor for each task was picked uniformly at random within the range [0, 0.3]. All of these values are taken from the parameters of the original study.

For the results of the schedulability tests, see Figure 1a. Our first observation is that the resulting schedulability graph closely matches the corresponding graph in Figure 9 of [1], which confirms the correctness of the independent implementations of schedulability analyses used in both papers.

The latency of actual main memory modules[3] are at least an order of magnitude lower than the 8 μs (800 cycles) used in the original experiment. In [2], Altmeyer et. al. use a memory latency of just 4 cycles for their WCET analysis and yet the generation of synthetic task characteristics uses a latency of 800 cycles. To gauge the effect of a more realistic memory latency, we re-ran the experiment, however this time with a lower memory latency of 20 cycles. The results of this experiment can be seen in Figure 1b. As is clearly evident from the figure, reducing the memory latency almost completely eliminates the effect of CRPD on overall schedulability. We must emphasize here that we do not claim that CRPD has a negligible effect on the overall schedulability, but rather that the methods used to synthetically generate task characteristics provide misleading results. With a lower memory latency, we see that even in the case of having to pay the penalty of a full cache refresh for each preemption, the schedulability of the task sets closely follows the case where preemption is considered to be free. The reason for such behavior is that the generation of synthetic task characteristics does not take into account the fact that both the WCET and the CRPD depend on the memory latency, and are thus correlated. However, in our experiment, decreasing the memory latency reduces the CRPD without similarly reducing the execution time of the tasks at all. At this point, the WCET almost completely dominates the response time of the task.

[2] UUnifast may generate values greater than one, indicating that the ECBs fill the entire cache. The number should be capped to the number of cache sets. However, for the computation of UCBs, the original value of ECBs is used.

[3] As example, consider this automotive SDRAM `https://www.micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf`

4.2 Comparison of Task Characteristics

In this section, we compare the per-task characteristics provided in [1, 2] for a subset of the Mälardalen benchmarks with the numbers calculated using `LLVMTA`. The results are shown in Table 1 for the subset of benchmarks analyzed in [1, 2] in the upper half and for the remaining benchmarks in the lower half. The apparent gap between the WCET bounds is mostly explained by different memory latencies. While we use a memory latency of 20 cycles throughout the paper, they use a low latency of only 4 cycles as described in [2]. Furthermore, the different compilers (`gcc` versus `clang+LLVM`), hardware platforms, and analysis settings (e.g. loop bound annotations) contribute to this gap. The lower WCET bound (and significantly lower ECB value) – compared to [1] – for some benchmarks, such as `sqrt` and `qurt`, are explained by the native floating point support of our pipeline model in contrast to software emulation, which is assumed in [1].

Taking the above differences in the underlying model into account, our ECB values and the cache utilizations resemble the ones provided by Altmeyer et al. [1].

As described in Section 3.1, column *DC-UCB* of Table 1 shows the size of the set containing all cache sets that exhibit a useful cache block at some program point. In addition, column *Max DC-UCB* shows the maximum number of cache blocks that are (definitely-cached) useful at a single program point. This number can be used to improve the *UCB-Only* approach, which we call *UCBMax-Only*. Note that *DC-UCB* and *Max DC-UCB* results from different schemes to aggregate program-point specific information of the static cache analysis. Thus, there is no reason in considering *UCBMax-Only* in Section 4.1 when task characteristics are synthetically generated.

Unlike the ECB values, our DC-UCB values differ significantly from the values in [1]. Compiling our benchmarks with optimizations[4] reduces the number of DC-UCBs, but also the number of ECBs which resulted in significant gaps for ECBs and DC-UCBs. The only way to get similarly low DC-UCB values using our toolchain is to disable the (virtual) loop peeling which worsens the must cache analysis and thus decreases the number of DC-UCBs. However, according to [2], their numbers were obtained with loop peeling enabled. As a result of the higher number of DC-UCBs in our analysis, we also see a higher value of the reuse factor in our tasks.

4.3 Timing Analysis With Analysis-Derived Task Characteristics

In this section, we conduct schedulability experiments with task characteristics derived using `LLVMTA` rather than generated synthetically. For deriving task characteristics, we used the programs in the Mälardalen suite and derived the WCET and the ECB and DC-UCB sets for each task as described in Section 3.1 and discussed in Section 4.2. These characteristics were then used to generate synthetic task sets. In addition to the approaches outlined in Section 4, in this section, we also consider the *UCBMax-Only* approach which was introduced in Section 4.2.

As before, 1000 task sets were generated for each utilization value from 0.025 to 1.000 in steps of 0.025. Each task set consists of 10 randomly selected tasks from the set of tasks found in Table 1. The utilization U_i of each of the 10 tasks within a task set was generated using the UUnifast algorithm [4] and the periods were computed based on the following equation: $T_i = C_i/U_i$. Since the WCETs of the tasks were derived through analysis, we did not have control over the range of task periods. Implicit task deadlines, i.e. $D_i = T_i$, were

[4] It is not specified whether the programs in [1] have been compiled with or without optimizations.

