Tightening the Bounds on Feasible Preemptions

HARINI RAMAPRASAD and FRANK MUELLER, North Carolina State University

Data Caches are an increasingly important architectural feature in most modern computer systems. They help bridge the gap between processor speeds and memory access times. One inherent difficulty of using data caches in a *real-time system* is the unpredictability of memory accesses, which makes it difficult to calculate worst-case execution times (WCETs) of real-time tasks.

While cache analysis for single real-time tasks has been the focus of much research in the past, bounding the preemption delay in a multi-task preemptive environment is a challenging problem, particularly for data caches.

This paper makes multiple contributions in the context of independent, periodic tasks with deadlines less than or equal to their periods executing on a single processor.

1) For every task, we derive **data cache reference patterns** for all scalar and non-scalar references. These patterns are used to derive an upper bound on the WCET of real-time tasks.

2) We show that, when considering cache preemption effects, the critical instant does not occur upon simultaneous release of all tasks. We provide results for task sets with phase differences to prove our claim.

3) We develop a method to calculate tight upper bounds on the maximum number of possible preemptions for **each job** of a task and, considering the worst-case placement of these preemption points, derive a much tighter bound on its WCET. We provide results using both static and dynamic priority schemes.

Our results show significant improvements in the bounds derived. We achieve up to an order of magnitude improvement over two prior methods and up to half an order of magnitude over a third prior method for the *number of preemptions*, the *WCET* and the *response time* of a task. Consideration of the best-case and worst-case execution times of higher priority jobs enables these improvements.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*scheduling*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Real-Time Systems, Preemptions, Worst-Case Execution Time, Timing Analysis, Data Caches, Cache-Related Preemption Delay

1. INTRODUCTION

Data caches are an invaluable architectural feature in modern computer systems. Being effective in bridging the gap between processor and memory speeds, they provide significant improvement in latency. However, they have one inherent complexity — the latency of memory references becomes unpredictable in the presence of data caches.

This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.
Author's address: Dept. of Computer Science, Center for Embedded Systems Research, North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu, +1.919.515.7889
Preliminary versions of this work appeared in the IEEE Real-Time Embedded Technology and Applications Symposium, 2006 [Ramaprasad and Mueller 2005] and in the IEEE Real-Time Systems Symposium, 2006 [Ramaprasad and Mueller 2006]. This journal version presents new results for phased task sets, performs a complexity analysis on the framework presented, gives analysis overheads for experiments and extends the framework for dynamic scheduling policies. It also proposes a method to calculate an upper bound on the feasible preemptions for any phasing of task sets. Sections 11.2 and 11.4 have been newly added and Section 13 has been significantly extended with the results stated above.

In a real-time system, *a priori* knowledge of the worst-case execution time (WCET) of every task is necessary in order to conduct schedulability tests. The unpredictability introduced by data caches in such a system significantly increases the complexity of timing a task with the aim of obtaining WCET estimates.

Data cache analysis for single tasks, itself a challenging problem, has been the focus of much research ([Lim et al. 1994], [Kim et al. 1996], [Li et al. 1996], [White et al. 1999], [Lundqvist and Stenström 1999]). However, most real-time systems operate in a prioritized, preemptive manner. Every task in the system is assigned a priority based on a particular scheduling policy and, at any time, a task with higher priority may preempt that with a lower priority. This implies that some cache blocks used by the preempted task may potentially be evicted from cache, causing additional reload delay when the lower-priority task resumes execution. Hence, the complexity of data cache analysis increases further in the context of preemptive systems.

In previous work [Ramaprasad and Mueller 2005], we presented a framework that extended the concept of Cache Miss Equations [Ghosh et al. 1997] to derive exact data cache miss/hit patterns for every memory reference in a loop nest. This analysis framework was integrated into a static timing analysis framework to provide tight WCET estimates for tasks in the absence of preemptions.

In this paper, we propose methods to provide tight estimates of the WCET of tasks in a multi-task preemptive environment. The fundamental steps involved in this calculation are as given below.

(1) **Preemption delay:** Given the preempted task, the set of possible preempting tasks and the preemption point, calculate the delay incurred due to the preemption.
(2) **Number of preemptions:** Calculate n_p, an upper bound on the number of times a task can be preempted during execution within a task set.
(3) **Worst-case scenario:** Identify the placement of the n_p preemption points in the iteration space such that the worst-case total delay / preemption cost is obtained.

These issues have been studied by Staschulat et al. for instruction caches [Staschulat and Ernst 2004], [Staschulat et al. 2005]. Our work is orthogonal and focuses on data caches.

In our current work, the calculation of the base execution time for a task (without preemption delay) uses a static cache simulator to account for instruction cache effects. However, in the preemption delay calculation, only data cache effects are taken into account. A similar method may be employed to account for instruction cache related preemption delay, but this has not been implemented yet.

We first present a method to calculate an upper bound on the maximum number of preemptions for a task and construct a pessimistic worst-case preemption scenario based on this number. This yields an upper bound on the WCET estimates. Next, we propose a method to significantly tighten the maximum number of preemptions using the entire range of execution times for a task. Using this new, tighter estimate, we construct a realistic worst-case preemption scenario and derive significantly tighter bounds of the WCET of a task in the context of a task-set.

We also show that the critical instant does *not* occur when all tasks are released simultaneously if we consider preemption delays. Hence, our second method performs a per-job analysis rather than a per-task analysis and considers all jobs within a hyperperiod.

The analysis presented in this paper is currently supported only by a mathematical proof of correctness. Experimental validation of our results using a cycle-accurate simulator is part of future work and has not been presented in this paper.

The remainder of this paper is organized as follows. Section 2 presents the task model used in this paper. Section 3 discusses related work. Section 4 discusses the effect of considering data cache related preemption delay on the critical instant. In Section 5, we briefly discuss our static timing analyzer framework. Section 6 gives an overview of prior work that analyses a single task. In Section 7, we describe our experimental setup. Section 8 introduces our basic methodology in detail. In Section 9, we discuss how to calculate an upper bound on the number of preemptions. Section 10 assesses the general task behavior with respect to preemption delay distributions. Section 11 discusses how WCET estimates are tightened further by eliminating infeasible preemption points. Section 12 provides detailed results of our analysis. Section 14 summarizes the contributions of this work.

2. TASK MODEL

In our work, we consider a periodic real-time task model with deadline less than or equal to the period of a task. A periodic task is a sequence of jobs where the interval (period) between any two consecutive job-releases is the same. The least common multiple of the periods of all tasks in such a system is known as the hyperperiod of the task. Throughout this work, we assume a list-based scheduling model where every job has a fixed priority.

The notation used in the remainder of this paper is described here. A task T_i has characteristics represented by the 5 tuple (Φ_i, P_i, C_i, c_i, D_i). Here, Φ_i is the phase, P_i is the period, C_i is the worst-case execution time, c_i is the best-case execution time and D_i is the relative deadline (less than or equal to the period) of the task. In the context of a specific task set, every task has a set of derived characteristics represented by the 3 tuple (B_i, R_i, $\Delta_{j,i}$). Here, B_i is the blocking time and R_i is the response time of the task. $\Delta_{j,i}$ is the preemption delay inflicted on the task due to a higher priority task T_j. $J_{i,j}$ represents the jth instance (job) of task T_i.

3. RELATED WORK

Several methods have been proposed in the past to bound data cache behavior for a single task without taking into account the effects that other tasks may have on the behavior ([Lim et al. 1994], [Kim et al. 1996], [Li et al. 1996], [White et al. 1999], [Lundqvist and Stenström 1999]). They use methods like data flow analysis, static cache simulation, etc. for this purpose.

Analytical methods for predicting data cache behavior have been proposed. They include the Cache Miss Equations by Ghosh et al. [Ghosh et al. 1999], a probabilistic analysis method proposed by Fraguella et al. [Fraguela et al. 1999] and another analytical method by Chatterjee et al. [Chatterjee et al. 2001]. The common idea behind these methods is to characterize data cache behavior by means of a set of mathematical equations. In prior work [Ramaprasad and Mueller 2005], we have extended the cache miss equations framework to produce exact data cache patterns for references. Techniques that make data caches more predictable and can be applied in preemptive systems are cache partitioning and cache locking [Lisper and Vera 2003], [Puaut and Decotigny 2002]. Both methods lead to a significant loss in average-case performance in order to gain predictability. Recent work shows improvements in these methods for the case of instruction caches [Puaut 2006]. However, since data caches stride over large data sets, it is difficult to prevent loss in performance.

Other techniques have been proposed specifically to calculate preemption delay and analyze schedulability in a multi-task preemptive system. These techniques do not specifically analyze data cache behavior. Instead, they provide a more generic solution applicable to a cache including specific solutions for instruction caches.

Early on, Basumallick et al. conducted a survey of cache related issues in real-time systems [Basumallick and Nilsen 1994]. This survey discussed some initial work related to the calculation of preemption delay. Busquets-Mataix et al. proposed a method to incorporate the effect of instruction caches on response time analysis (RTA) [Busquets-Matraix 1996]. They compared cached RTA with cached Rate Monotonic Analysis (RMA) and concluded that cached RTA outperforms cached RMA. Lee et al. proposed and enhanced a method to calculate an upper bound for cache related preemption delay in a real-time system [Lee et al. 1998; Leung], [Lee et al. 2001]. They used cache states at basic block boundaries and data flow analysis on the control flow graph of a task to analyze cache behavior and calculate preemption delay.

