Pre-emption cost aware response time analysis for fixed priority pre-emptive systems

Sebastian Altmeyer
Compiler Design Lab
Saarland University, Germany
Email: altmeyer@cs.uni-saarland.de

Robert I. Davis
University of York
York, UK
Email: rob.davis@cs.york.ac.uk

Claire Maiza
Verimag
INP Grenoble, France
Email: Claire.Maiza@imag.fr

***Abstract*—Without the use of cache the increasing gap between processor and memory speeds in modern embedded microprocessors would have resulted in memory access times becoming an unacceptable bottleneck. In such systems, cache related pre-emption delays can be a significant proportion of task execution times. To obtain tight bounds on the response times of tasks in pre-emptively scheduled systems, it is necessary to integrate worst-case execution time analysis and schedulability analysis via the use of an appropriate model of pre-emption costs.**

In this paper, we introduce a new method of bounding pre-emption costs, called the ECB-Union approach. The ECB-Union approach complements an existing UCB-Union approach. We combine the two into a simple composite approach that dominates both. These approaches are integrated into response time analysis for fixed priority pre-emptively scheduled systems. Further, we extend this analysis to systems where tasks can access resources in mutual exclusion, in the process resolving omissions in existing models of pre-emption delays. A case study and empirical evaluation demonstrate the effectiveness of the ECB-Union and combined approaches for a wide range of different cache configurations including cache utilization, cache set size, reuse, and block reload times.

I. Introduction

During the last two decades, applications in aerospace and automotive electronics have progressed from deploying embedded microprocessors clocked in the 10's of MHz range to significantly higher performance devices operating in the high 100's of MHz to GHz range. The use of such high performance embedded processors has meant that memory access times have become a significant bottleneck, necessitating the use of cache to tackle the increasing gap between processor and memory speeds.

In the majority of research papers on fixed priority pre-emptive scheduling an assumption is made that the costs of pre-emption can either be neglected or sub-summed into the worst-case execution time of each task. With today's high performance embedded processors, pre-emption costs can make up a significant proportion of each task's execution time. Such costs cannot be neglected nor is it necessarily viable to simply subsume them into worst-case execution times, as this can lead to a pessimistic overestimation of response times.

In this paper, we consider the costs incurred when a pre-empting task evicts useful cache blocks of a pre-empted task. These useful cache blocks subsequently need to be reloaded after the pre-empted task resumes execution, introducing an additional cache related pre-emption delay (CRPD).

Non-pre-emptive scheduling is one way of avoiding such cache-related pre-emption costs; however, disabling pre-emption is often not an option. Systems that include tasks or interrupt handlers with short deadlines typically cannot disable pre-emption for the full duration of each task's execution. An alternative approach is co-operative scheduling, with re-scheduling only possible at specific pre-emption points within each task, or after a pre-determined time has elapsed, thus dividing each task into a series of non-pre-emptable sections. Recently, significant progress has been made in this area, with algorithms designed to make an optimal selection of pre-emption points [10, 11]. These algorithms minimise the overall cost of pre-emption for each task while maintaining the schedulability of the taskset as a whole. However, difficulties remain, for example in determining the placement of pre-emption points when the code includes branches and loops.

Exact response time analysis for fixed priority pre-emptive systems was developed during the 1980's and 1990's and subsequently refined into a set of engineering techniques [20, 5, 19]. However, basic response time analysis does not consider cache-related pre-emption costs explicitly. Explicit integration of pre-emption costs has previously been considered in a number of ways: analyzing the effect of the pre-empting task [15, 32], the effect on the pre-empted task [22], or a combination of both [30, 31]. With later refinements giving an upper bound on the number of pre-emptions [27].

In fixed priority pre-emptive systems, there are a number of ways of managing task priorities that can be used to reduce the number of pre-emptions and hence the overall pre-emption costs. These include; non-pre-emption groups [18], pre-emption thresholds [21, 28, 33], and FP-FIFO scheduling [25], which is supported by a large number of real-time operating systems, including the Linux kernel (SCHED_FIFO).

In this paper, we build upon previous work that integrates pre-emption costs into response time analysis for fixed priority pre-emptive scheduling. Section II introduces the scheduling model, terminology, and notation used. In Section III, we review existing approaches to integrating pre-emption costs into response time analysis. Building on the insights gained from this review, Section IV introduces the new ECB-Union approach to computing pre-emption costs. The ECB-Union approach complements an existing UCB-Union approach. We combine the two into a simple composite that dominates both. In Section V, we extend our analysis to systems where tasks can access resources in mutual exclusion, in the process resolving omissions in existing models of pre-emption delays. A case study in Section VI and an empirical evaluation in Section VII demonstrate the effectiveness of the ECB-Union and combined approaches for a wide range of different task parameters and cache configurations. Section VIII concludes with a summary of the main contributions of the paper.

The research in this paper focuses on fixed priority pre-emptive scheduling with unique priority levels; however, the approaches derived are also applicable to FP-FIFO scheduling. Extension to FP-FIFO scheduling is described in Appendix A.

II. Task model, Terminology, and Notation

We are interested in an application executing under a fixed priority pre-emptive scheduler on a single processor. The application is assumed to comprise a static set of n tasks $(\tau_1, \tau_j, \ldots, \tau_n)$, each assigned a fixed priority. We use the notation hp(i) (and lp(i)) to mean the set of tasks with priorities higher than (lower than) that of τ_i. Similarly, we use 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) that of τ_i. We consider systems where each task has a unique priority.

Application tasks may arrive either periodically at fixed intervals of time, or sporadically after some minimum inter-arrival time has elapsed. Each task, is characterized by: its relative deadline D_i, worst-case execution time C_i, minimum inter-arrival time or period T_i and release jitter J_i, defined as the maximum time between the task arriving and it being released (ready to execute). It is assumed that once a task starts to execute it will never voluntarily suspend itself. The processor utilization U_i of task τ_i is given by C_i/T_i. The total utilization U of a taskset is the sum of the individual task utilizations. The worst-case response time R_i of a task τ_i, is the longest time from it becoming ready to execute to it completing execution. A task is referred to as schedulable if its worst-case response time is less than or equal to its deadline less release jitter ($R_i \leq D_i - J_i$). A taskset is referred to as schedulable if all of its tasks are schedulable.

In Section III and Section IV we assume that tasks are independent. In Section V, we relax this restriction, permitting tasks to access shared resources in mutual exclusion according to the Stack Resource Policy (SRP) [8]. As a result of the operation of the SRP, a task τ_i may be blocked by lower priority tasks for at most B_i, referred to as the blocking time.

In our analysis of cache related pre-emption delays, we use aff(i, j) to mean the set of tasks that can not only execute between the release and completion of task τ_i and so affect its response time, but can also be pre-empted by task τ_j. For the basic task model, without shared resources, aff(i, j) = hep(i) $\cap$ lp(j).

With respect to a given system model, a schedulability test is said to be *sufficient* if every taskset it deems to be schedulable is in fact schedulable. Similarly, a schedulability test is said to be *necessary* if every taskset it deems to be unschedulable is in fact unschedulable. Tests that are both sufficient and necessary are referred to as *exact*.

A schedulability test A is said to *dominate* another schedulability test B if all of the tasksets deemed schedulable by test B are also deemed schedulable by test A, and there exist tasksets that are schedulable according to test A but not according to test B. Schedulability tests A and B are said to be *incomparable* if there exists tasksets that are deemed schedulable by test A and not by test B and also tasksets that are deemed schedulable by test B and not by test A.

Preemption Costs

We now extend the sporadic task model introduced above 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 $g \times \text{BRT}$ 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. [22] 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 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 this bound can be improved by counting the j-th highest number of UCBs at 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:

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

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

A bound on the pre-emption cost due to task τ_j *directly* pre-empting τ_i is therefore given by $\text{BRT} \cdot |\text{UCB}_i \cap \text{ECB}_j|$. Precise computation is more complex as different program points may exhibit different sets of UCBs. Hence, the worst-case pre-emption delay considering a pre-empting and pre-empted task may not necessarily occur at the pre-emption point with the highest number of UCBs—see [3] for a detailed description of the computation of pre-emption costs. Note that the simplification we apply, using ECB_i and UCB_i, does not impact the correctness of the equations.

Note that a separate computation of the pre-emption cost is restricted to architectures without timing anomalies [24] but is independent of the type of cache used, i.e. data, instruction or unified cache.

Set-Associative Caches: In the case of set-associative LRU caches[1], a single cache-set may contain several useful cache blocks. For instance, $\text{UCB}_1 = \{1, 2, 2, 2, 3, 4\}$ 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, 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 $\text{UCB}_j \cap' \text{ECB}_i = \{m | m \in \text{UCB}_j : m \in \text{ECB}_i\}$, 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.

III. Response Time Analysis for pre-emptive Systems

Response time analysis [5, 20] for fixed priority pre-emptive scheduling calculates the worst-case response time R_i of task τ_i, using the following equation.

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

Note that the worst-case response time appears on both the LHS and the RHS of the equation. As the RHS is a monotonically non-decreasing function of R_i, the equation can be solved using fixed point iteration: Iteration starts with an initial value for the response time, typically $r_i^0 = C_i + B_i$, and ends either when $r_i^{n+1} = r_i^n$ in which case the worst-case response time R_i is given by r_i^n or when $r_i > D_i - J_i$ in which case the task is unschedulable. We note that (1) does not explicitly include pre-emption costs.

[1]The concept of UCBs and ECBs cannot be applied to FIFO or PLRU replacement policies as shown in [13].

A. Existing Analyses including pre-emption costs

Equation (1) can be extended by $\gamma_{i,j}$ representing the pre-emption cost due to each job of a higher priority *pre-empting* task τ_j executing within the worst-case response time of task τ_i [15]:

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

Note that task τ_j does not necessarily pre-empt task τ_i directly; a nested pre-emption is also possible. Any pre-emption by task τ_j of a task τ_k that executes while τ_i is pre-empted may also increase the response time of task τ_i. The problem of obtaining a valid yet tight upper bound on the pre-emption costs is made difficult by the effects of nested pre-emption, as a pre-empting task may evict useful cache-blocks belonging to a number of pre-empted tasks.

The precise meaning of $\gamma_{i,j}$ and its computation depends on the approach used. Below, we review a number of existing approaches and discuss their advantages and disadvantages.

ECB-Only

Busquets and Wellings [15] and later Tomiyama and Dutt [32], used the ECBs of the pre-empting task to bound the pre-emption costs:

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

In this case, $\gamma_{i,j}$ represents the worst-case effect of task τ_j on any arbitrary lower priority task, independent of such a task's actual UCBs.

UCB-Only

By contrast, Lee et al. [22] used the number of UCBs to bound the pre-emption costs. Here, however one has to correctly account for nested pre-emptions. The cost of τ_j pre-empting some task τ_k of intermediate priority may be higher than that of τ_j pre-empting τ_i. Thus, the pre-emption cost due to a job of task τ_j executing during the response time of task τ_i is only bounded by the maximum number of UCBs over all tasks that may be pre-empted by τ_j and have at least the priority of τ_i (i.e. tasks from the set $\text{aff}(i, j) = \text{hep}(i) \cap \text{lp}(j)$).

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

The disadvantage of the ECB-Only and UCB-Only approaches is clear: considering only the pre-empted tasks *or* alternatively only the pre-empting tasks leads to an over-approximation. Not every UCB may be evicted during pre-emption, and not every ECB may evict a UCB. This is illustrated in Figure 1.

The LHS of Figure 1 shows the scenario leading to the worst-case number of cache reloads due to pre-emption, the RHS shows the set of cache sets accessed by each task (ECB) and the set of cache sets that may contain a useful cache block (UCB). Note the example assumes a 4-set cache memory.