Table 1 Comparison of task characteristics. In parenthesis: results without virtual loop peeling. * denotes use of floating-point calculations.

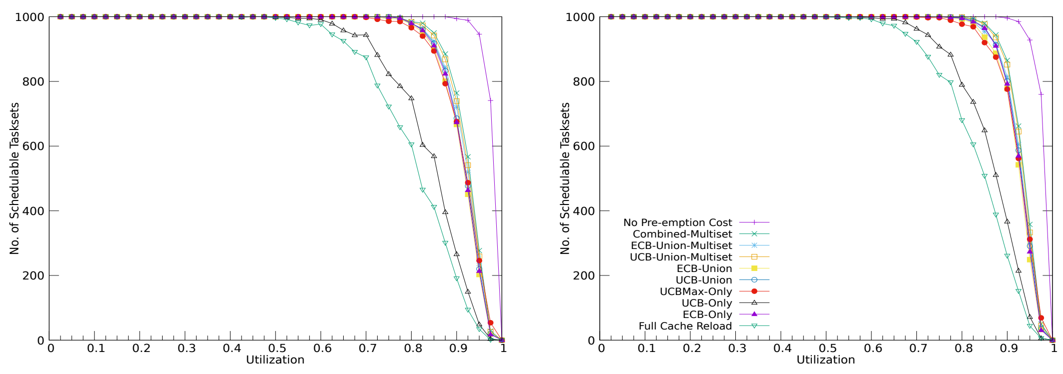
	WCET		ECB		DC-UCB		Max	CU		RF	
	Theirs	Ours	Theirs	Ours	Theirs[5]	Ours	DC-UCB	Theirs	Ours	Theirs	Ours
bs	445	3052	35	43	5	23 (9)	20	0.137	0.168	0.143	0.535
bsort100	1567222	3146185	62	57	8	40 (11)	30	0.242	0.223	0.129	0.702
crc	290782	1621246	144	127	14	63 (28)	35	0.562	0.496	0.097	0.496
fibcall	1351	8406	24	28	5	16 (6)	16	0.094	0.109	0.208	0.571
fir	29160	12406071	105	90	9	41 (16)	22	0.410	0.352	0.086	0.456
insertsort	6573	11291	41	29	10	16 (5)	15	0.160	0.113	0.244	0.552
matmult	742585	1447379	100	85	23	51 (26)	31	0.391	0.332	0.230	0.600
ns	43319	126865	64	55	13	37 (12)	34	0.250	0.215	0.203	0.673
qsort-exam *	22146	163089	170	142	15	83 (38)	39	0.664	0.555	0.088	0.585
qurt *	214076	71655	484	130	14	40 (32)	26	1.891	0.508	0.029	0.308
select *	17088	6306	151	159	15	73 (35)	55	0.590	0.621	0.099	0.459
sqrt *	39962	22436	477	53	14	21 (13)	12	1.863	0.207	0.029	0.396
adpcm	-	82368867	-	256	-	229	108	-	1.000	-	0.895
cnt	-	127558	-	123	-	58	44	-	0.480	-	0.472
compress	-	1098331	-	247	-	149	57	-	0.965	-	0.603
cover	-	71967	-	256	-	38	15	-	1.000	-	0.148
edn	-	739514	-	256	-	220	120	-	1.000	-	0.859
expint	-	2144875	-	113	-	63	36	-	0.441	-	0.558
fdct	-	10258	-	126	-	113	62	-	0.492	-	0.897
fft1 *	-	257657	-	222	-	154	63	-	0.867	-	0.694
janne_complex	-	33778	-	39	-	28	27	-	0.152	-	0.718
jfdctint	-	21742	-	132	-	122	54	-	0.516	-	0.924
lcdnum	-	6129	-	50	-	14	10	-	0.195	-	0.280
lms *	-	10793664	-	242	-	134	38	-	0.945	-	0.554
ludcmp *	-	116312	-	210	-	168	44	-	0.820	-	0.800
minver *	-	67157	-	256	-	178	47	-	1.000	-	0.695
ndes	-	1050167	-	253	-	178	38	-	0.988	-	0.704
nsichneu	-	201969	-	256	-	183	2	-	1.000	-	0.715
prime	-	7726328	-	79	-	50	38	-	0.309	-	0.633
st *	-	3763684	-	192	-	95	52	-	0.750	-	0.495
statemate	-	41776	-	256	-	111	2	-	1.000	-	0.434
ud	-	349120	-	188	-	161	42	-	0.734	-	0.856

considered and tasks priorities were assigned in deadline-monotonic order. We considered two values for the block reload time, BRT: 20 cycles and 800 cycles, which are used both in the calculation of the WCET and the CRPD bounds. The resulting schedulability graphs can be found in Figures 2b and 2a respectively.