Another approach by Tomiyama et al. calculates cache related preemption delay for the program path that requires the maximum number of cache blocks [Tomiyama and Dutt 2000]. This path is determined by an integer linear programming technique. In this paper, an empty cache is assumed at the beginning of every job and hence, each preemption is analyzed individually. Effects of multiple preemptions are not considered. Negi et al. combined the techniques proposed by Tomiyama et al. [Tomiyama and Dutt 2000] and by Lee et al. [Lee et al. 1998], [Lee et al. 2001] to develop an enhanced framework [Negi et al. 2003]. Once again, however, multiple preemptions are not considered in their work since an empty cache is assumed at the beginning of a task.

The work by Lee et al. was enhanced by Staschulat et al. [Staschulat and Ernst 2004], [Staschulat et al. 2005]. The authors propose a complete framework for the calculation of response time for tasks in a given task set. Response times are determined as shown below.

$$R_i = C_i + B_i + \sum_{j=1..i-1} ((\lceil \frac{R_i}{P_j} \rceil * C_j) + \Delta_{j,i}(R_i))$$

where the blocking time, B_i, is not considered in the example and $\Delta_{j,i}(R_i)$ is the overhead incurred by higher priority tasks preempting the current one.

They address the three issues enumerated in the Section 1, namely calculation of the maximum number of preemption points, identification of their placement and calculation of the delay at each point. However, their focus is not on data caches, but on instruction caches.

In their work, Staschulat et al. discuss the concept of indirect preemptions [Staschulat et al. 2005]. Table I provides a sample task set with phase Φ, period P, WCET C and preemption delay Δ, respectively, for tasks T_1 to T_4. The third column shows Φ', another possible set of phases for the tasks, and is used later, in Section 4, for the purpose of comparison. For simplicity, Δ is assumed to be fixed per task, *i.e.*, incurred when inflicted by any higher priority task.

In Figure 1, execution is depicted by shaded boxes and the preemption delay is depicted by black boxes. Staschulat *et al.* observe that several indirect preemptions affect lower priority tasks only once. For example, in the figure, T_2 is affected by the first two invocations of T_1. T_3 is actually only affected by the first and third invocations since, after being preempted once, it is not scheduled at all until T_2 completes execution. Furthermore,

	Φ	Φ'	P	C	Δ
T1	2	1	3	1	0
T2	1	0.875	15	4.625	0.125
T3	0	0.125	20	2.25	0.75
T4	0	0	25	1	0.125

Table I. Task Set, Optional Phasing

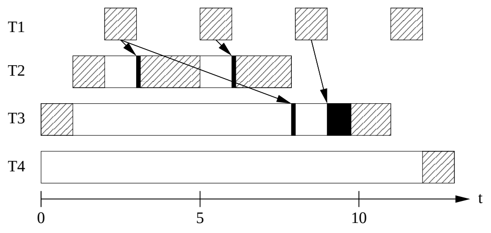


Fig. 1. Preemption with Φ Phasing

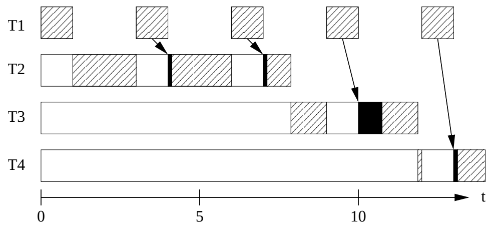


Fig. 2. Preemption without Phasing: $\Phi_i = 0$

while incurring the delay due to preemption, T_3 is preempted again at time eight. Hence, the entire preemption cost is charged again when T_3 resumes at time nine. This results in a response time of R_3 of 11 units. We will show in this work that considering indirect preemption along the lines of Staschulat *et al.* produces pessimistic results.

In more recent work [Staschulat and Ernst 2006], Staschulat et al. propose a timing framework that considers predictable and unpredictable (input-dependent) data cache accesses. For unpredictable accesses, a tight bound of their impact on predictable accesses and a worst-case estimate of the number of additional data cache misses is calculated. As such, their work considers any reused cache content to be replaced when a conflicting range of accesses for unpredictable data references exists, up to the number of cache blocks in either set. Alternatively, they handle cold misses for small arrays that entirely fit into cache and do not suffer replacements at all. Our work makes no assumption on the size of arrays. Furthermore, we assume only predictable data accesses. Notice that for array traversals exceeding cache size, their scheme breaks down as they assume that the entire cache has been replaced. As their and our schemes are complementary, it would be interesting to study the compatibility of these methods. However, this study is beyond the scope of this paper.

In other related work, Ju et al. propose a method to extend CRPD calculations using abstract cache states to dynamic scheduling policies [Ju et al. 2007]. Once again, this work focuses on instruction caches. Our handling of data caches differs significantly.

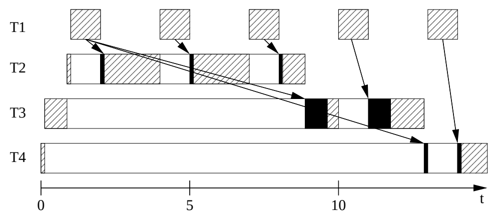


Fig. 3. Preemption with Φ' Phasing

4. PREEMPTION DELAY AFFECTS RESPONSE TIME

Prior work often assumes that the worst-case response time occurs at the theoretical critical instant for fixed-priority scheduling, *i.e.*, upon simultaneous release of all tasks. However, this is not necessarily the critical instant when preemption delays are considered. Consider Figure 2, which uses the same task set but with a phase of zero for all tasks and Figure 3, which has phase differences between tasks. We observe that, in Figure 3, the response time of T_4 (15 units) exceeds its response time in Figure 2 (14 units).

In general, our method is linear (in terms of analysis time) to the length of the hyperperiod. Hence, if the hyperperiod is large, our method would have higher analysis time. However, in practice, the hyperperiod of tasks is often a relatively small number. Hence, releases of tasks can occasionally coincide and are otherwise separated by some minimum time interval (typically 1 ms). For this reason, we consider in our work, *all jobs* of a task within a hyperperiod. We calculate the maximum number of possible preemptions for a job and the data cache related preemption delay at every point. This enables us to consider *ranges of execution* where preemptions can occur within the code. Such analysis yields more accurate results than the calculation of preemption delay per task and helps us significantly tighten the estimates of the number of preemptions and the response times of jobs.

5. STATIC TIMING ANALYSIS

A priori knowledge of the WCET of every task in a task set is assumed for performing schedulability analysis in real-time systems. These estimates need to be *safe* upper bounds on the actual execution times of tasks.

Two methods for calculating the WCET are dynamic timing analysis and static timing analysis. It has been demonstrated in earlier studies [Wegener and Mueller 2001] that dynamic analysis by actual execution of a task does not guarantee worst-case estimates. Further, exhaustive testing of the input space is impractical. In contrast, static timing analysis is a viable approach to derive *safe* WCET bounds. Here, all execution paths in a program are traversed and a conservative upper bound for the execution time of the longest path is calculated.

Several features of the program under consideration (*e.g.*, data dependent control flow, pointer accesses, etc.) affect the calculation of WCET. Similarly, several *architectural features* also cause unpredictability in the timing analysis. One such feature that is particularly hard to model is the *data cache*. Inefficient modeling of the data cache could lead to overly pessimistic WCET estimates, hence affecting the results of schedulability tests.

Figure 4 depicts our framework for static timing analysis to derive WCET bounds. The shaded portions indicate the components responsible for data cache analysis and the actual timing analysis. The framework uses a static cache simulator that simulates the instruction cache and a data cache analyzer framework (developed in prior work [Ramaprasad and Mueller 2005]) to produce data cache reference patterns.

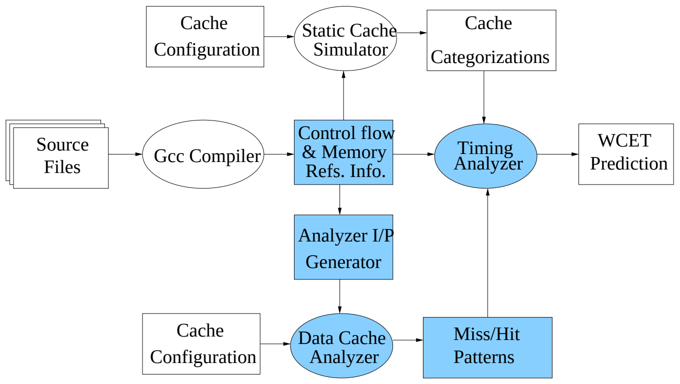


Fig. 4. Static Timing Analysis Framework

6. PRIOR WORK

In prior work [Ramaprasad and Mueller 2005], we propose a data cache analyzer framework that produces data cache reference patterns, in terms of hits and misses, for every scalar and non-scalar memory reference in a scientific, loop nest oriented task.

Our work enhances a framework developed by Vera et al. [Vera et al. 2000], [Vera and Xue 2002] that uses the concept of Cache Miss Equations to statically characterize data cache behavior for a single task. While this original framework only determines the number of data cache misses, our enhanced framework provides information about the position of every miss.

This data cache analyzer framework is integrated into the static timing analyzer framework shown in Figure 4 to calculate the WCET of a task including data cache miss latency. The constraints the tasks need to adhere to in order to be analyzable by our framework are summarized in Section 7.