Figure 1 shows an example taskset that leads to an overestimation when the pre-emption cost is estimated using (3) or

(4). Task τ_1 accesses blocks in cache sets 1 and 2. Task τ_2 accesses blocks in cache sets 1, 2, 3 and 4. However, only sets 3 and 4 may contain useful cache blocks, hence a pre-emption of task τ_2 by task τ_1 never evicts any useful cache blocks; and so there are no cache reloads due to pre-emption. However, (4) and (3) account for 2 additional reloads; an overestimation of the pre-emption cost.

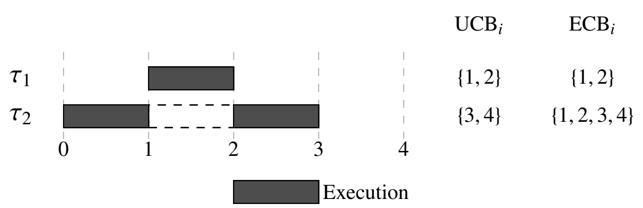
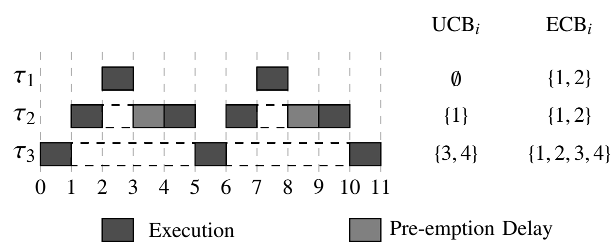
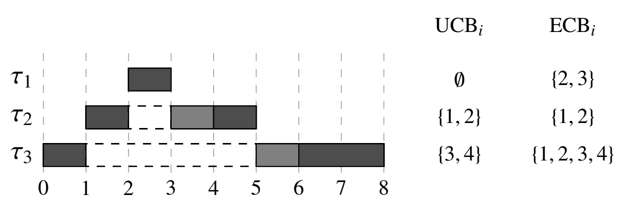


Fig. 1. Taskset $\{\tau_1, \tau_2\}$ with $C_1 = 1$, $C_2 = 2$ and block reload time 1. Response time analysis of task τ_2: only counting the number of possibly evicted UCBs (4) or possibly evicting ECBs (3) leads to a pre-emption cost of 2, whereas the actual pre-emption cost is 0.

Since both (3) and (4) can over-estimate the actual pre-emption cost, combining both UCBs and ECBs might be expected to result in precise bounds. However, the naive computation $\gamma_{i,j} = \text{BRT} \cdot |\text{UCB}_i \cap \text{ECB}_j|$ is optimistic and thus cannot be used. It may lead to underestimation in two cases: when the cost of task τ_j pre-empting a task τ_k of intermediate priority is higher than that of τ_j pre-empting τ_i (see Figure 2(a)) and when the execution of τ_j may evict useful cache blocks of both task τ_i and of task τ_k (see Figure 2(b)).



(a) τ_1 pre-empting τ_2 causes higher costs than τ_1 pre-empting τ_3.



(b) Nested pre-emption: τ_1 pre-empting τ_2 pre-empting τ_3, causes higher costs than any non-nested pre-emption.

Fig. 2. Two taskets $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = 2$, $C_3 = 3$, and a block reload time of 1.

UCB-Union

Tan and Mooney [31] considered both the pre-empted and the pre-empting task. They take the union of all possible affected useful cache blocks and combine this with the set of ECBs of the pre-empting task.

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

This UCB-Union approach dominates the ECB-only approach since:

$$\gamma_{i,j}^{\text{ecb}} \geq \gamma_{i,j}^{\text{tan}}$$

but may be worse than the UCB-only approach in some cases. Consider the taskset shown in Figure 3, the values of $\gamma_{i,j}$ for the response time analysis of task τ_3 are as follows:

$$\gamma_{3,1}^{\text{tan}} = |\{(\text{UCB}_2 \cup \text{UCB}_3) \cap \text{ECB}_1\}| = |\{1,2,3,4\} \cap \{1,2,3,4\}| = 4$$

$$\gamma_{3,2}^{\text{tan}} = |\{(\text{UCB}_3) \cap \text{ECB}_2\}| = |\{3,4\} \cap \{1,2,3,4\}| = 2$$

Given that each task is executed at most once, the total computed pre-emption cost is 6. However, the actual pre-emption cost is only 4: either UCBs in cache sets $\{1,2,3,4\}$ have to be reloaded (in the case of nested pre-emption) or UCBs in cache sets $\{3,4\}$ are reloaded twice (in the case of consecutive pre-emption of τ_3 by τ_1 and then by τ_2).

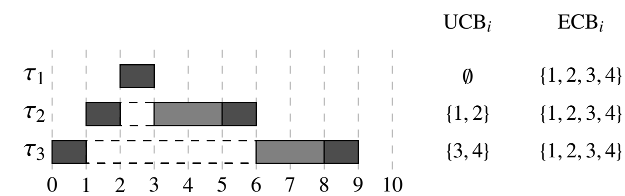


Fig. 3. Taskset $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = C_3 = 2$, and a block reload time of 1. Equation (5) computes total pre-emption costs of 6, whereas the actual cost is only 4.

Note that in the case of set-associative caches, Tan and Mooney [31] account only for those cache blocks that are actually evicted due to pre-emption. We note that this can be optimistic, as shown in [13].

Staschulat's Formula

Staschulat et al. [30] also combine information about the pre-empting and the pre-empted task. Their analysis is extended to account for the fact that each additional pre-emption of task τ_i may result in a smaller pre-emption cost than the last. (Their approach is an improvement over that of Petters and Färber [26]). The basic response time analysis used differs from (2): $\gamma_{i,j}$ does not refer to the cost of a single pre-emption, but instead to the total cost of all pre-emptions due to jobs of task τ_j executing within the response time of task τ_i.

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left(\left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j + \gamma_{i,j}^{\text{sta}} \right) \quad (6)$$

Staschulat et al. compute the maximum number of pre-emptions q, including nested pre-emptions, which may impact the response time of task τ_i due to cache blocks evicted by task τ_j. Thus q is given by the sum of the maximum number of jobs of task τ_j and tasks of lower priority than τ_j but higher priority than τ_i that can execute during the response time R_i of task τ_i

$$q = \sum_{\forall k \in \mathrm{hp}(i) \cap (\mathrm{lp}(j) \cup \{j\})} E_k(R_i) \tag{7}$$

where $E_k(R_i)$ is used to denote the maximum number of jobs of task τ_k that can execute during response time R_i. For our task model, $E_k(R_i) = \lceil (R_i + J_k)/T_k \rceil$. The total pre-emption cost $\gamma^{\mathrm{sta}}_{i,j}$ due to jobs of task τ_j pre-empting during the response time of task τ_i is then bounded by the q largest costs of task τ_j pre-empting jobs of any lower priority task $\tau_k \in \mathrm{hep}(i) \cap \mathrm{lp}(j)$ that can execute during the response time of task τ_i. As each job of such a task τ_k may execute up to $E_k(R_i)$ times during R_i, and each of those jobs could potentially be pre-empted at most $E_j(R_k)$ times by task τ_j, the $E_j(R_k)$ highest pre-emption costs of τ_j directly pre-empting τ_k must be considered $E_k(R_i)$ times:

$$\gamma^{\mathrm{sta}}_{i,j} = \mathrm{BRT} \cdot \sum_{l=1}^{q} |M^l| \tag{8}$$

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

$$M = \bigcup_{k \in \mathrm{hep}(i) \cap \mathrm{lp}(j)} \left(\bigcup_{E_k(R_i)} \left\{ (\mathrm{UCB}_k \cap \mathrm{ECB}_j)^n | n \in [1; E_j(R_k)] \right\} \right) \tag{9}$$

and $(\mathrm{UCB}_k \cap \mathrm{ECB}_j)^n$ gives the n-th highest pre-emption cost for task τ_j pre-empting task τ_k. Note that M is a multiset and the union over $E_k(R_i)$ means that the set of values for τ_k are repeated $E_k(R_i)$ times.

The drawback of this approach is that the number of pre-emptions taken into account strongly over-estimates the number of pre-emptions that have an actual influence on the response time; particularly when there are a large number of tasks. In addition, the reduction in the pre-emption costs for a sequence of pre-emptions is typically rather limited ([11] shows that the maximal pre-emption cost can occur at various program points within a task's execution). The program point $\mathcal{P}$ in a task which exhibits the highest number of UCBs often occurs within a loop, thus, it has to be taken into account as often as the loop iterates. In addition, program points close to $\mathcal{P}$ will often have a similar number of UCBs. We note that Staschulat et al. also present an improvement to their analysis in [30]; however, the problem of strongly over-estimating the number of pre-emptions remains.

IV. ECB-Union Approach

We now introduce a new *ECB-Union* approach to computing pre-emption costs. To account for nested pre-emptions, we compute the union of all ECBs that may affect a pre-empted task. The intuition here is that direct pre-emption by task τ_j is represented by the pessimistic assumption that task τ_j has itself already been pre-empted by all of the tasks of higher priority and hence may result in eviction of $\bigcup_{h \in \mathrm{hp}(j) \cup \{j\}} \mathrm{ECB}_h$

$$\gamma^{\mathrm{new}}_{i,j} = \mathrm{BRT} \cdot \max_{\forall k \in \mathrm{aff}(i,j)} \left\{ \left| \mathrm{UCB}_k \cap \bigcup_{h \in \mathrm{hp}(j) \cup \{j\}} \mathrm{ECB}_h \right| \right\} \tag{10}$$

Task τ_j may directly pre-empt any task $\tau_k \in \mathrm{aff}(i,j)$ impacting the response time of task τ_i. Thus taking the maximum over all of the tasks in $\mathrm{aff}(i,j)$ ensures that the pre-emption cost for the highest number of evicted useful cache blocks is considered. Note we use $\mathrm{hp}(j) \cup \{j\}$ to mean task τ_j and all tasks of higher priority than task τ_j, rather than $\mathrm{hep}(j)$. This is because the two sets are different in the more general case where tasks can share priority levels, see Appendix A for further details. Note that (10) is combined with (2) to determine task response times.

The ECB-Union approach (10) dominates the UCB-only approach, since:

$$\gamma^{\mathrm{ucb}}_{i,j} \geq \gamma^{\mathrm{new}}_{i,j}$$

The ECB-Union approach is incomparable with the UCB-Union approach [31]. Figure 3 provides an example where the ECB-Union approach outperforms the UCB-Union approach: here the ECB-Union approach covers both a nested pre-emption (τ_3 pre-empted by τ_2 which is pre-empted by τ_1) and consecutive pre-emption (of τ_3 by τ_1 and τ_2), obtaining for each pre-emption a cost of 2 and thus, a total cost of 4. In contrast, the UCB-Union approach gives a total cost of 6.

$$\begin{aligned}
\gamma^{\mathrm{new}}_{3,1} &= \max_{\forall k \in \{2,3\}} \{|\mathrm{UCB}_k \cap \mathrm{ECB}_1|\} \\
&= \max\{|\mathrm{UCB}_2 \cap \mathrm{ECB}_1|, |\mathrm{UCB}_3 \cap \mathrm{ECB}_1|\} \\
&= \max\{|\{1,2\}|, |\{3,4\}|\} = 2 \\
\gamma^{\mathrm{new}}_{3,2} &= \max_{\forall k \in \{3\}} \{|\mathrm{UCB}_k \cap (\mathrm{ECB}_1 \cap \mathrm{ECB}_2)|\} \\
&= |\mathrm{UCB}_3 \cap (\mathrm{ECB}_1, \mathrm{ECB}_2)| \\
&= |\{3,4\} \cap \{1,2,3,4\}| = |\{3,4\}| = 2
\end{aligned}$$

Figure 4 provides an example where the UCB-Union approach outperforms the ECB-Union approach. Here, for the latter approach, the pre-emption costs increasing the response time R_3 of task τ_3 are computed as follows:

$$\begin{aligned}
\gamma^{\mathrm{new}}_{3,1} &= \max_{\forall k \in \{2,3\}} \{|\mathrm{UCB}_k \cap \mathrm{ECB}_1|\} \\
&= \max\{|\mathrm{UCB}_2 \cap \mathrm{ECB}_1|, |\mathrm{UCB}_3 \cap \mathrm{ECB}_1|\} \\
&= \max\{|\emptyset|, |\{1,2\}|\} = 2 \\
\gamma^{\mathrm{new}}_{3,2} &= \max_{\forall k \in \{3\}} \{|\mathrm{UCB}_k \cap (\mathrm{ECB}_1 \cap \mathrm{ECB}_2)|\} \\
&= |\mathrm{UCB}_3 \cap (\mathrm{ECB}_1, \mathrm{ECB}_2)| \\
&= |\{1,2,3,4\} \cap \{1,2,3,4\}| = |\{1,2,3,4\}| = 4
\end{aligned}$$

With the ECB-Union approach, the eviction of UCBs of task τ_3 ($\{1,2\}$) are considered twice, even though they must be reloaded at most once, leading to an over-estimation of the total pre-emption costs of 6. The UCB-Union approach, in this case, computes the precise total of 4.