The first observation is that the results in Figures 2a and 2b look very similar; very much unlike those in Figures 1a and 1b. As the BRT value is taken into account *both* during WCET and during CRPD analysis, the WCET and CRPD values in the BRT=800 case are both about 40 times higher than in the BRT=20 case. As the task periods are generated based on the WCET values, the periods are similarly 40 times higher on the average, and thus the relative impact of the preemptions is essentially the same in both cases.

In contrast to the results based on synthetically-generated task characteristics, the *UCB-Only* approach performs considerably worse than the other approaches, in particular much worse than *ECB-Only*. This is due to the higher number of DC-UCBs in our analysis as we

[5] Note, that the authors of [1] refer to DC-UCBs as UCBs in their evaluation.



(a) Block reload time, BRT = 800. **(b)** Block reload time, BRT = 20.

Figure 2 Results using synthetic task sets with experimentally determined task characteristics.

have explained in Section 4.2. The *UCBMax-Only* approach however, performs similarly well as the rest of the approaches.

In contrast to the results based on synthetically-generated task characteristics, the *ECB-Only* approach performs similarly to the more sophisticated approaches that take into account both ECBs and DC-UCBs. This happens due to the fact that the high number of DC-UCBs does not help to improve the performance of these other approaches much.

Comparing Figure 2b with Figure 1b, we observe a wider gap between the optimistic *No Preemption Cost* and the pessimistic *Full Cache Reload* approaches. This implies that there is a greater CRPD overhead in the task sets than is apparent in the case of synthetically-generated task characteristics at BRT=20, but less than at BRT=800.

5 Conclusions and Open Questions

In this paper, we experimentally evaluate state-of-the-art CRPD-aware timing analysis approaches. First, we reproduce the response-time analysis results of prior work [1] for purely synthetic task sets based on our own independent implementation of schedulability analyses, thereby increasing confidence in the correctness of both implementations.

Choosing a more realistic value for the block reload time than prior work, we obtain very different results that superficially seem to indicate that the CRPD overhead would be negligible. To further investigate this issue, we next obtain task characterizations using static analysis for all benchmarks in the Mälardalen suite. Our analysis results closely match prior analysis results in case of ECBs, but, surprisingly, not in case of DC-UCBs. Based on these task characterizations, we generate synthetic task sets and use them to evaluate the state-of-the-art approaches. We find that

1. The effect of the cache-related preemption delay on the overall response times – and thus overall schedulability – is not negligible, but smaller than suggested by the original results in [1].
2. The block reload time does not significantly influence the impact of the CRPD on overall schedulability, *if* it is taken into account both during WCET and CRPD analysis.
3. The simple *ECB-Only* approach is competitive with the more sophisticated alternatives, such as the *UCB-Union-Multiset* approach. This is due to the fact that our static analysis classifies more blocks as DC-UCBs than prior work.

Our findings lead us to conclude that *either* synthetic task sets should be generated from characteristics derived by low-level analysis of actual programs rather than synthetically-generated characteristics; *or* better task characteristics generators are required that do not miss important dependencies between different characteristics such as WCET and number of ECBs/UCBs. Analogously, we suspect existing synthetically-generated task sets are not representative of real-world system-level workloads. Thus, obtaining real-world system-level benchmarks together with the low-level characteristics of the involved tasks is an important future step.

References

1 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Syst.*, 48(5):499–526, 2012.

2 Sebastian Altmeyer and Claire Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57:707–719, August 2011.

3 Neil C. Audsley et al. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineerung Journal*, 1993.

4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. `doi:10.1007/s11241-005-0507-9`.

5 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches—pitfalls and solutions. In *WCET*, June 2009.

6 José V. Busquets-Mataix et al. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *RTAS*, pages 204–212, 1996.

7 Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999. `doi:10.1016/S0167-6423(99)00010-6`.

8 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *WCET*, pages 136–146, 2010.

9 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *RTNS*, pages 299–308, 2016. `doi:10.1145/2997465.2997471`.

10 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. `doi:10.1145/2752801.2752805`.

11 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann, 2012.

12 Mathai Joseph and Paritosh Pandya. Finding response times in real-time system. *The Computer Journal*, 29(5), 1986.

13 Chang-Gun Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

14 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS)*, pages 88–98, 1995. `doi:10.1145/216636.216666`.

15 Yudong Tan and Vincent John Mooney III. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embedded Comput. Syst.*, 6(1):7, 2007. `doi:10.1145/1210268.1210275`.

16 Hiroyuki Tomiyama and Nikil D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, pages 67–71, 2000.