7. BENCHMARKS AND CACHE CONFIGURATION

In our analysis, we use the static timing analyzer framework described in Section 5. The data cache analyzer in this framework produces data cache reference patterns for every task, the details of which are described in prior work [Ramaprasad and Mueller 2005]. We use the SimpleScalar tool-set [Burger et al. 1996] for compiling our source code. We use this tool-set in conjunction with the portable instruction set architecture (PISA), which is a widely used generic ISA.

Our data cache analyzer poses certain restrictions on the programs it can analyze. Primarily, loop bounds must be known at compile-time, no pointer-based or dynamic memory accesses are allowed and array subscript expressions must be affine functions of the loop induction variables. Furthermore, in our current framework, we only support non-partitioned, direct-mapped data caches.

Benchmarks from the DSPStone benchmark suite [Zivojnovic et al. 1994] are used in our experiments. Pointer accesses in these benchmarks were replaced by equivalent array accesses to make them analyzable by our framework. Abstract inlining [Ramaprasad and Mueller 2005] is used on the function calls to lift all data references into one (main) function. A brief description of the benchmarks used are provided in Table II. We used the benchmarks with different data-set sizes in order to obtain varying timing characteristics. In all our experiments, a 4KB direct-mapped data cache with a hit penalty of 1 cycle and a miss penalty of 100 cycles was used.

Conceptually, our methods could be applied to any processor model. However, in our current implementation, we use the SimpleScalar processor model [Burger et al. 1996].

Benchmark	**Description**
dot-product	Program to find the dot product of two vectors
convolution	Program to implement a convolution filter
fir	Program to implement a finite impulse response filter
lms	Program to implement a least mean-square filter
n-real-updates	Program to perform n real updates of the form D(i) = C(i) + A(i)*B(i), where A(i), B(i), C(i) and D(i) are real numbers, and i = 1,...,N
matrix1	Program to find the product of two matrices

Table II. Description of benchmarks in the DSPStone suite

8. METHODOLOGY

In our prior work, we model data cache behavior for single tasks. However, in reality, most real-time systems consist of multiple tasks operating in a prioritized, preemptive manner. In other words, every task in the system has a priority determined by some scheduling policy. At any point during its execution, a particular task may be interrupted by a task with higher priority. In our work, we consider non-partitioned data caches, which means cache lines may be shared across tasks. Thus, when a task is preempted, a subset of its memory lines may be evicted from the data cache by the execution of the preempting tasks.

Assuming that all memory lines of a task are evicted during preemption leads to overestimation of the Data Cache Related Preemption Delay (D-CRPD), thus affecting the schedulability of task sets. In this paper, our aim is to calculate a safe, but tight estimate of the D-CRPD and hence the WCET and response time of a task. We propose a method to incorporate D-CRPD calculations during WCET calculation itself. Furthermore, to make the calculation as accurate as possible, we use the intersection of the cache blocks that are useful to the preempted task on resuming execution and those that are potentially used by the tasks that execute before the preempted task is restarted.

8.1 Response Time Analysis

We use response time analysis to determine schedulability of a task set [Lehoczky et al. 1989; Audsley et al. 1993]. Task sets are assumed to be periodic and each task is assumed to have a deadline less than or equal to its period. Response time is calculated using an iterative approach as indicated in Equation 1.

$$R_i^n = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^n}{P_j} \rceil \cdot C_j \tag{1}$$

The set $hp(i)$ denotes the set of tasks with a higher priority than task i. For every task, the value of R that converges this equation is its response time. The worst-case execution time of a task i is denoted as C_i and the period, as P_i. Since our method incorporates D-CRPD calculations within the WCET calculation of a task, we do not require an additional term for the delay in Equation 1.

8.2 Phase 1: Calculation of Base Time and Data Cache Patterns

In order to compute the WCET of a task that includes D-CRPD, we first calculate a base execution time for every task. For this purpose, we analyze every task individually using the data cache analyzer to produce data cache reference patterns as described in prior work [Ramaprasad and Mueller 2005]. These patterns are used by the static timing analyzer while building timing trees for every task. The timing tree provides information about the timing of individual nodes (functions/loops) in a given task. It is to be noted that this base time does not include the D-CRPD and is calculated only *once* for every task.

8.3 Phase 2: Preemption Delay Calculation

The data cache analyzer and the timing analyzer interact repeatedly to calculate the WCET of a task with D-CRPD in this phase. The timing analyzer times the program up to a preemption point. At this point, the data cache analyzer is invoked to calculate the number of additional misses due to preemption. This delay is added to the remaining execution time of the task under consideration. The same process is repeated for every interval.

In prior work [Ramaprasad and Mueller 2005], we propose a method to derive *exact* hit/miss patterns for every reference in the loop nest. These patterns are known as data cache reference patterns and indicate the number and position of every data cache miss for every reference. We devised the following method to calculate the actual preemption delay at a given preemption point. All data cache reference patterns for a task are merged, maintaining the order of accesses. References that access the same cache set are connected together to form a chain that effectively indicates cache reuse. Chains for different cache sets are shown using a different line-style. Within each chain, a *miss* is represented by the letter 'M' and a *hit* is represented by a dot. An example with just three cache set chains is shown in Figure 5.

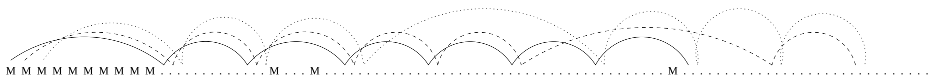

M M M M M M M M M M . . . . M . . . M . . . . M . . . .

Fig. 5. Cache Line Access Chains for Lines 1, 2 and 3

Each point in the access chains of a task is assigned a weight that indicates the number of additional data cache misses that would be incurred if a preemption were to occur at that point. This weight is calculated as follows. First, the number of differently styled chains that *cross over* this point is counted. This effectively eliminates cache sets that are not used after the preemption point. Additionally, we perform two checks.

(1) Chains in which the access point on the chain immediately following the current point is a *miss* in the pattern are not counted. The rationale behind this is that, if the point were a *miss* in the first place, it would be due to some intra-task interference. Hence, a preemption just before that point will not cause any further delay with respect to the particular cache set that the chain represents.
(2) Chains that correspond to a cache set that is not used by any task with a higher priority than the task under consideration are not counted. This ensures that only the cache blocks that could potentially be replaced during preemption of the current task are considered.

Access chains for a task need to be constructed only **once**. However, since the assignment of weights for every point is further dependent on the higher-priority tasks in the task set, the weights are *task set* specific rather than just *task* specific.

9. AN UPPER BOUND ON PREEMPTIONS

In the following, we discuss the methods to calculate the maximum number of preemption points for a task and their placement in the worst case.

9.1 Identification of Preemption Points

First, we need to calculate an upper bound on the number of preemptions for any given task in the worst case. Consider a task T_i. The upper bound on the number of preemptions incurred by T_i is given by Equation 2.

$$n_p^i = \sum_{j \in hp(i)} (\lceil \frac{D_i}{P_j} \rceil) \tag{2}$$

For calculation of preemption delay due to these n_p^i preemptions, we use the access chains of the task as described in Section 8.3. The sum of the n_p^i most expensive delays in the chains is used to calculate an *upper bound* on the worst-case preemption delay for the task and this is added to the WCET of the task. This method of calculating the number of preemptions is denoted as HJ-P in the experimental results section.

9.2 Actual Calculation of WCET and Response Time

Response times of tasks are calculated using the formula shown in Equation 1. In order to calculate the response time of a particular task, we need to know the response times of all higher priority tasks. Hence, we start with the highest priority task and proceed towards the lowest priority task. For every task, the D-CRPD, calculated using the method described above, and the WCET with D-CRPD is used in the equation.

10. PREEMPTION DELAY COSTS

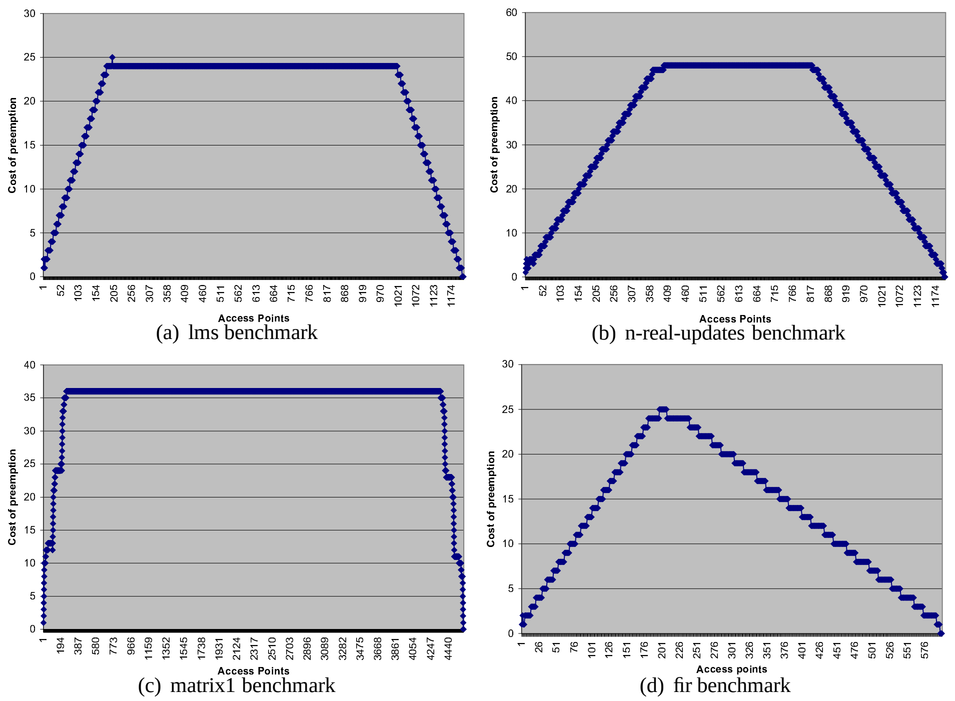
As shown in Section 9, an upper bound on the worst-case preemption delay for a task may be calculated using the n_p most expensive preemption delays. In this process, we do not impose any constraints on the interval between consecutive preemption points.