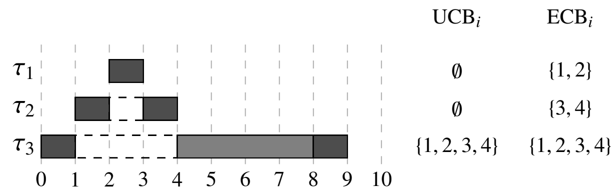


Fig. 4. Taskset $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = C_3 = 2$, and block reload time 1. Equation (10) computes a total pre-emption cost of 6, whereas the actual cost is only 4.

A. Combined Approach

The UCB-Union approach dominates the ECB-Only approach, similarly the ECB-Union approach dominates the UCB-Only approach. Given that the UCB-Union approach (5) and the ECB-Union approach (10) are incomparable, we can combine both to deliver a more precise bound on task response times that, by construction, dominates the use of either approach alone:

$$R_i = \min(R_i^{\tan}, R_i^{\text{new}}) \qquad (11)$$

where $R_i^{\tan}$ is the response time of task τ_i computed using (5) and R_i^{new} is the response time of task τ_i computed using (10).

V. Blocking Time

The discussion in Section III and Section IV assumes non-blocking execution, i.e. no shared resources. In this section, we relax this restriction, permitting tasks to access mutually exclusive shared resources according to the Stack Resource Policy (SRP) introduced by Baker [8], extending the Priority Ceiling Protocol of Sha et al. [29].

The SRP associates a *ceiling* priority with each resource. This ceiling priority is equal to the highest priority of any task that can access the resource. At run-time, when a task accesses a resource, its priority is immediately increased to the ceiling priority of the resource. Thus SRP bounds the amount of blocking B_i which task τ_i is subject to, to the maximum time for which any lower priority task holds a resource that is shared with task τ_i or any other task of equal or higher priority. SRP ensures that a task can only ever be blocked prior to actually starting to execute.

We note that when a lower priority task τ_k locks a resource and so blocks task τ_i, it can still be pre-empted by tasks with priorities higher than that of the ceiling priority of the resource. B_i does not account for the additional pre-emption cost due to such pre-emptions.

Previous work integrating pre-emption costs into response time analysis [15, 22, 30, 31] extend (2) to include blocking via the simple addition of the blocking factor B_i:

$$R_i = C_i + B_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j + \gamma_{i,j}) \qquad (12)$$

In the case of Busquets and Wellings analysis [15], this is correct, as the pre-emption cost is accounted for only via the ECBs of the pre-empting tasks and is therefore unaltered by the addition of resource accesses that could potentially also be pre-empted. In contrast, [22, 30, 31] make use of the UCBs of pre-empted tasks. If, as is the case with the SRP, pre-emption can still occur during resource access, then these analyses are optimistic and need to be modified to correctly account for the additional pre-emption costs that can occur[2]. The key point is that the blocking factor B_i does not represent execution of task τ_i, but instead represents execution of some resource access within a lower priority task. Such a resource access may be pre-empted, during the response time of task τ_i and therefore its UCBs need to be taken into account, as illustrated by the example in Figure 5.

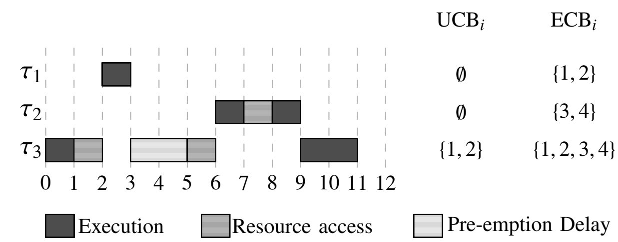


Fig. 5. Tasks τ_2 and τ_3 share a common resource x, τ_3 starts to execute, blocks τ_2, which is released at time 1, and is pre-empted by τ_1. Thus, the finishing time of τ_2 is delayed not only by the time for which τ_3 accesses the resource, but also by the additional pre-emption delay, reloading UCBs $\{1, 2\}$ after the resource access of task τ_3 is pre-empted by task τ_1.

We now extend the ECB-Union and UCB-Union approaches to take account of blocking. Specifically, we extend the pre-emption cost equations (10) and (5) to include the UCBs of tasks in the set $b(i, j)$, where $b(i, j)$ is defined as the set of tasks with priorities lower than that of task τ_i that lock a resource with a ceiling priority higher than or equal to the priority of task τ_i but lower than that of task τ_j. These tasks can block task τ_i, but can also be pre-empted by task τ_j. Hence they need to be included in the set of tasks $\text{aff}(i, j)$ whose UCBs are considered when determining the pre-emption cost $\gamma_{i,j}$ due to task τ_j:

$$\text{aff}(i, j) = (\text{hep}(i) \cap \text{lp}(j)) \cup b(i, j) \qquad (13)$$

Note that the tasks in $b(i, j)$ have lower priorities than task τ_i and so cannot pre-empt during the response time of task τ_i, hence their ECBs do not need to be considered when computing $\gamma_{i,j}$. Using (13) extends the ECB-Union approach (10) and the UCB-Union approach (5) to correctly account for pre-emption costs when tasks share resources according to the SRP.

Revisiting the example given in Figure 5, we observe that as the set of affected tasks $\text{aff}(2, 1)$ now includes task τ_3 as well as task τ_2, (5) correctly accounts for the overall pre-emption cost of 2 due to the resource access of task τ_3 being pre-empted by task τ_1 during the response time of task τ_2.

We note that in the simplest case of the SRP where tasks

[2] If all resource accesses are non-pre-emptive, then there are no additional pre-emption costs to be accounted for.

share resources that are accessed non-pre-emptively (i.e. with ceiling priorities equal to that of the highest priority task), then the set of tasks $b(i, j)$ is empty (since no task can pre-empt during a resource access) and hence the pre-emption cost $\gamma_{i,j}$ is the same as for the basic task model, with no increase in pre-emption costs due to blocking.

Although providing valid upper bounds on the pre-emption costs, the above extension can be pessimistic. This is because it includes the UCBs of each lower priority task in aff(i, j), rather than just the UCBs of each resource access within those tasks. More precise analysis can be obtained by considering each resource access as a sub-task with its own UCBs, as explained in the following sub-section.

When determining the blocking factor B_i we cannot use the resource access execution times as they occur within the non-pre-emptive execution of each containing task τ_k. This is because we must assume that task τ_k could be pre-empted immediately before a resource access and any useful cache blocks evicted. Instead, the execution time of each resource access must be determined assuming execution of that section of code with no pre-emption, and starting from the worst-case initial state.

Finally, we note that it is possible for the SRP to cause push-through or indirect blocking where a resource access delays execution of a higher priority task which then executes during the response time of task τ_i; however, such behaviour cannot increase the overall response time of τ_i beyond that calculated assuming direct blocking. This is because indirect blocking removes the possibility of resource access pre-emption, without including the possibility that any additional tasks pre-empt or are pre-empted, as all other candidates for pre-emption are already included in the calculations.

A. Refined analysis based on sub-tasks

We now provide more precise analysis of the pre-emption costs when tasks share resources according to the SRP. We do so by considering any resource access made by a task τ_k, that results in an increase in priority[3], as a sub-task of τ_k executing at that higher priority. Thus $S_x^{k,h}$, is a resource access sub-task of task τ_k that increases the priority of task τ_k to that of task τ_h, where task τ_h is the highest priority task that accesses the same resource, and x is an arbitrary index of resource accesses. The worst-case execution time of sub-task $S_x^{k,h}$ is denoted by $C_x^{k,h}$. The value of $C_x^{k,h}$, is determined assuming stand-alone execution of $S_x^{k,h}$, i.e. with no pre-emption and an unknown initial cache state. We denote the useful cache blocks of sub-task $S_x^{k,h}$, by $\text{UCB}_x^{k,h}$, again determined by considering stand-alone execution.

Under SRP, the maximum blocking time B_i that task τ_i can be subject to is given by:

$$B_i = \max_{\forall S_x^{k,h}: k \in \text{lp}(i) \wedge h \in \text{hep}(i)} (C_x^{k,h}) \tag{14}$$

The ceiling priority that each sub-task $S_x^{k,h}$ executes at ensures mutually exclusive resource access; however, higher priority tasks ($\in$ hp(h)) can still pre-empt $S_x^{k,h}$. B_i does not account for the additional cache related pre-emption cost of such a pre-emption.

We now extend the ECB-Union and the UCB-Union [31] approaches to take account of both blocking, and the additional pre-emption costs which may occur when a resource access sub-task is pre-empted. Specifically, we extend the pre-emption cost equations (5) and (10) to include the UCBs of sub-tasks in the set $b(i, j)$, where $b(i, j)$ is defined as the set of sub-tasks each belonging to a task of priority lower than that of task τ_i that lock a resource with a ceiling priority higher than or equal to the priority of task τ_i but lower than that of task τ_j .

$$b(i, j) = \{\forall s_x^{k,h} : k \in \text{lp}(i) \wedge h \in \text{hep}(i) \cap \text{lp}(j)\} \tag{15}$$

These sub-tasks can block task τ_i , but can also be pre-empted by task τ_j, hence they need to be included in the set of tasks aff(i, j) whose UCBs are considered when determining the pre-emption cost $\gamma_{i,j}$:

$$\text{aff}(i, j) = (\text{hep}(i) \cap \text{lp}(j)) \cup b(i, j) \tag{16}$$

Note that the tasks in $b(i, j)$ cannot pre-empt any task in the set hep(k) (hep$(k) \supset$ hep(i)), hence they cannot pre-empt during the response time of task τ_i and so their ECBs do not need to be considered when computing $\gamma_{i,j}$.

Using (16) extends the ECB-Union approach (8) and the UCB-Union approach [31] (5) to account for the additional pre-emption costs when tasks share resources according to the SRP. Because this analysis includes the UCBs of sub-tasks (e.g. $s_x^{k,h}$) rather than tasks (e.g. τ_k) and $\text{UCB}_x^{k,h} \subseteq \text{UCB}_k$, it dominates the earlier analysis based on task rather than sub-task UCBs. The above analysis, while an improvement, is still pessimistic in that it independently maximizes both (i) the blocking factor due to all of the potentially blocking resource access sub-tasks, and (ii) the pre-emption costs due to pre-emption of any of those sub-tasks.