The reason for using this upper bound is based on an observation regarding the reuse of cache lines in a task. The distribution of preemption costs for the second, third, fourth and fifth tasks of a sample task set (Table III) are shown in Figure 6. The X-axis shows the memory access points in order and the Y-axis shows the cost of preemption at a point

Benchmark	Period (=deadline)	Stand alone WCET
convolution	62500	7491
fir	125000	9537
lms	125000	14536
n-real-updates	250000	16738
matrix1	250000	54168

Table III. Example Task Set Characteristics - Task Set 1

in terms of the calculated weight at the point. The calculation of the weight at a given iteration point is described in Section 8.3.



(a) lms benchmark (b) n-real-updates benchmark (c) matrix1 benchmark (d) fir benchmark

Fig. 6. Distribution of preemption costs across the iteration space

From these graphs for the benchmarks in Figures 6(a), 6(b) and 6(c), we observe that a large number of access points have the highest preemption cost. Furthermore, they are all consecutive. This indicates that a preemption at any of these points would result in the same preemption delay and, hence, picking the n_p most expensive points gives a reasonably tight bound on the worst-case preemption delay.

The distribution of these delays is a direct indication of the reuse of cache lines in the respective tasks. In most programs, ninety percent of the time is spent in ten percent of the code. Within this ten percent, there are repetitive reuse patterns, which implies temporal and spatial reuse. Hence, during the course of this section, all data that is used in the section is already in the data cache. Preemption at any point in this section would result in more or less the same cache lines being evicted, hence causing the same preemption delay.

Although the above behavior is observed in most cases, there are some benchmarks (Figure 6(d)) in which we observe a gradual increase in the preemption cost up to some point and then a decrease in the cost for successive access points. Hence, in the next section, we discuss methods to tighten the worst-case preemption delay bound.

11. TIGHTENING PREEMPTION DELAY BOUNDS

In Section 9, we describe a method to calculate the WCET of a task with D-CRPD. However, simplified methods were used to calculate the maximum number of preemptions and the total preemption delay incurred. We now propose methods to calculate significantly tighter estimates of the number of preemptions and a more realistic, yet safe method to identify the placement of these preemption points that leads to the worst-case D-CRPD.

11.1 Eliminating Infeasible Preemption Points

The formula used to calculate the maximum possible number of preemptions for a task in Section 9 is based on the number of jobs of higher priority tasks that are released in the period of the lower priority task and the amounts of time they each take to execute. This leads to the consideration of several infeasible preemption points either because the lower priority job has not been scheduled at all and, hence, cannot be preempted, or because it has already finished executing. Hence, we developed a method that considers the best and worst case execution times of higher priority tasks to eliminate these infeasible points. Since we showed in Section 4 that the critical instant does not necessarily occur when all tasks are released at the same time, we calculate the WCET for each job of a task within a hyperperiod with the individual phasing.

Our method to eliminate infeasible preemption points is described in the remainder of this section. However, we do not explicitly add the preemption delay at every stage in this explanation for the sake of simplicity. The actual calculation of preemption delay and the identification of the placement of preemption points in the iteration space of the preempted task are discussed in the next section.

We use an example to explain the basic methodology involved in the elimination of infeasible preemption points. The characteristics of the task set used in the example is shown in Table IV. The *hyperperiod* of this task set is *200* and all jobs within this hyperperiod are considered in our analysis.

Task	**Period = deadline**	**WCET**	**BCET**
T_0	20	7	5
T_1	50	12	10
T_2	200	30	25

Table IV. Example Task Set Characteristics - Task Set 2

For the purposes of our analysis, we require the construction of a timeline for every task indicating release points for higher priority jobs. All these release points are potential preemption points for the task under consideration and the goal is to eliminate the infeasible ones. Figures 7 and 8 show the time lines for tasks T_1 and T_2, respectively. The arrows represent job releases and are numbered consecutively. The preemption points that get eliminated as a result of our analysis are circled. BCETs of higher priority jobs are laid out above the time axis and WCETs of higher priority jobs, below the time axis. In this example, black rectangles are used for jobs of task T_0, gray rectangles for those of task T_1 and the red rectangles, for those of T_2.

Let us first consider the timeline for task T_1 (Figure 7). In order to determine whether release point 1 is a feasible preemption point for $J_{1,0}$, we need to perform two checks. First, we need to calculate whether $J_{1,0}$ can get scheduled in the previous interval, *i.e.*, between points 0 and 1. Secondly, we need to check whether any execution of $J_{1,0}$ remains beyond point 1. For the first condition, we use the BCETs of all higher priority jobs (in this examples, $J_{0,0}$). Since there is idle time after placing the BCET of $J_{0,0}$ (5 units), we determine that $J_{1,0}$ could be scheduled before point 1. To check if any execution of $J_{1,0}$ remains beyond point 1, we use the sum of the WCETs of $J_{0,0}$ and $J_{1,0}$, namely 7 and 12 units respectively. Since this does not exceed point 1, $J_{1,0}$ is guaranteed to finish within the current interval. Hence, we conclude that no preemptions are possible for $J_{1,0}$.

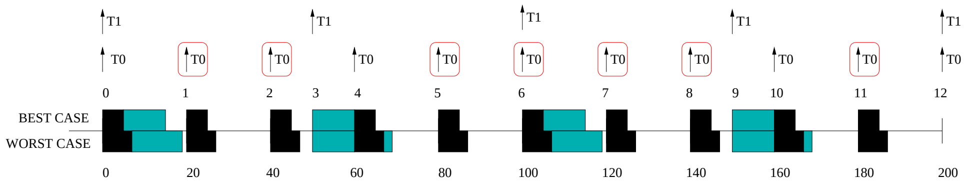


Fig. 7. Timeline for Task T_1

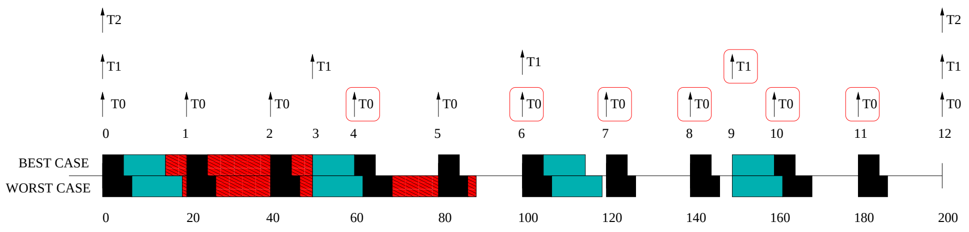


Fig. 8. Timeline for Task T_2

Second, proceed to the next release of T_1, *i.e.*, $J_{1,1}$. During the interval between release points 3 and 4, in the best case, we note that no higher priority job needs to be scheduled. Hence, $J_{1,1}$ can be scheduled in this interval. Next, we calculate that, in the worst case, $J_{1,1}$ is not guaranteed to finish before point 4. This leads to the conclusion that point 4 is a potential preemption point for $J_{1,1}$. Proceeding this way, we calculate the maximum number of preemption points for $J_{1,1}$ to be 1. The analysis is repeated for further releases of T_1 within the hyperperiod.

In the above example, we see that the second release of T_1, namely $J_{1,1}$, has a higher number of preemptions than $J_{1,0}$, hence creating the possibility of a worse response time for $J_{1,1}$ by the addition of the preemption delay. This proves our claim that the critical instant does not necessarily occur when all tasks are release simultaneously.

The maximum number of preemptions for releases of task T_2 are calculated using the same analysis. The timeline for T_2 is shown in Figure 8. For this task, we need to consider two higher priority tasks, namely T_0 and T_1. Starting with $J_{2,0}$, we determine that release point 1 is a feasible preemption point since $J_{2,0}$ can be scheduled before point 1 and is not guaranteed to finish before point 1. Similarly, we determine that points 2 and 3 are feasible. However, when we consider the interval between points 3 and 4, we calculate that, even in the best case, the execution of $J_{1,1}$, which has higher priority, occupies the entire interval. Hence, there is no possibility of $J_{2,0}$ being scheduled in this interval. This leads to the elimination of point 4 as a feasible preemption point for $J_{2,0}$. In the worst-case, $J_{2,0}$ finishes execution at time 82. Hence, points 6, 7, 8, 9, 10 and 11 are also eliminated. At the end of the hyperperiod, our analysis determines that the maximum number of preemptions for $J_{2,0}$ is 4. Our original method (described in Section 9) produces a bound of 9 for the same job.

In summary, the method is as follows. Consider a set of tasks T_0, ..., T_n. Let $J_{i,0}$, ..., $J_{i,k}$ represent the jobs of task T_i. Assume that task T_0 has the highest priority and that task T_n has the lowest priority using a static priority scheme.