In practice the SRP ensures that any job of task τ_i can only be blocked by a single resource access sub-task. An alternative and more precise approach is therefore to evaluate the response time $R_{x,i}$ of task τ_i for every potentially blocking sub-task $s_x^{k,h} : k \in \text{lp}(i) \wedge h \in \text{hep}(i)$ individually; with the blocking factor given by $B_{i,x} = C_x^{k,h}$ and the pre-emption delay given by:

$$\gamma_{i,j,x}^{help-1} = \begin{cases} \left(\left| \text{UCB}_x^{k,h} \cap \bigcup_{m \in \text{hp}(j) \cup \{j\}} \text{ECB}_m \right| \right) & \text{if } h \in \text{lp}(j) \\ 0 & \text{otherwise} \end{cases}$$

[3] Under SRP, resource accesses may be nested. An inner nested access does not result in an increase in task priority if the outer nested access has a higher ceiling priority. A resource access by the highest priority task that uses a particular resource similarly does not result in an increase in priority. Resource accesses that do not change the priority of a task can be safely ignored in response time analysis.

$$\gamma_{i,j,x}^{help-2} = \max_{\forall k \in \text{aff}(i,j)} \left\{ \left| \text{UCB}_k \cap \bigcup_{m \in \text{hp}(j) \cup \{j\}} \text{ECB}_m \right| \right\}$$

$$\gamma_{i,j}^{\text{new}} = \max \left\{ \gamma_{i,j,x}^{help-1}, \gamma_{i,j,x}^{help-2} \right\} \quad (17)$$

(where aff(i, j) does not include $b(i, j)$) and then take the maximum response time obtained over all of the blocking sub-tasks:

$$R_i = \max_{\forall S_x^{k,h} : k \in \text{lp}(i) \wedge h \in \text{hep}(i)} \{R_{i,x}\} \quad (18)$$

VI. Case Study

In this section, we evaluate the effectiveness of the different approaches based on a case study. The worst-case execution times and the set of useful cache blocks and evicting cache blocks have been derived from the Mälardalen benchmark suite[4], see Table I, where the values are taken from [3]. The target architecture is an ARM7 processor[5] with direct-mapped instruction cache of size 4kB with a line size of 8 Bytes (and thus, 256 cache sets) and a block reload time of $8\mu s$. The ARM7 features an instruction size of 4 Bytes.

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

TABLE I
Execution times and number of UCBs and ECBs for a selection of benchmarks from the Mälardalen Benchmark Suite.

We note that although the case study tasks do not represent a set of tasks scheduled on an embedded real-time system, they do represent typical components of real-time applications and thus deliver meaningful values. We created a taskset from the above data by assigning periods and implicit deadlines such that all 15 tasks had equal utilization[6]. The periods where generated by multiplying each execution time by a constant c ($\forall_i : T_i = c \cdot C_i$). We varied c from 15 upwards hence varying the utilization of the taskset from 1.0 downwards. The tasks were assigned priorities in deadline monotonic priority order[7].

Table II lists the breakdown utilization; the maximum utilization at which a scaled version of the case study taskset was deemed schedulable by each approach.

Analysis	Breakdown utilization:
No Pre-emption Cost	0.95
Combined	0.767
ECB-Union	0.767
UCB-Only	0.75
UCB-Union	0.698
ECB-Only	0.612
Staschulat	0.508

TABLE II
Case study taskset: breakdown utilization for different approaches.

Staschulat's approach performs worst, with a breakdown utilization of 0.508. Equation (7) computes the number of pre-emptions taken into account. For the effect of task τ_1 (bs) to task τ_5 (insertsort), only the 8 highest costs of τ_1 pre-empting any task from τ_2 to τ_5 need to be considered. However, for the effect of task τ_1 (bs) to task τ_{15} (bsort100), the 47362 highest costs need to be considered. Although the single pre-emption costs (for τ_i pre-empted by τ_j) are much smaller, the total cost is very pessimistic.

The ECB-Union approach and the UCB-only approach perform best, with breakdown utilizations of 0.767 and 0.75. As the cache contention is high (3 out of the 15 tasks fill the whole cache), a single pre-emption often evicts all of the UCBs of the pre-empted task(s). In addition, the total number of ECBs is much higher than the total number of UCBs hence the ECB-only approach (3) is much more pessimistic than the UCB-only approach (4) and so has a lower breakdown utilization of 0.612. As a consequence, the ECB-Union approach (10) outperforms the UCB-Union approach (5) which has a breakdown utilization of 0.698. The combination of both approaches (11) does not improve upon the ECB-Union approach. Finally, (1) deems the case study taskset schedulable up to a utilization of 0.95 ignoring pre-emption costs.

VII. Evaluation

In this section, we evaluate the effectiveness of the different approaches to pre-emption cost computation on a large number of tasksets with varying cache configurations and varying taskset parameters. 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 [12] algorithm.
- Task periods were generated according to a log-uniform distribution with a factor of 100 difference between the minimum and maximum possible task period and a minimum period of 5ms. This represents a spread of task

[4]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[5]http://www.arm.coms/products/CPUs/families/ARM7Family.html
[6]This is an entirely arbitrary choice. Evaluation with randomly generated taskset parameters is reported in section VII.
[7]Deadline monotonic priority order is optimal in this case only when pre-emption costs are zero.

periods from 5ms to 500ms, as found in most automotive and aerospace hard real-time applications.

- Task execution times were set based on the utilization and period selected: $C_i = U_i \cdot T_i$.
- Task deadlines were implicit[8], i.e., $D_i = T_i$.
- Priorities were assigned in deadline monotonic order.

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 [12] algorithm (for a total cache utilization $CU = 10$). In such a case, 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.

Staschulat's approach exploits the fact that for the i-th pre-emption only the i-th highest number of UCBs has to be considered. As our case study and other measurements [11] have shown, a significant reduction typically only occurs at a high number of pre-emptions. For the purposes of evaluation, for Staschulat's approach, we simulated what in practice is likely to be an optimistic reduction: reducing the number of UCBs per pre-emption by one each time.

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 valid tasksets were generated and the schedulability of those tasksets determined using the appropriate pre-emption cost computation integrated into response time analysis.

A. *Base configuration*

We conducted experiments varying the number of tasks, the cache-size (i.e. number of cache-sets (CS)), the block reload time (BRT), the cache utilization (CU) and the reuse factor (RF). As a base configuration we used the default values of 10 tasks, a cache of 256 cache-sets, a block-reload time of $8\mu s$, a reuse factor of 30% and a cache-utilization of 10. The latter two parameters were chosen according to the actual values observed in the case-study. Figure 6 illustrates the performance of the different approaches for this base configuration. The graph also shows a line marked Simulation-UB. This refers to the use of simulation to form a necessary schedulability test. We simulated execution and pre-emption of the tasks starting from near simultaneous release. (The tasks were released in order, lowest priority first, to increase the number of pre-emptions considered). If any task missed its deadline, then the taskset was proven to be unschedulable w.r.t. the pre-emption cost model used[9], thus providing a valid upper bound on taskset schedulability including pre-emption costs. Note that the lines on the graphs appear in the same order as they are described in the legend. The graphs are best viewed online in colour.

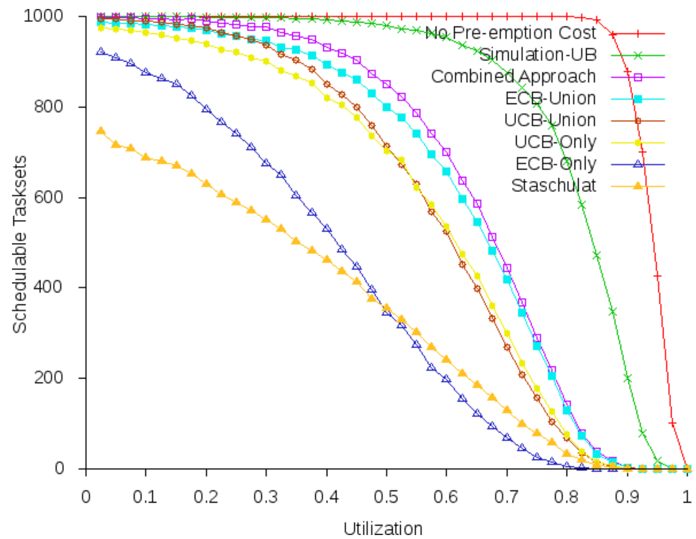


Fig. 6. Evaluation of base configuration. Number of tasksets deemed schedulable at the different total utilizations.

For each approach, we determined the average breakdown utilization for the tasksets generated for the base configuration, see Table III. These results show that the ECB-Union, and Combined approaches significantly improve upon the performance of previous methods.

Analysis	Average Breakdown Utilization:
No Preemption Cost	0.93
Combined	0.64
ECB-Union	0.62
UCB-Union	0.57
UCB-Only	0.55
ECB-Only	0.39
Staschulat	0.35

TABLE III
AVERAGE BREAKDOWN UTILIZATION OF BASE CONFIGURATION TASKSETS FOR EACH APPROACH.

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 graphs below show the weighted schedulability measure $W_y(p)$ [9] for schedulability test y as a function of parameter p. For each value of p, this measure combines data for all of the tasksets τ generated for all of a set of equally

[8] Evaluation for constrained deadlines, i.e., $D_i \in [2C_i; T_I]$ gives broadly similar results although fewer tasksets are deemed schedulable by all approaches, see Appendix B).

[9] The simulation assumed that any partial execution of a task evicted all its ECBs and used all its UCBs

spaced utilization levels. Let $S_y(\tau, p)$ be the binary result (1 or 0) of schedulability test y for a taskset τ and parameter value p then:

$$W_y(p) = (\sum_{\forall\tau} u(\tau) \cdot S_y(\tau, p)) / \sum_{\forall\tau} u(\tau) \qquad (19)$$

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 [9]. Weighting the individual schedulability results by taskset utilization reflects the higher value placed on being able to schedule higher utilization tasksets.

B. Cache Utilization & Cache-Reuse

Cache utilization and cache-reuse are the most important factors for pre-emptively scheduled systems. If all tasks fit into the cache, i.e. the cache utilization is less than one or there is no cache-reuse at all, then no additional cache-related pre-emption delays occur. The other extreme is when each task completely fills the cache. In this case, each UCB must be assumed to be evicted, and hence the overall pre-emption delay depends solely on the number of UCBs.

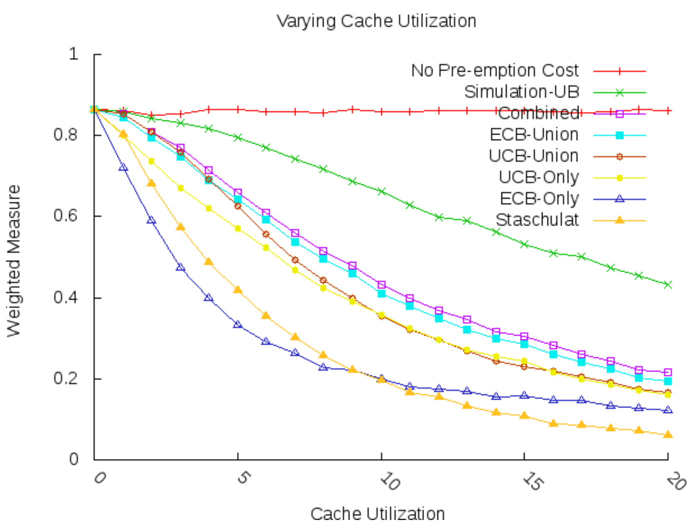


Fig. 7. Weighted schedulability measure; varying cache utilization from 0 to 20, in steps of 2

Figure 7 shows the weighted schedulability measure for each approach as a function of the cache utilization. At a low cache utilization, only a few UCBs are actually evicted. The set of ECBs per task is low, and often smaller than the number of UCBs of all possibly pre-empted tasks. Thus, an upper bound on the possibly evicted UCBs per pre-empting task (as computed by the UCB-Union approach) is slightly pessimistic, while the ECB-Union approach is in this case more pessimistic. The situation changes with increased cache utilization. As each task uses a larger proportion of the whole cache on average, the UCB-Union approach becomes significantly more pessimistic than the ECB-Union approach.