A timeline between 0 and the hyperperiod of the task set is constructed for every task T_i and the releases of all higher priority jobs are marked on this timeline. The feasibility of a release point (say x) as a preemption point for $J_{i,j}$ is determined by performing two checks for the interval between release points x and x-1. First, we check whether $J_{i,j}$ has a chance of being scheduled in this interval based on the BCETs of higher priority jobs. If not, point x is not a feasible preemption point. If yes, we proceed to the next check of determining if any portion of execution of $J_{i,j}$ remains beyond point x. If yes, point x is a feasible preemption point. For this, we use the WCETs of all jobs executing in this interval, in order of priority, including $J_{i,j}$.

The above calculations are repeated for every interval between potential preemption points for $J_{i,j}$ until it is guaranteed to finish. *This analysis is performed for every job in the hyperperiod of the task set*.

11.2 Extension to a Dynamic Scheduling Policy

The analysis presented in Section 11 assumes a static priority scheme. The analysis may be extended to support dynamic priority schemes as follows. Instead of calculating priorities in the beginning of the analysis and assuming that they never change, we recalculate job priorities at the beginning of every interval between consecutive preemption points. By doing this, we add the flexibility of conceptually being able to use different scheduling policies. Our current implementation supports the static Rate Monotone (RM) policy and the dynamic Earliest Deadline First (EDF) policy.

11.3 Correctness of Analysis

Consider a task set with *n* tasks, T_0, ..., T_{n-1}. Let us assume that the tasks are in decreasing order of priority. Let C_0, ..., C_{n-1} be the WCETs of the tasks and c_0, ..., c_{n-1} be their BCETs. The WCET and BCET are safe upper and lower bounds, respectively, on the longest and shortest possible execution time of a task.

Preemption of a task can only occur when it is currently *running*. Furthermore, in the case of scheduling policies such as Rate-Monotone, Deadline-Monotone, Earliest-Deadline-First, *etc.*, the positions of potential preemption points for a task are fixed since they are the *release* points of a task with higher priority.

Let us consider the interval between two *consecutive* preemption points, p_{-1} and p. Let us assume that jobs J_{0,k_0}, ..., J_{i,k_i} have been released at some prior point and have not yet completed execution at p_{-1}. Let us assume that J_{i,k_i} is the job for which we need to calculate the maximum number of preemptions possible.

Let *x* be the length of the interval between preemption points p_{-1} and p. We have three cases to consider.

Case 1: $\sum_{j=0}^{i-1} c_{j,k_j} < x$, $\sum_{j=0}^{i} C_{j,k_j} > x$. Assume J_{i,k_i} cannot be preempted at p, *i.e.*, it cannot be running at time p. However, $\underset{j=0..i-1}{\exists} \quad e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$, *i.e.*, J_{i,k_i} is running at p. Contradiction. Hence, p is a feasible preemption point.

Case 2: $\sum_{j=0}^{i-1} c_{j,k_j} < x$, $\sum_{j=0}^{i} C_{j,k_j} < x$. Assume J_{i,k_i} can be preempted at p, *i.e.*, it may be running at time p. Hence, $\underset{j=0..i-1}{\exists} \quad e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$. However, $\sum_{j=0}^{i} C_{j,k_j} < x$ implies $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} < p$. Contradiction. Hence, J_{i,k_i} cannot be running at p, and p is not a feasible preemption point.

Case 3: $\sum_{j=0}^{i-1} c_{j,k_j} > x$. Assume J_{i,k_i} can be preempted at p, *i.e.*, it may be running at time p. Hence, $\underset{j=0..i-1}{\exists} \quad e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$. However, $\sum_{j=0}^{i-1} c_{j,k_j} > x$ implies $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} > p$. Contradiction. Hence, J_{i,k_i} cannot be running at p, and p is not a feasible preemption point.

Hence, preemptions can only occur under Case 1, which is the condition checked by our algorithm (see Figure 12) with the summations of WCET and BCET in the for loop and the check implemented in the subsequent conditions.

The algorithm described above is shown as pseudocode in Figure 12 in the appendix. The variables used are first described. The rest of the algorithm describes a loop that iterates over every interval between consecutive preemption points (comment 2). For each interval, after performing some initializations (comment 3), tasks released at the beginning of the interval are identified (comment 4) and the variables related to the release of a new job are initialized (comment 5). Next, the current priorities are calculated using the scheduling policy for the task set (comment 6). Tasks active in the current interval are traversed in order of priority (comment 7). For each task, best case and worst case scenarios are calculated (comments 8 and 9 respectively). During this process, the minimum and maximum times available for execution of a task in the current interval are calculated. Once all active tasks have been processed, preemption information for every task that could be preempted at the end of the interval is updated (comment 10). For all jobs that are guaranteed to be done before the end of the current interval variables used during the analysis are reset (comment 11).

11.4 Complexity of Analysis

Static data cache analysis to produce data cache reference patterns is performed only once for each task. Here, we iterate through the iteration space of a task, hence making the time and space complexity proportional to the number of data references, n_d in the task, namely $O(n_d)$.

The complexity of our algorithm to calculate D-CRPD and, hence, WCET of a task is $O(n_J * n_T * n_d)$ where n_J is the number of job releases in the hyperperiod of the task set and n_T is the number of tasks. Our algorithm iterates over every interval between job release points. This introduces a complexity of n_J. Within each interval, the algorithm iterates over currently active jobs in order of priority. Since we consider systems where a task has a deadline equal to or less than its period, there can be at most one job of a particular task active at any time. Hence, iterating over active jobs adds a complexity of n_T. Finally, at every identified preemption point, maximum possible delay incurred by the preempted task is calculated using its access chains and information about the range of iteration points at which the preempted task is determined to be when it is preempted. This introduces a complexity of n_d since n_d is the length of the access chain of a task. However, in reality, this range is usually much smaller than n_d for a given preemption point since it is limited by the largest interval between two consecutive potential preemption points for a task.

11.5 Calculation of the Preemption Delay

In Section 11.1, we describe our algorithm to eliminate infeasible preemption points in isolation, without details of the actual calculation of the preemption delay incurred at every feasible point. Here, we discuss the calculation of preemption delay at every point and its addition to the remaining WCET of the preempted task.

Every preemption point determined to be feasible for a task is a point in time. This point in time needs to be translated into an execution point in the iteration space of the preempted task. The preemption delay at this point may then be calculated using the access chains of the preempted task.

To explain the above with an example, consider the task set whose characteristics are shown in Table IV. On the timeline for task T_2 shown in Figure 8, point 1 is identified to be a feasible preemption point. In order to obtain the delay due to preemption at that point, we need to identify the iteration point (the loop iteration number of a particular loop within the task [Ramaprasad and Mueller 2005]) within $J_{2,0}$ that has been reached at the time that the preemption occurs.

The static timing analyzer framework described in Section 5 provides best-case and worst-case execution time estimates for a program. Furthermore, given a certain interval of time, it can provide information about the iteration point that the program could be at the end of the interval both in the best and the worst-case scenarios.

Using our feasible preemption point analysis, we are able to obtain the minimum and maximum time available for a task within every interval between preemption points. This information is obtained using the best and worst-case execution times of higher priority tasks as described in Section 11.1. In the current example, we use the BCETs and WCETs of jobs $J_{0,0}$ and $J_{1,0}$ since they have higher priority that job $J_{2,0}$. In the first interval, after subtracting the time required for the higher priority jobs, we are left with 5 units of time in the best case and 1 unit of time in the worst case for $J_{2,0}$. These provide an upper and lower bound, respectively, for the time available for $J_{2,0}$ in the interval.

The upper and lower bound thus identified are each supplied as inputs to the static timing analyzer framework. The framework performs best and worst-case timing analysis of the preempted task to produce, for each input, two iteration points. One iteration point represents the latest possible point that could be reached (*i.e.*, cannot be exceeded) by the preempted task in the given time and is obtained from best-case timing analysis of the task. The second iteration point represents the earliest iteration point that is guaranteed to be reached in the given time and is obtained from worst-case timing analysis of the preempted task.

Considering the earliest and latest iteration points among the four iteration points obtained provides us with the range of iteration points that the preempted task could have reached when it is preempted. Now, we calculate the preemption delay at each point in this range and choose the maximum delay among those and consider that to be the worst-case preemption delay due to preemption at the particular preemption point. This delay is added to the remaining WCET of the preempted task.

Let us revisit the example of task T_2 that is preempted at point 1 as shown in Figure 8. Assume T_2 is a program that has a loop with 100 iterations. In the interval between points 0 and 1, T_2 is guaranteed to execute for at least 1 unit of time and is guaranteed not to exceed 5 units of time as calculated using our algorithm described in Section 11.

Using the lower bound of the time available for its execution (1 unit), assume the static timing analyzer determines that $J_{2,0}$ is guaranteed to reach at least iteration 4 and cannot proceed beyond iteration 7. Similarly, assume it determines that $J_{2,0}$ is guaranteed to reach at least iteration 9 in 5 units of time (upper bound of the time available for its execution) and does not proceed beyond iteration 13 in the same amount of time. We now consider the entire range of iteration points between 4 and 13 and calculate the delays at every point. Among these, we identify the highest delay and add this delay to the remaining WCET of $J_{2,0}$.

The algorithm used to implement the above method of calculating worst-case preemption delay at a given preemption point is summarized below. The static timing analyzer framework is invoked to perform worst-case partial timing on the minimum available execution time for the preempted task. This yields the beginning of the range of iteration points to be considered. Next, the timing analyzer is invoked to perform best-case partial timing on the maximum available execution time for the preempted task. This yields the end of the range of iteration points. The range thus identified is provided to a function that iterates through the access chains of the preempted task and calculates the highest delay in the given range.

In the method described thus far, we assume that, for every task, we know the values of its period, deadline and *phase*. For a given phasing of tasks, our method calculates the worst-case response times for all tasks. Instead, we now propose a modification to our method to calculate the worst-case response times for tasks regardless of the phasing of the tasks.

In the algorithm described above, for every preemption point, we calculate a *range* of iteration points where the preempted task could be when it is preempted. We then consider the maximum preemption delay in this range as the preemption delay at the preemption point.

Instead, we now assume that the maximum delay in the *entire* iteration space is incurred at *every* preemption point. Further, in order to allow any phasing among tasks, we add extra preemptions for every job at its release. Let us assume that the maximum possible phasing for any task is x units. Furthermore, for any given task, the maximum phasing is less than or equal to its own period. Under these conditions, Equation 3 gives the number of extra preemptions to add in case of a static-priority policy, and Equation 4 gives the number of extra preemptions in the case of a dynamic-priority scheduling policy.

$$pextra_x^{i,j} = \sum_{hp=1}^{i-1} (\lceil \frac{min(x, P_i)}{P_{hp}} \rceil) \tag{3}$$

$$pextra_x^{i,j} = \sum_{hp=1}^{n} (\left\{ \begin{array}{ll} \lceil \frac{min(x,P_i)}{P_{hp}} \rceil) & , if hp \neq i \\ 0 & , otherwise \end{array} \right\}) \tag{4}$$

However, we also observe that, every time a lower-priority task gets preempted, at least one higher-priority task executes. The minimum amount of execution time of this higher-priority task may be safely assumed to be at least equal to the shortest best-case execution time (BCET) among all higher-priority tasks. Hence, the shortest BCET is subtracted from the maximum possible phase of the lower-priority task before adding any more extra preemptions, thereby tightening the bound on the number of extra preemptions. This calculation effectively gives us an upper bound on the number of preemptions and on the worst case response times for a maximum phasing of x.

Consider the example used earlier in Section 3. The task-set characteristics for this example are shown in Table I. Figure 2 shows response times of tasks when all tasks are released simultaneously. In order to calculate the response time of a task irrespective of phasing, we use this scenario. To every task, we add one preemption more than the number calculated. For example, for task T_2, we calculate the response time assuming three preemptions instead of two and considering the maximum preemption delay at each of these points. In our current example, since we assume that a task has a constant preemption delay, Δ, for any preemption, we use that value as the maximum preemption delay and obtain a response time of 8 units instead of 7.875 units. In Section 12, we provide worst-case response times calculated in this manner in addition to the worst-case response time for a given phasing.

When a preemption takes place, it results in a context-switch at the operating system level. The operating system code that is executed in order to perform the context-switch may also use the data cache and, hence, alter the results of our analysis. In our current experiments, we have not considered this factor. However, conceptually, this issue may be dealt with in the following manner.

First, we need to identify the data cache lines that are used by the operating system code. Since this code may not adhere to the constraints that our analysis framework poses on the programs that we analyze, our analysis cannot be used. Hence, we allocate a predetermined area in the memory to hold the operating system code and thereby constrain the data cache lines that it may use.

Second, we need to consider the effects that the execution of operating system code would have on the tasks being analyzed. For this purpose, while calculating the preemption delay incurred by a task at a given preemption point using access chains, we consider the cache lines allocated for the operating system code as potential candidates for eviction. In other words, the operating system code would be treated as the highest priority task in the system and would execute every time there is a preemption.

12. EXPERIMENTAL FRAMEWORK

For our experiments, several task sets were constructed using the DSPStone benchmarks with different data set sizes. The benchmarks used, along with their stand-alone WCETs and BCETs, are shown in Table V. A benchmark ID is given to each of the benchmarks. This ID will be referenced in the result tables.

In our experiments, we used tasks sets that have a base utilization (utilization without considering preemption delays) of 0.5, 0.6, 0.7 and 0.8. Task sets of different sizes (2, 4,

ID	Name	WCET	BCET	ID	Name	WCET	BCET
1	convolution	7491	7491	15	matrix1	59896	54015
2	200convolution	14191	14191	16	fir	9537	9537
3	300convolution	20891	20891	17	500fir	43937	43937
4	500convolution	34291	34291	18	600fir	54837	52537
5	600convolution	45291	40991	19	700fir	65937	61137
6	700convolution	55491	47691	20	800fir	77037	69737
7	800convolution	66191	54391	21	900fir	88137	78337
8	900convolution	76391	61091	22	1000fir	99237	86937
9	1000convolution	87091	67791	23	lms	14536	14536
10	n-real-updates	16738	16738	24	600lms	89636	79536
11	300n-real-updates	56538	47338	25	700lms	112636	92536
12	400n-real-updates	92238	62638	26	800lms	135636	105536
13	500n-real-updates	127538	77938	27	900lms	158636	118536
14	dot-product	750	750	28	1000lms	181636	131536

Table V. Stand-Alone WCETs and BCETs of DSPStone Benchmarks

6, 8) were constructed for each of these utilizations. For 0.8 utilization, we were able to construct a task set consisting of 10 tasks as well. In all these task sets, we assume a static priority scheme.

The maximum number of preemptions, n_p, possible for a task are calculated using three different methods to provide a comparison to our methods described in this paper.

(1) An upper bound on the number of preemptions, n_p, for a task is determined by using Equation 2. This is denoted as **HJ-P** in our results.

(2) We calculate n_p by considering indirect preemption effects as proposed by **Staschulat et al.** This method uses the periods and response times of tasks [Staschulat et al. 2005]. This is denoted as **Stas-R**.

(3) We calculate n_p using the range of execution times of higher priority jobs as proposed in Section 11. This new method uses the periods, WCETs and BCETs of tasks to calculate feasible preemption points. We use two methods for the actual calculation of preemption delay as described in Section 11.5. The method using the maximum delay in a range of iteration points (where the task is determined to be) is denoted as **OurFP-RangeMax** and the one using the maximum delay in the entire iteration space is denoted as **OurFP-ItSpMax**.

.

In the current set of experiments, the maximum phasing for a task, used in Equations 4 and 3, is assumed to be 1000 cycles. Although all the methods calculate the maximum number of preemptions for a task, the first two methods do not provide information about the placement of the preemption points. Hence, in these cases, we consider the n_p largest delays for a task in order to obtain its D-CRPD.

13. EXPERIMENTAL RESULTS

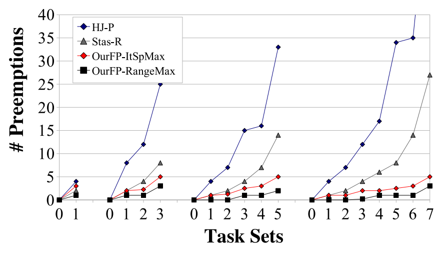
We performed several experiments to demonstrate the working of our new methods (OurFP-RangeMax and OurFP-ItSpMax) in comparison to prior methods (HJ-P and Stas-R). The results of these experiments are shown and discussed in the following sections.

13.1 Response Time Analysis

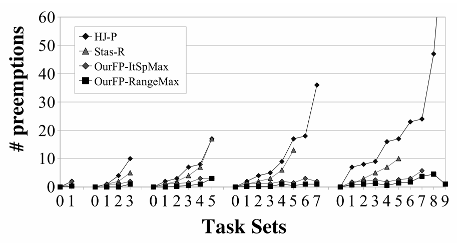
In the first set of experiments, we use the task sets described in Table VI and perform response time analysis using all the methods described in Section 12 for calculating the number of preemptions. Results obtained for the task sets with base utilization of 0.5 and 0.8 are shown in Figure 9. Results for utilizations of 0.6 and 0.7 are similar and are omitted due to space constraints. Different graphs are used to represent the three metrics studied — maximum number of preemptions, WCET with preemption delay and response times for tasks. In each graph, the x-axis represents the several task sets used. Tasks within each task set are numbered in decreasing order of priority. It is to be noted that, in this set of experiments, we use a static priority scheme for the task sets.