Figure 8, shows the weighted schedulability measure for each approach as a function of the reuse factor. At low values of the reuse factor, the set of UCBs per task is low compared to the ECBs, and so the UCB-Union method is more pessimistic than the ECB-Union method, while at high values of the reuse factor, the opposite applies as the set of UCBs for each task becomes similar to its set of ECBs. Observe that in Figure 8 these two lines cross at a medium level of reuse, while the Combined approach outperforms both, providing the best performance in all cases. Since the reuse factor only affects the number of UCBs, the performance of the ECB-only approach is independent of the reuse factor. As expected, performance of the ECB-only approach is relatively poor at low levels of reuse, but competitive at high levels.

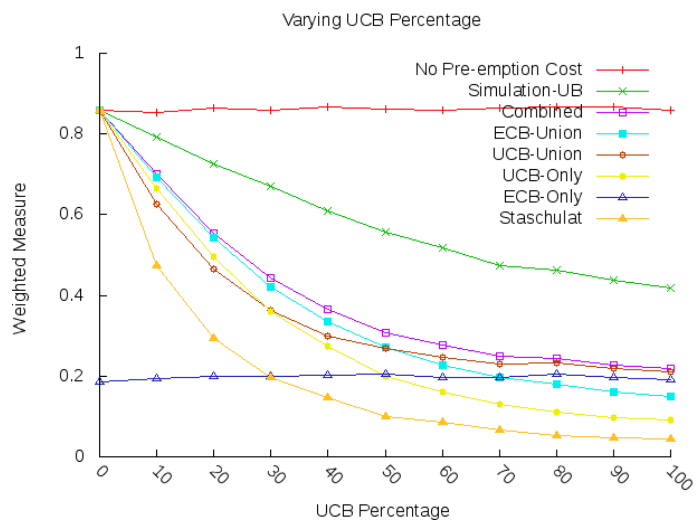


Fig. 8. Weighted schedulability measure; varying reuse factor from 0% to 100%, in steps of 10%

C. Number of Tasks

In this experiment, we varied the number of tasks with the other parameters fixed at their default values. Figure 9 shows

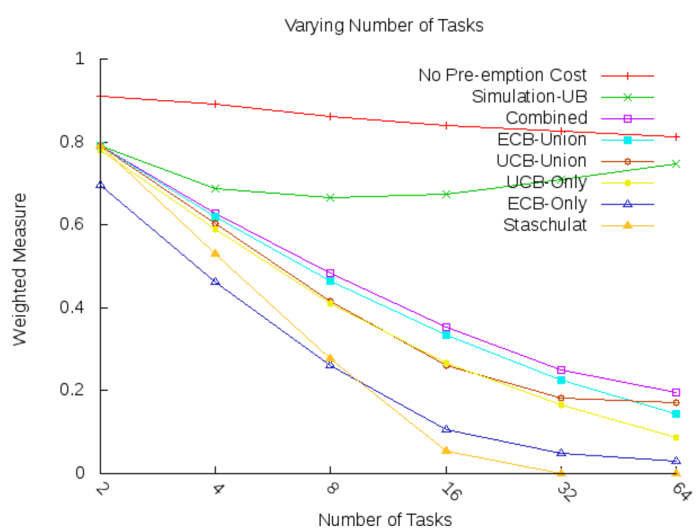


Fig. 9. Weighted schedulability measure; varying number of tasks from $2 = 2^1$ to $2^6 = 64$.

that the more tasks there are, the less likely a taskset of a given utilization is to be schedulable. This is because with an increased number of tasks the number of pre-emptions and hence the overall pre-emption costs increase, reducing the schedulability of the taskset[10]. This reduction in schedulability with increasing taskset size holds for all of the approaches, with a greater reduction observed with Staschulat's approach for the reasons explained in Section VI.

Note that the upper bound derived by simulation shows a much smaller reduction. This is because, as the number of tasks increases, the number of possible execution scenarios increases rapidly, thus it becomes less likely that the simulation will deliver the worst-case scenario.

D. Cache-Size

The number of cache-sets also has an influence on the overall performance of the different approaches. The more cache-sets there are, the higher the impact of a pre-emption may be, given the same cache utilization and block reload time. Hence as the number of cache sets is increased, all of the approaches show a similar decrease in schedulability with the exception of the basic response time analysis which does not include pre-emption costs, see Figure 10.

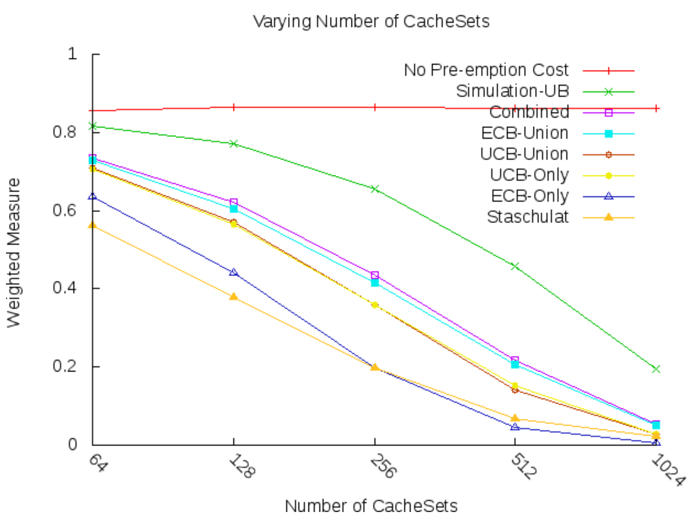


Fig. 10. Weighted schedulability measure; varying number of cache sets from $2^5 = 64$ to $2^{10} = 1024$

Varying the block reload time results in similar behaviour, see Figure 11.

E. Range of task periods

The range of task periods may also influence the performance of the different approaches. We therefore conducted experiments varying the task period generation. Our base configuration used task periods in the range 5ms to 500ms, typical of many real-time systems. In Figure (12), we varied the number of orders of magnitude v spanning the minimum to the maximum task period and hence the range of task periods and deadlines given by $5[1, 10^v]$ms. In Figure (12) we see that as the range of task periods is increased, by

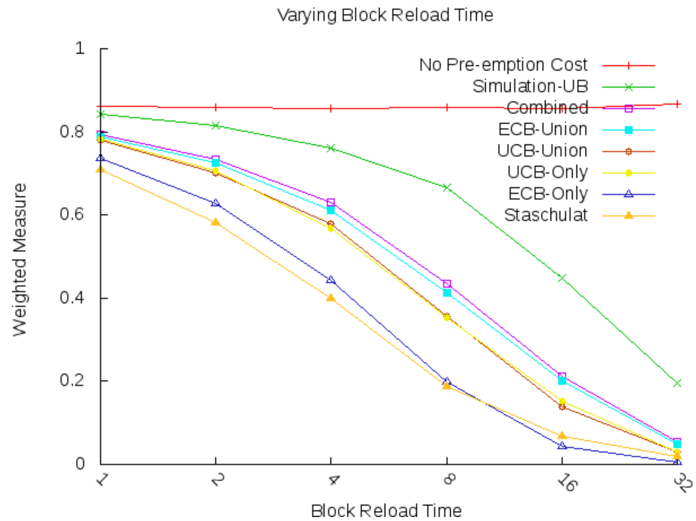


Fig. 11. Weighted schedulability measure; varying block reload time from $2^0 = 1\mu s$ to $2^4 = 32\mu s$

making the maximum period larger, schedulability improves for all of the approaches. This is because increasing the maximum period has the effect of reducing the proportion of task that are generated with smaller periods (e.g. in the range 1-10ms). Given that the block reload time is constant in this experiment, the ratio of pre-emption costs to taskset utilization reduces for increasing ranges of task periods, thus improving schedulability[11]. Note the smaller improvement with no pre-emption costs is a property of fixed priority scheduling; which on average can schedule higher utilization tasksets when there is a wide disparity in task periods.

In Figure (13), we varied the scaling factor w from 1 to 10 and hence the range of task periods given by $w[1, 100]$ms. Given that the block reload time is again 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, thus increasing schedulability for all of the approaches that include pre-emption costs. In fact these results are similar to the ones for varying block reload times, but with the results for larger values of the scaling factor w corresponding to those for smaller block reload times.

VIII. Conclusions

The major contribution of this paper is the introduction of a new method of bounding pre-emption costs, called the ECB-Union approach. This approach dominates the UCB-Only approach of Lee [22]. The ECB-Union approach complements the UCB-Union approach of Tan and Mooney [31], which dominates the ECB-only approach of Busquets and Wellings [15] and Tomiyama and Dutt [32]. The ECB-Union and UCB-Union approaches are incomparable and so we combined them into a composite response time test that dominates the use of either approach on its own.

We extended the ECB-Union and UCB-Union approaches to systems that permit tasks to access shared resources in mutual

[10] The pre-emption costs are not included in the taskset utilization

[11] Recall that pre-emption costs are not included in taskset utilization.

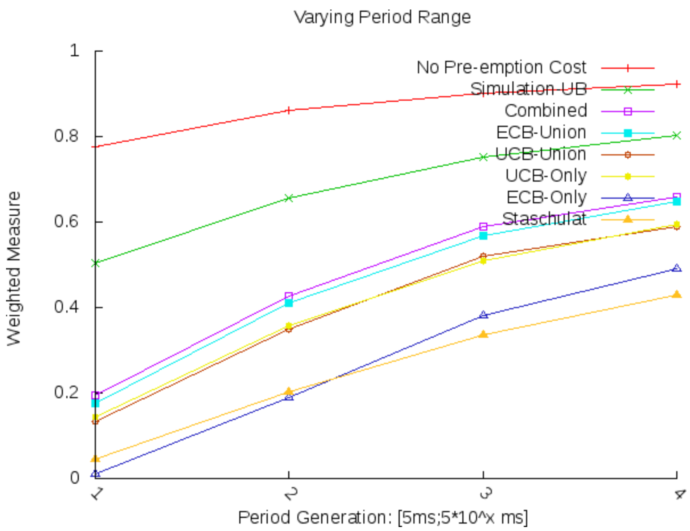


Fig. 12. Weighted schedulability measure; varying the range of task periods [5, 50] to $[5, 5 \cdot 10^4]$

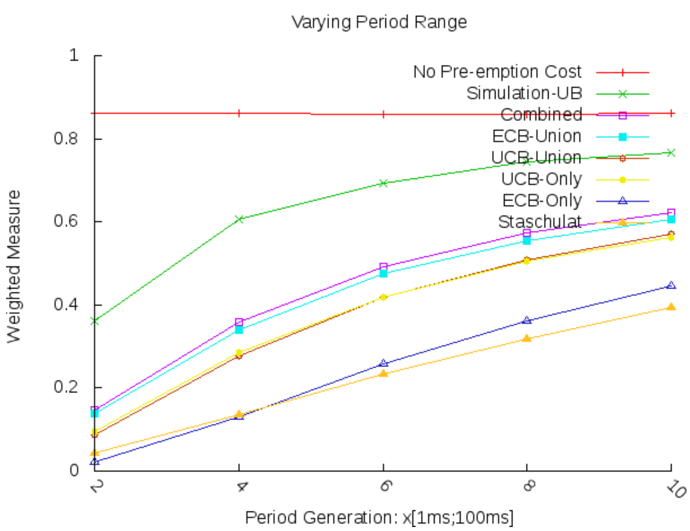


Fig. 13. Weighted schedulability measure; varying the scale of task periods $w[1, 100]$ from $w = 2$ to $w = 10$

exclusion according to the Stack Resource Policy. Our work in this area revealed that previous approaches to computing pre-emption delays, although including blocking factors in their schedulability analyses, did not account for the pre-emption of blocking tasks during a resource access. This omission can lead to optimistic (unsound) response times, an issue that we corrected.

We extended the ECB-Union and UCB-Union approaches to FP-FIFO scheduling. FP-FIFO scheduling typically requires far fewer priority levels than fixed priority pre-emptive scheduling with unique task priorities. Hence FP-FIFO scheduling can lead to significantly fewer pre-emptions, as well as a reduction in individual pre-emption costs, due to a reduction in the number of levels of nested pre-emption. Our results for FP-FIFO scheduling dominate those for fixed priority scheduling, as shown in Appendix A.

Finally, we examined the performance of the various approaches to computing pre-emption costs via a case study and an empirical evaluation of taskset schedulability. The latter showed that a combined response time analysis test using both the new ECB-Union approach derived in this paper, and the UCB-Union approach of Tan and Mooney [31] provides an effective method of determining task schedulability. This combined approach offers a significant improvement in performance over previous approaches for a wide range of different task and cache configurations, including cache utilization level, amount of reuse, cache size, and block reload times.

Acknowledgements

This research and collaboration came about as a result of the 1st Real-Time Scheduling Open Problems Seminar (RTSOPS 2010) [2]. This work was partially funded by the UK EPSRC funded Tempo project (EP/G055548/1), the Transregional Collaborative Research Center AVACS of the German Research Council (DFG) and the EU funded ArtistDesign Network of Excellence.

References

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

[2] S. Altmeyer and C. Burguière. Influence of the task model on the precision of scheduling analysis for preemptive systems. In *Proceedings RTSOPS*, pages 5–6, July 2010.

[3] S. Altmeyer and C. Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 2010.

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

[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] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, Dept. Computer Science, University of York, UK, 1991.

[7] N.C. Audsley. On priority assignment in fixed priority scheduling. Technical report, May 2001.

[8] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3:67–99, April 1991.

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

[10] M. Bertogna, G. C.. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *Proceedings ECRTS*, pages 251–260, 2010.

[11] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. C. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Proceedings ECRTS (to appear)*, 2011.

[12] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005. 10.1007/s11241-005-0507-9.

[13] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches—pitfalls and solutions. In *Proceedings WCET*, 2009.

[14] A. Burns, K. Tindell, and A. J. Wellings. Fixed priority scheduling with deadlines prior to completion. In *Proceedings of the 6th Euromicro Workshop on Real-Time Systems*, June 1994.
[15] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings RTAS*, pages 204–212, 1996.
[16] R.I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Real-Time Systems, Volume 47, Issue 1*, pages 1–40, 2010.
[17] R.I. Davis, S. Kollmann, V. Pollex, and F. Slomka. Controller area network (can) schedulability analysis with fifo queues. In *Proceedings ECRTS (to appear)*, 2011.
[18] R.I. Davis, N. Merriam, and N.J. Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings Work in Progress RTSS*, 2000.
[19] 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.
[20] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986.
[21] U. Keskin, R.J. Bril, and J.J. Lukkien. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *Proceedings Work-in-Progress Session ETFA*, 2010.
[22] 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.
[23] J.Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. In *Performance Evaluation 2(4)*, pages 237–250, 1982.
[24] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings RTSS*, page 12, 1999.
[25] S. Martin, P. Minet, and L. George. Non pre-emptive fixed priority scheduling with fifo arbitration: uniprocessor and distributed cases. Technical report, INRIA Rocquencourt, December 2007.
[26] S. M. Petters and G. Farber. Scheduling analysis with respect to hardware related preemption delay. In *In Workshop on Real-Time Embedded Systems*, 2001.
[27] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *Proceedings RTSS*, pages 212–224, 2006.
[28] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings RTSS*, 2002.
[29] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39:1175–1185, September 1990.
[30] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings ECRTS*, 2005.
[31] Y. Tan and V. Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *Trans. on Embedded Computing Sys.*, 6(1), 2007.
[32] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings CODES*, pages 67–71, 2000.
[33] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with pre-emption threshold. In *Proceedings RTCSA*, 1999.
[34] A. Zuhily and A. Burns. Optimality of (d-j)-monotonic priority assignment. *Information Processing Letters*, Number 103:247–250, 2007.

Appendix
A FP-FIFO Scheduling

In this appendix, we extend the ECB-Union and UCB-Union approaches to systems using FP-FIFO scheduling. With FP-FIFO scheduling a number of tasks may share the same priority level. Such tasks are scheduled on a FIFO basis and so cannot pre-empt each other; however, they can still be pre-empted by tasks of higher priority. For constrained deadline tasksets, (2) can be extended to the FP-FIFO case in a similar way to the analysis described in [25]:

$$R_i = \sum_{\forall e \in eq(i)} C_e + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j + \gamma_{i,j}) \qquad (20)$$

where $eq(i)$ is the set of tasks with the same priority as task τ_i. Equation (20) holds provided that all of the tasks are schedulable. In this case the maximum time that any task τ_i can spend in a FIFO queue is $D_i \leq T_i$, hence there can be at most one job of each task from the set $eq(i)$ in the FIFO queue for that priority level at any given time. Assuming that task τ_i is the last of these tasks gives the worst-case response time.

We now show how the equations giving the pre-emption cost $\gamma_{i,j}$, can be extended to the case of FP-FIFO scheduling.

Equation (5) (UCB-Union analysis) takes the union of all possible affected useful cache blocks and combines this with the set of evicted cache blocks of the pre-empting task. The union is therefore over the set of tasks that can be directly pre-empted by task τ_j and is given by $\text{aff}(i, j) = \text{hep}(i) \cap \text{lp}(j)$, in the case of fixed priority pre-emptive scheduling with no shared resources. We observe that $\text{aff}(i, j)$ represents exactly those tasks that can be pre-empted by task τ_j in the FP-FIFO case. The only difference is that $\text{hep}(i)$ now contains all of the tasks with priorities equal to that of task τ_i and all tasks of higher priority. Equation (5) therefore applies unchanged to FP-FIFO scheduling. Note that the second summation term in (20) iterates over all of the tasks of higher priority than task τ_i, even though some of these tasks may share priority levels, hence all possible pre-empting tasks are also considered. Equation (20) combined with (5) therefore extends the UCB-Union approach to FP-FIFO scheduling.

The ECB union approach is similarly extended to FP- FIFO scheduling. Equation (10) takes the maximum value over the UCBs of tasks that can affect the response time of task τ_i by being pre-empted by task τ_j. Again this set is given by $\text{aff}(i, j)$ which represents exactly those tasks that can be directly pre-empted by task τ_j in the FP-FIFO case. The union of ECBs in (10), over task τ_j and all higher priority tasks accounts for the evicted cache blocks due to pre-emption by task τ_j including the effects of nested pre-emption of task τ_j by tasks of higher priority. In the FP-FIFO case, this represents exactly the ECBs required. Note, we take care to use the set $\text{hp}(j) \cup \{j\}$ rather than $\text{hep}(j)$ as task τ_j cannot be pre-empted by tasks of the same priority. Instead, the impact of such tasks on the response time of task τ_i is accounted for when the second summation term in (20) iterates over those tasks. Equation (20) combined with (10) therefore extends the ECB union approach to FP-FIFO scheduling.

We note that in the case of both the ECB-Union and UCB-Union approaches, (20) is what is referred to as a *FIFO-symmetric* schedulability test [17]. This means that the response times computed for all tasks at the same priority level are equal. Hence, all of the tasks at a given priority level are schedulable provided they meet the tightest time constraint (deadline minus release jitter) of any of those tasks.

In FP-FIFO scheduled systems with shared resources, the Stack resource Policy can again be used to control resource access. In this case, the analysis of blocking given in Section V applies, including the addition of blocking tasks to the set $\text{aff}(i, j)$ and the extension of (20) via the addition of the blocking factor B_i. (The only difference is that the set of tasks $\text{hep}(i)$ can include other tasks with the same priority as task τ_i).

A. Priority assignment

The analysis presented in this paper is independent of the priority ordering used; however, in fixed priority scheduling appropriate priority assignment is key to obtaining a schedulable system. For the simple case of sporadic tasksets with constrained deadlines, with no blocking, no release jitter, and no pre-emption costs, Leung and Whitehead [23] showed that deadline monotonic priority ordering (DMPO) is optimal. With blocking according to the Stack Resource Policy and release jitter, but no pre-emption costs, then deadline minus jitter monotonic priority ordering (DJMPO) is optimal [34]. It is trivial; however, to construct examples with non-zero pre-emption costs where DJMPO is not optimal. For example, consider two tasks τ_A and τ_B with parameters $C_A = 5$, $D_A = T_A = 10$, and $C_B = 5$, $D_B = T_B = 11$. Further, let $\text{UCB}_A = \emptyset$, $\text{ECB}_A = \{1, 2\}$, and $\text{UCB}_B = \{1, 2\}$, $\text{ECB}_B = \{1, 2\}$. Now with DJMPO, task τ_A is assigned the higher priority, resulting in a pre-emption cost of 2 when it pre-empts task τ_B. This is enough to render task τ_B unschedulable. However, with the priorities reversed, the pre-emption cost is zero and both tasks will meet their deadlines. Even with the simple ECB-only approach, DJMPO is not optimal as the response time analysis resembles that for tasks with deadlines prior to completion [14].

An alternative approach might be to use Audsley's Optimal Priority Assignment (OPA) algorithm [6, 7]. The OPA algorithm is compatible with any schedulability test that meets three conditions set out in [16]. However, we note that when using the ECB-Union or the UCB-Union approach, the response time of task τ_i depends not only on the set of higher priority tasks, but also on their relative priority ordering, thus breaking Condition 1 in [16]. Both of these approaches are therefore incompatible with the OPA algorithm. By contrast, the ECB-only approach is OPA-compatible.

With pre-emption costs dependent on task UCBs and ECBs there is significant scope for priority assignment to improve schedulability via avoiding specific evictions and reloads: A group of tasks that use and evict the same set of cache blocks would clearly benefit from sharing the same priority level. Research into this joint priority assignment and scheduling problem forms an interesting area for future research. While a comprehensive investigation of priority assignment is beyond the scope of this paper, we can make use of DJMPO as a simple heuristic policy enabling a meaningful comparison between the different analyses. In the case of FP-FIFO scheduling, we adopt a greedy approach based on DJMPO as detailed in Algorithm 1.

Algorithm 1: Greedy DJMPO for FP-FIFO.

```
For each priority level p starting at the highest level down
to however many priority levels are required.  begin
    For each unassigned task in deadline minus jitter
    monotonic order  begin
        If the task can be added to priority level p such
        that it and all the tasks already assigned to
        priority level p are schedulable, then assign the
        task to priority level p. Otherwise break out of
        this inner loop.
    end
    If no tasks remain unassigned then return - a
    schedulable solution has been found.
    If some tasks remain unassigned but no task was
    assigned to this priority level then return - no
    schedulable solution found.
end
```

We observe that greedy DJMPO for FP-FIFO dominates fixed priority pre-emptive scheduling with DJMPO. This is the case because: (i) with greedy DJMPO for FP-FIFO, if we only assign one task to a priority level, then we get the same solution and the same schedulability test as fixed priority pre-emptive scheduling with DJMPO order, (ii) if we are able to assign a task τ_j to a shared priority level p, then such assignment cannot reduce the schedulability of any lower priority tasks. This is the case because the only difference in the response time analysis for a lower priority task τ_i is the pre-emption cost $\gamma_{i,j}$. $\gamma_{i,j}$ is no larger when task τ_j is at shared priority level p, than it is when τ_j is at priority level $(p + 1)$ on its own, since:

$$\text{ECB}_j \cup \bigcup_{m \in \text{hp}(p)} \text{ECB}_m \subseteq \text{ECB}_j \cup \bigcup_{m \in \text{hep}(p)} \text{ECB}_m \tag{21}$$

where the LHS of (21) is the ECB union used in calculating $\gamma_{i,j}$ when task τ_j is at the shared priority level p, and the RHS is the ECB union used when τ_j is alone at the lower priority level $(p + 1)$.