# Tasks	**2**	**4**	**6**	**8**	**10**
			U = 0.5		
IDs	16, 19	1, 15, 18, 22	23, 3, 6, 11, 19, 26	2, 3, 4, 11, 15, 18, 7, 27	
Periods	50K, 200K	50K, 400K, 500K, 100K	400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
			U = 0.6		
IDs	21, 27	1, 15, 8, 27	3, 4, 6, 11, 19, 26	2, 5, 6, 11, 15, 18, 7, 27	
Periods	300K, 500K	50K, 400K, 500K, 1000K	100K, 400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
			U = 0.7		
IDs	27, 21	16, 9, 7, 27	3, 17, 8, 7, 20, 27	3, 5, 20, 11, 15, 19, 8, 26	
Periods	300K, 500K	50K, 400K, 500K, 1000K	100K, 400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
			U = 0.8		
IDs	27, 26	28, 13, 27, 19	21, 8, 20, 13, 25, 19	8, 26, 20, 15, 9, 11, 8, 21	10, 8, 15, 9, 5, 11, 20, 27, 22, 17
Periods	300K, 500K	500K, 500K, 1000K, 2000K	400K, 500K, 500K, 1000K, 1000K, 2000K	400K, 500K, 800K, 800K, 1000K, 2000K, 2000K, 4000K	100K, 625K, 625K, 625K, 1000K, 1000K, 1250K, 1250K, 2500K, 5000K

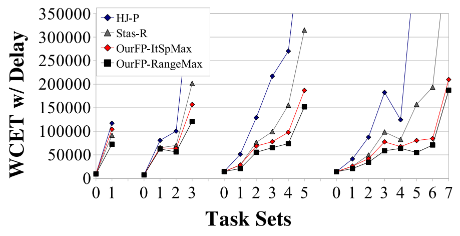
Table VI. Task Set Characteristics: Benchmark IDs per Task Set and Periods [cycles]



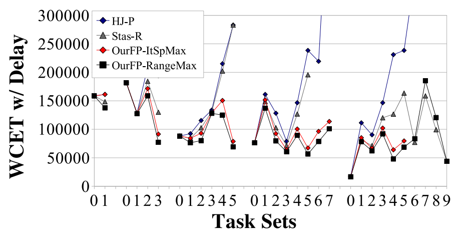
(a) # Preemptions for U = 0.5



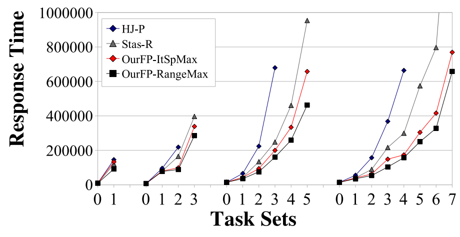
(b) # Preemptions for U = 0.8



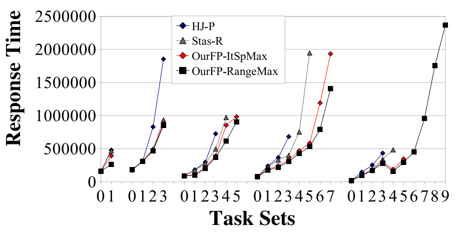
(c) WCET w/ delay for U = 0.5



(d) WCET w/ delay for U = 0.8



(e) Response Time for U = 0.5



(f) Response Time for U = 0.8

Fig. 9. Results for U=0.5 and U=0.8 using RM policy

Our method of using only *feasible* preemption points consistently derives a much tighter bound on the number of preemptions for a given task as compared to the two prior methods (HJ-P and Stas-R). Since the number of preemption points identified is smaller, bounds for the WCET with preemption delay and the response time for each task are also significantly tighter, as indicated in our results. Even the method that calculates an upper bound on the maximum number of feasible preemption points for any phasing of tasks (OurFP-ItSpMax) provides significantly tighter bounds than the two prior methods.

We observe from the graphs that, for some of the tasks, response times are not indicated for the first two methods of comparison (HJ-P and Stas-R). This means that the response exceeded the deadline of the task, making the task set unschedulable. Results from our methods (OurFP-RangeMax and OurFP-ItSpMax) show that these task sets are, in reality, schedulable. This underlines the potential benefits of our new methods. For the calculation of response time, we use a fixed-point approach and proceed only as long as the deadline of the given task is not exceeded. In the Stas-R method, the value of the response time obtained in one iteration of this fixed-point approach is used to calculate the number of preemptions in the next iteration. Hence, if the response time of a task exceeds its deadline, we stop the iterative calculation and do not report number of preemptions.

Widening gaps between results using our new methods (OurFP-RangeMax and OurFP-ItSpMax) and using the two prior methods (HJ-P and Stas-R) show an increase in the effectiveness of our methods as we proceed towards lower priority tasks. Since lower priority tasks are less likely to be scheduled in the initial intervals, more preemption points are deemed infeasible by our new methods, hence producing tighter bounds. This feature of our new methods prevents the exponential increase in the number of preemptions for successive tasks observed in the method HJ-P.

Analysis times using the method OurFP-RangeMax are indicated in Table VII. The most significant factors affecting the analysis time of a task are the number of memory accesses within the task and the actual loop nest structure of the task. The actual task set characteristics, which determine the number of jobs of each task in the hyperperiod of the task set, also contribute to this time, albeit in a much less significant way.

Utilization = 0.5		**Utilization = 0.8**	
Task Set Size	**Analysis Time**	**Task Set Size**	**Analysis Time**
2	33.65	2	558.44
4	177.80	4	626.47
6	175.20	6	436.49
8	417.85	8	419.33
		10	415.11

Table VII. Analysis times in seconds

Between the two base utilizations whose results are shown in Figure 9, we observe that the 0.8 utilization shows a higher number of preemptions than the 0.5 utilization. This is due to the increased WCET in the case of higher utilization. Increased WCET means that the tasks have a greater number of feasible preemptions once they have started execution and, hence, the response times of tasks increase. It may be observed from the results that the WCET bound of a task does not depend significantly on its priority unlike the response time. This is due to the fact that the stand-alone or base WCET dominates the total preemption delay cost. Thus, the WCET with preemption delay does not necessarily increase monotonically with decreasing priority.

From our results, several observations can be made regarding the two prior methods (HJ-P and Stas-R). While both of them produce very similar results for the first two tasks in a task set, they start to exhibit differences as we proceed towards lower priority tasks. The Stas-R method consistently performs better than the HJ-P method since it correctly takes effects of indirect preemptions into account.

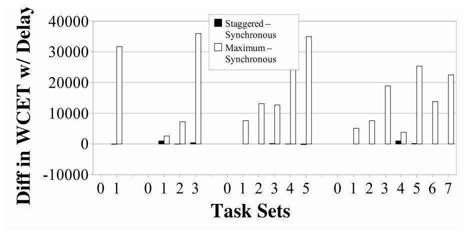
As already observed, the new methods OurFP-RangeMax produces significantly tighter bounds than the two prior methods in *all* cases. Furthermore, the method OurFP-ItSpMax, which produces an upper bound for the number of preemptions for any phasing of tasks, also usually performs significantly better than both prior methods. However, in the case of the task with second-highest priority, it gives a higher number of preemptions (and, hence, WCET with delay and response time) than the Stas-R method. This is explained as follows. Since the task with second-highest priority has only one higher-priority task, the Stas-R and OurFP-ItSpMax methods actually calculate the same number of preemptions. However, in OurFP-ItSpMax, we add one extra preemption (and, hence, extra preemption delay) to account for any possible phasing. This makes the number of preemptions higher than that produced by Stas-R.

To illustrate the variation in the maximum number of preemptions obtained by our new method OurFP-RangeMax between the various jobs of a task, we provide results for two of the task sets in Table VIII. As already observed from the graphs in Figure 9, OurFP-RangeMax always produces a *significantly* lower value for the number of preemptions than that produced by the two prior methods. Moving towards lower priority tasks, we further observe that there is a difference between the number of preemptions for different jobs of the same task by noting that the minimum, maximum and average values obtained over all the jobs are different from each other. In the case of the task set with base utilization 0.8, we notice that number of preemptions for certain tasks are not reported in the Stas-R method. This is for the same reason already explained with reference to the graphs in Figure 9 — the number of preemptions could not be calculated since the response times of those tasks exceeded their respective deadlines.

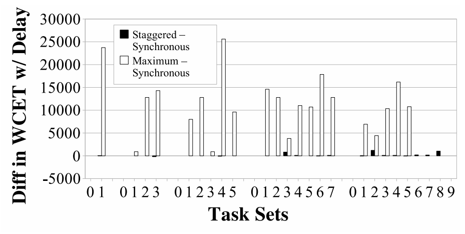
Benchmark	**Period (cycles)**	**WCET (cycles)**	**BCET (cycles)**	**# Jobs**	**# P OurFP-RangeMax**			**# P HJ-P**	**# P Stas-R**
					avg	min	max		
U = 0.5									
200convolution	100k	14191	14191	40	**0**	**0**	**0**	0	0
300convolution	400k	20891	20891	10	**0**	**0**	**0**	4	1
500convolution	500k	34291	34291	8	**0**	**0**	**0**	7	2
300n-real-updates	800k	56538	47338	5	**0.2**	**0**	**1**	12	4
matrix1	1000k	59896	54015	4	**1**	**1**	**1**	17	6
600fir	2000k	54837	52537	2	**1**	**1**	**1**	34	8
800convolution	2000k	66191	54391	2	**1**	**1**	**1**	35	14
900lms	4000k	158636	118536	1	**3**	**3**	**3**	71	27
U = 0.8									
n-real-updates	100k	16738	16838	50	**0**	**0**	**0**	0	0
900convolution	625k	76391	61091	8	**0.75**	**0**	**1**	7	1
matrix1	625k	59896	54015	8	**1**	**1**	**1**	8	3
1000convolution	625k	87091	67791	8	**1.25**	**1**	**2**	9	5
600convolution	1000k	45291	40991	5	**0.6**	**0**	**2**	16	7
300n-real-updates	1000k	56538	47338	5	**1.4**	**0**	**3**	17	10
800fir	1250k	77037	69737	4	**1.75**	**1**	**2**	23	
900lms	1250k	158636	118536	4	**3.75**	**3**	**5**	24	
1000fir	2500k	99237	86937	2	**4.5**	**3**	**6**	47	
500fir	5000k	43937	43937	1	**1**	**1**	**1**	94	

Table VIII. Number of Preemptions (# P) for Task Set with U=0.5 and U = 0.8

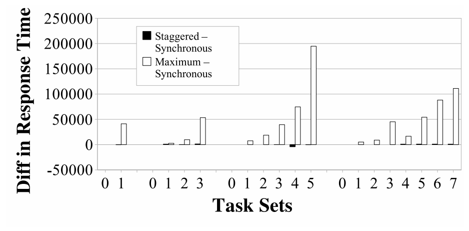
In our experiments, we also observed that the number of preemptions obtained for the first job of every task (released at the same time as all higher priority jobs) was not always the maximum value obtained across all jobs. This proves the claim we make in Section 4 about the critical instant *not* being the instant at which jobs of all tasks are released at the same time and underlines the necessity to perform our analysis for every job in the hyperperiod of a task set rather than once for every task.