B. Case Study and Evaluation

We repeated the experiments described in the case study (Section VI) and evaluation (Section VII), using the combined ECB-Union and UCB-Union approach and response time analysis for FP-FIFO scheduling, assuming greedy DJMPO priority assignment.

For the taskset in the case study, FP-FIFO scheduling was no more effective than fixed priority scheduling with unique priority levels, hence the breakdown utilization achieved was the same as that given in Table II i.e. 0.767.

In terms of the more general evaluation, we found that as expected, the results for FP-FIFO scheduling dominated those for fixed priority scheduling with unique priority levels; however, the margin of improvement was relatively small; see the set of graphs from Section VII, reproduced below with the addition of a line for FP-FIFO response time analysis using the combined ECB-Union and UCB-Union approach. The largest improvement obtained by using FP-FIFO scheduling, occurs when the range of task periods is small, and hence many tasks have similar deadlines, and can be scheduled using a limited number of priority levels, see Figure 20.

The reason that FP-FIFO shows only a marginal improvement over fixed priority scheduling with unique priority levels is because, although placing more than one task at a given priority level reduces the pre-emption costs, it also tightens the timing constraints that must be met. This happens because the response times of all of the tasks at a given priority level must be less than or equal to the shortest deadline of any of those tasks. This effect negates the advantage of reduced pre-emption costs for many tasksets.

The average breakdown utilization of the tasksets from the baseline configuration was 0.66 with FP-FIFO (combined ECB-Union and UCB-Union approach), compared to 0.64 with fixed priority scheduling (see Table III in Section VII).

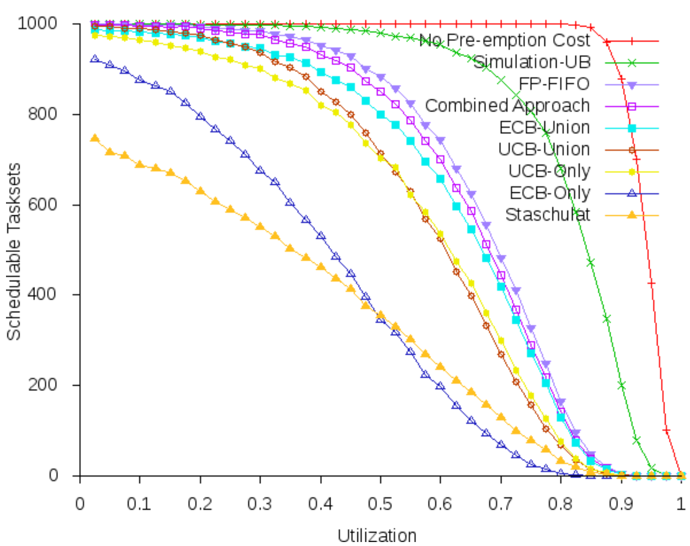


Fig. 14. Evaluation of base configuration. Number of tasksets deemed schedulable at the different total utilizations (with FP-FIFO).

Appendix
B Evaluation Results, constrained deadline

Figures 22, 23, 24, 25, 26, and 27 show results for tasksets with constrained deadlines ($D_i \in [2C_i, T_i]$). Note, we used a minimum deadline of $2C_i$ rather than C_i to ensure that

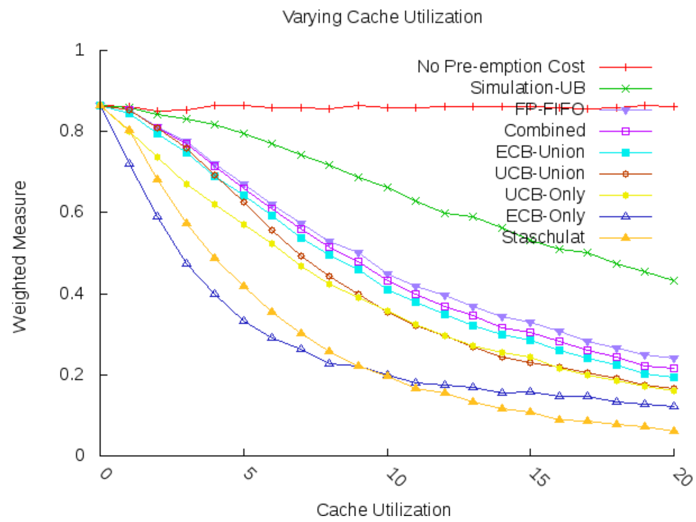


Fig. 15. Weighted schedulability measure; varying cache utilization from 0 to 20, in steps of 2 (with FP-FIFO)

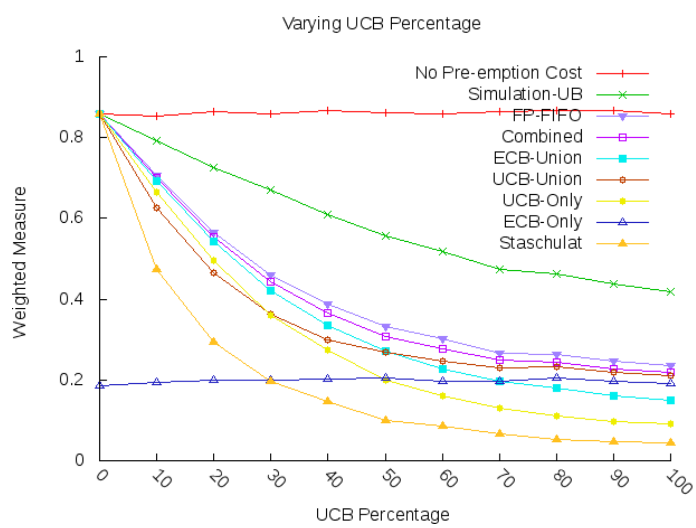


Fig. 16. Weighted schedulability measure; varying reuse factor from 0% to 100%, in steps of 10% (with FP-FIFO)

tasks would not be trivially unschedulable as soon as any pre-emption costs were included. Task deadlines were capped at T_i in the case that $2C_i > T_i$.

The graphs for tasksets with constrained deadlines all show broadly similar behaviour to those for tasksets with implicit deadlines ($D_i = T_i$), save for the fact that at any given level of taskset utilization, the number of schedulable tasksets is reduced.

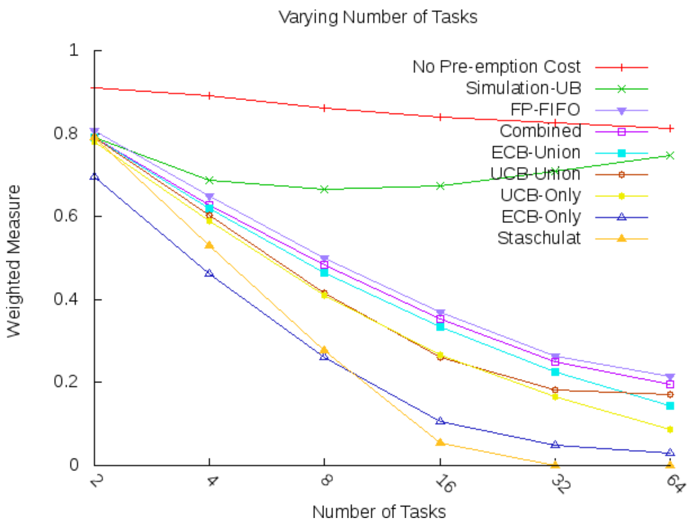


Fig. 17. Weighted schedulability measure; varying number of tasks from $2 = 2^1$ to $2^6 = 64$. (with FP-FIFO)

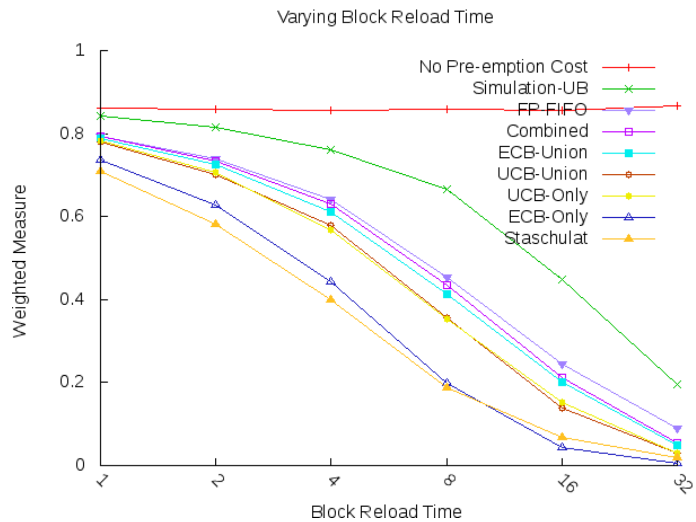


Fig. 19. Weighted schedulability measure; varying block reload time from $2^0 = 1\mu s$ to $2^4 = 32\mu s$ (with FP-FIFO)

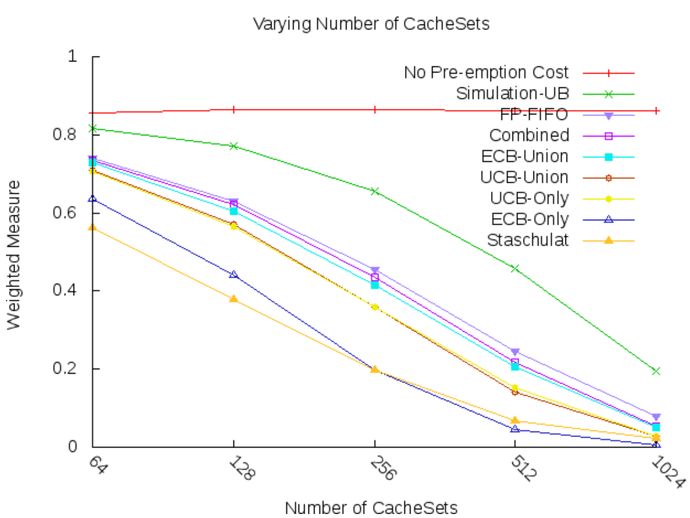


Fig. 18. Weighted schedulability measure; varying number of cache sets from $2^5 = 64$ to $2^{10} = 1024$ (with FP-FIFO)

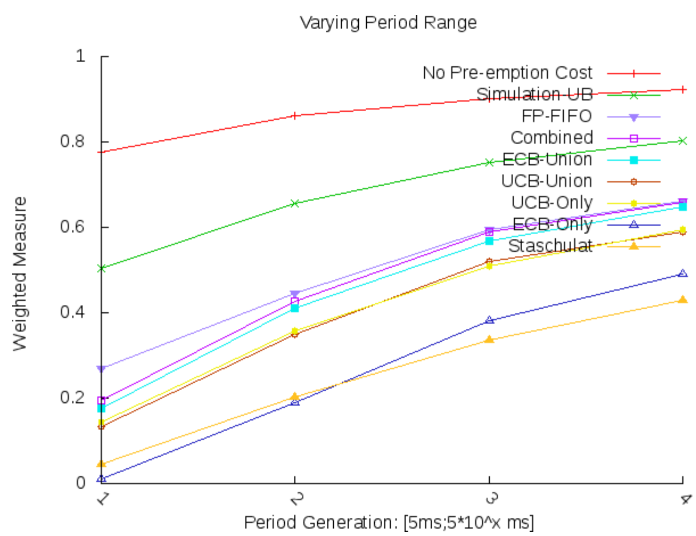


Fig. 20. Weighted schedulability measure; varying the range of task periods $[5, 50]$ to $[5, 5 \cdot 10^4]$ (with FP-FIFO)

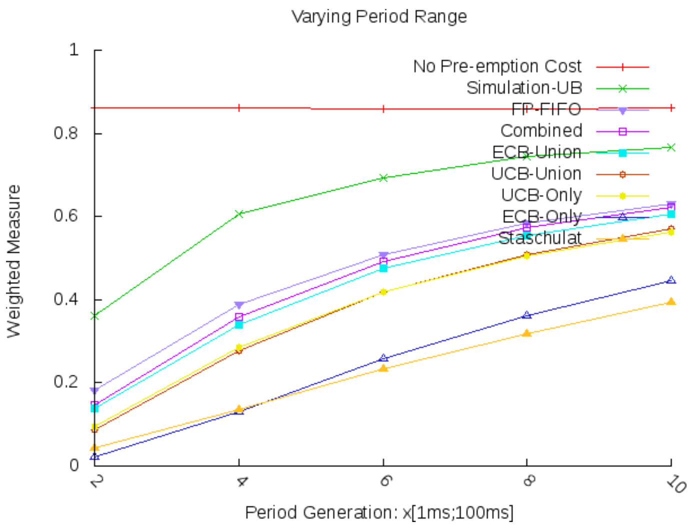


Fig. 21. Weighted schedulability measure; varying the scale of task periods $w[1, 100]$ from $w = 2$ to $w = 10$ (with FP-FIFO)

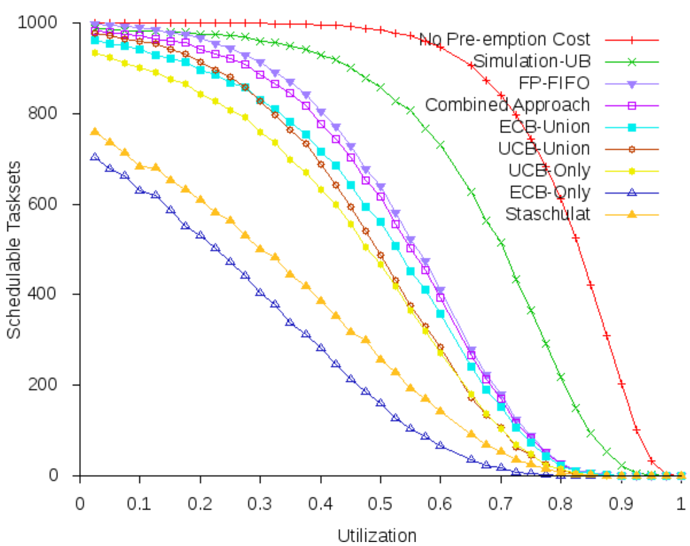


Fig. 22. Evaluation of base configuration. Number of tasksets deemed schedulable at the different total utilizations (constrained deadline).

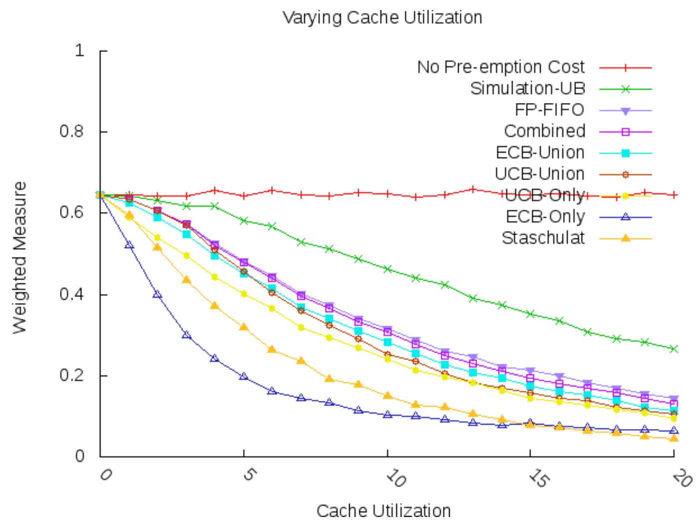


Fig. 23. Weighted schedulability measure; varying cache utilization from 0 to 20, in steps of 2 (constrained deadline)

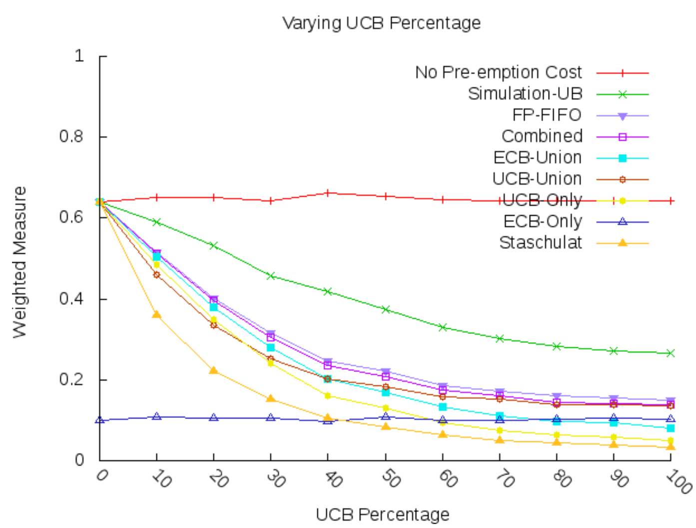


Fig. 24. Weighted schedulability measure; varying reuse factor from 0% to 100%, in steps of 10% (constrained deadline)

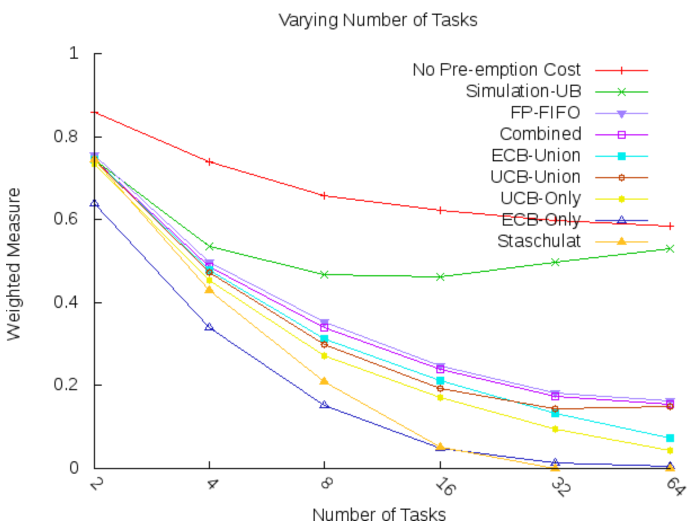


Fig. 25. Weighted schedulability measure; varying number of tasks from $2 = 2^1$ to $2^6 = 64$. (constrained deadline)

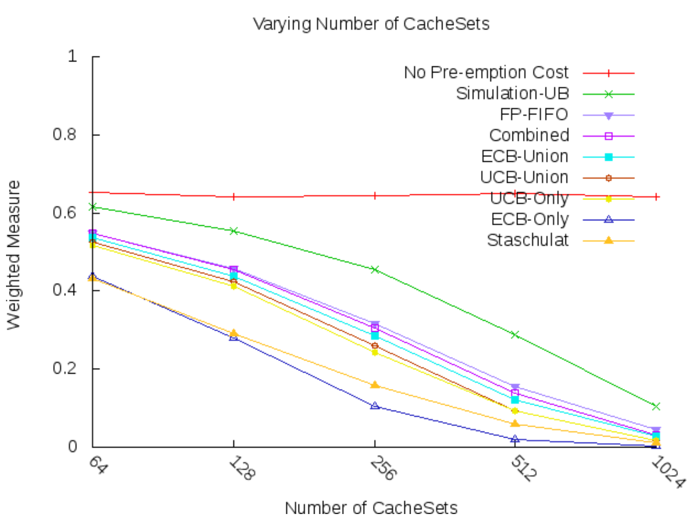


Fig. 26. Weighted schedulability measure; varying number of cache sets from $2^5 = 64$ to $2^{10} = 1024$ (constrained deadline)

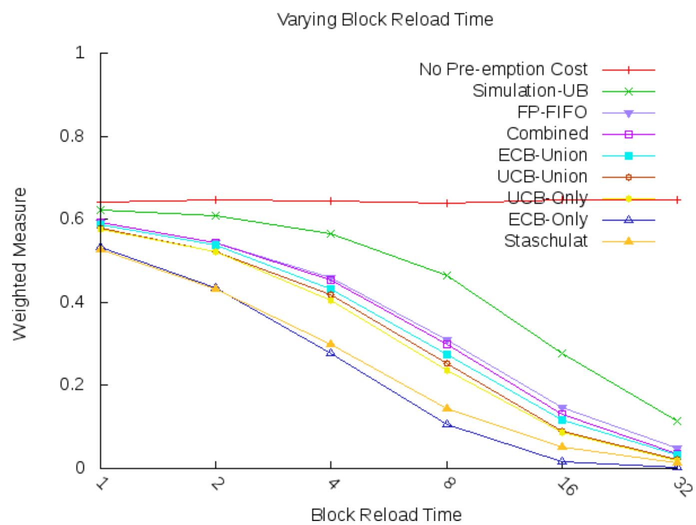


Fig. 27. Weighted schedulability measure; varying block reload time from $2^0 = 1\mu s$ to $2^4 = 32\mu s$ (constrained deadline)

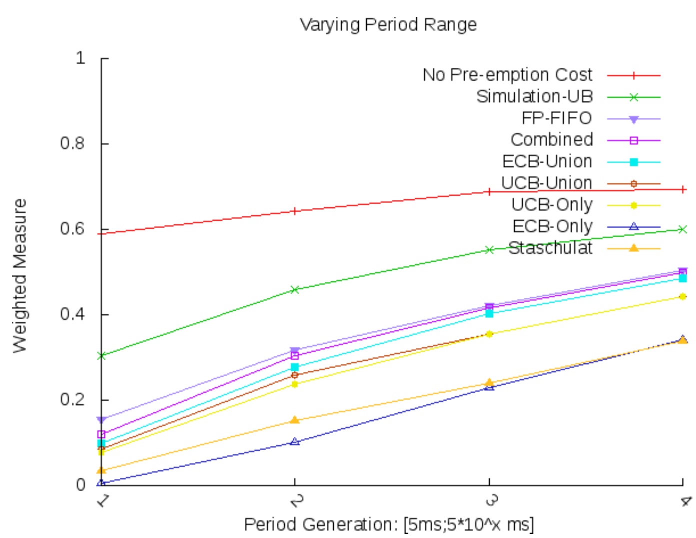


Fig. 28. Weighted schedulability measure; varying the range of task periods $[5, 50]$ to $[5, 5 \cdot 10^4]$ (constrained deadline)

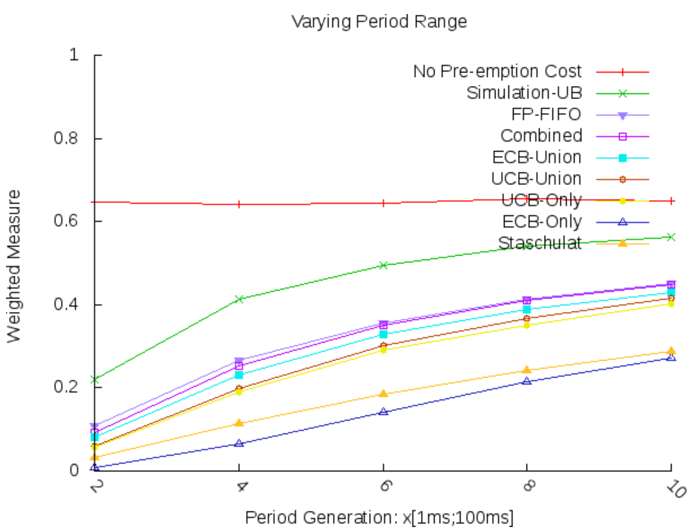


Fig. 29. Weighted schedulability measure; varying the scale of task periods $w[1, 100]$ from $w = 2$ to $w = 10$ (constrained deadline)