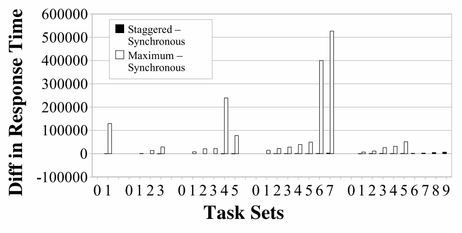
(a) Difference in WCET w/ delay for U = 0.5



(b) Difference in WCET w/ delay for U = 0.8



(c) Difference in Response Time for U = 0.5



(d) Difference in Response Time for U = 0.8

Fig. 10. Results for staggered task-set for U = 0.5 and U = 0.8

13.2 Task Sets with Staggered Releases

In our first set of experiments, we assume that all tasks in a task set are released simultaneously (synchronous release). However, since our analysis is capable of producing worst-case response time bounds for a task set with a particular phasing, we thought it useful to illustrate such a case with experimental results. For this purpose, we reuse the same task set characteristics from Table VI. However, in this experiment, we change the phasing of the tasks. Tasks in every task set are released in reverse order of priority. Every task has a maximum phase of 1000 cycles, or its own period, whichever is smaller.

In these experiments, the differences in the results of OurFP-RangeMax between the first set and the current set of experiments are not significantly different. For this reason, we decided to present the *differences* obtained in the bounds for the WCET and response time for task sets with synchronous releases and task sets with staggered releases. The differences are shown in Figure 10. The graph also shows differences between maximum possible values (obtained using OurFP-ItSpMax) and synchronous release.

For most tasks, there are very small changes in the values of WCET and response times for the phased task sets when compared to the difference between the maximum possible values and synchronous release. This is because, in OurFP-ItSpMax, we add extra preemptions to account for any possible phasing and assume the maximum possible delay at every one of them. Since the increase or decrease in WCET is entirely dependent on the relative positioning of jobs at different points in the hyperperiod, it is difficult to determine the phasing of a task set that would result in the worst possible response-times for tasks. This illustrates the merit of the method that calculates an upper bound on the number of preemptions *irrespective* of phasing (OurFP-ItSpMax).

13.3 Effects of WCET/BCET on # of Preemptions

In order to study the effects that the ratio of WCET of a task to its BCET has on the upper bound for the number of preemptions it incurs, we performed a set of experiments with *synthetic* task sets. We vary the ratio of WCET to BCET for every task, maintaining all other parameters. The results of this set of experiments are shown in Table IX. Since we use synthetic tasks, we do not have actual code from which to construct access chains for calculation of preemption delay. Hence, we need to assume a fixed value for the preemption delay. Since the preemption delay is significantly less than the base WCET of a task, we assume a delay value of 0 in our experiments for the sake of simplicity. Our primary goal in this set of experiments is to show how the WCET/BCET ratio affects the number of preemptions.

Task ID	**Period (cycles)**	**WCET (cycles)**	**# Preempts OurFP-RangeMax (Min/Max/Avg) W/B = 1**	**W/B = 1.5**	**W/B = 2**	**W/B = 2.5**	**W/B = 3**	**# P HJ-P**	**# P Stas-R**
U = 0.5									
1	80k	16k	1/1/1	1/1/1	1/1/1	1/1/1	1/1/1	8	2
2	100k	5k	0/1/0.25	0/1/0.25	0/2/0.5	0/2/0.5	0/2/0.5	12	4
3	200k	30k	3/3/3	3/4/3.5	3/4/3.5	3/5/4	3/5/4	25	8
U = 0.8									
1	80k	20k	2/2/2	2/2/2	2/2/2	2/2/2	2/2/2	8	3
2	100k	15k	1/2/1.5	1/3/1.75	1/3/1.75	1/4/2	1/4/2	12	6
3	200k	50k	6/7/6.5	8/8/8	8/9/8.5	8/9/8.5	8/8/8	25	19

Table IX. Preemptions for Task Set with U=0.8 with Varying WCET/BCET (W/B) ratios

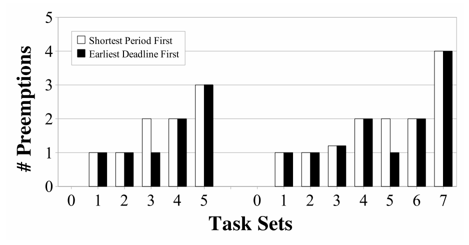
For utilizations of 0.5 and 0.8, we consider WCET/BCET ratios of 1, 1.5, 2, 2.5 and 3. As before, in every case, the bounds obtained by our method of eliminating infeasible preemption points is significantly lower than those obtained by the two prior methods (HJ-P and Stas-R). As the ratio of WCET/BCET increases, the upper bound on the number of preemptions increases slightly for small ratios. After a ratio of around 3, the number of preemptions start to decrease once again. However, the number of preemptions does not go below the number obtained with ratio equal to 1. This is expected since the schedule with WCET/BCET ratio of 1 has the least amount of slack.

The maximum increase in the number of preemptions as compared to the number with ratio equal to 1 was found to be approximately 30 percent. Hence, even if we set the BCET of a task to 0, the pessimism in the results obtained are not very significant. In fact, they would still be tighter bounds than those produced by the two prior methods. This is a useful observation since several timing analyzers only provide WCET bounds for a task, but not BCET bounds. Even in these cases, our analysis would be applicable and useful to obtain tight bounds on the worst-case number of preemptions.

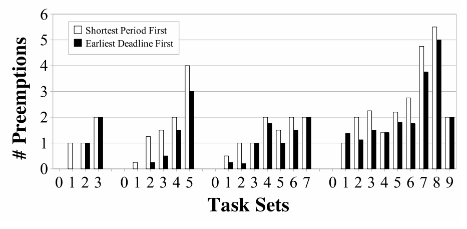
13.4 Static-Priority vs Dynamic-Priority Scheduling Policy

In all the above experiments, we use the RM scheduling policy, which is a static-priority scheduling policy. However, our framework is conceptually able to support dynamic-priority scheduling policies as well. In order to demonstrate this, we performed a set of experiments using the EDF scheduling policy. For this purpose, once again we used the task sets whose characteristics are shown in Table VI. Tasks in every task set were released in reverse order of the lengths of their periods (*i.e.*, in reverse order of priority as determined by the RM scheduling policy). A phase difference of 10 cycles was used between successive tasks. Figure 11 compares results obtained using both the RM and the

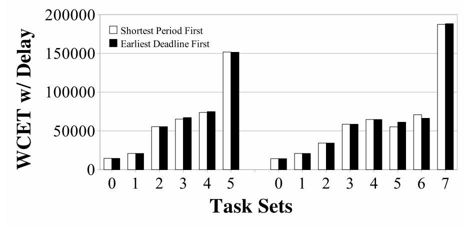
EDF scheduling policies for base utilizations of 0.5 and 0.8. Only the task sets that actually exhibit a difference in behavior between the two policies are shown.



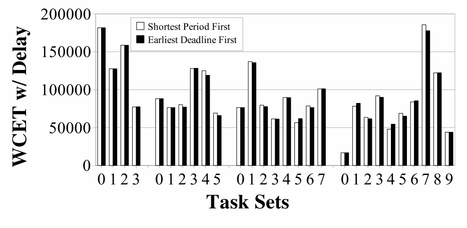
(a) # Preemptions for U = 0.5



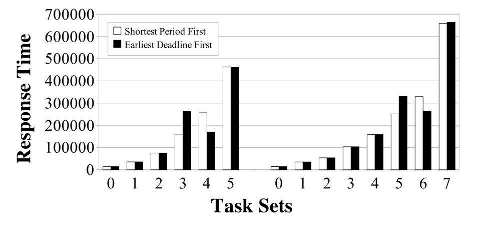
(b) # Preemptions for U = 0.8



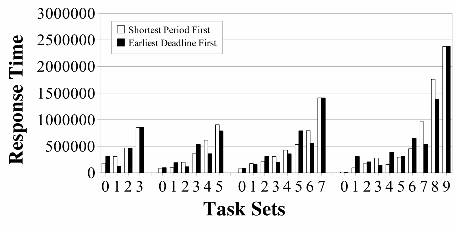
(c) WCET w/ delay for U = 0.5



(d) WCET w/ delay for U = 0.8



(e) Response Time for U = 0.5



(f) Response Time for U = 0.8

Fig. 11. Comparison of results for RM and EDF for U=0.5 and U=0.8

From the above results, we observe that, in some cases, the EDF policy decreases the number of preemptions for a task in comparison to RM, yet increases its response time. This is due to the fact that the relative deadlines of tasks alter their priorities. The experiments demonstrate the applicability of our method to systems with dynamic-priority scheduling policies.

14. CONCLUSIONS AND FUTURE WORK

In this work, we propose methods to calculate preemption delay suited to data caches and integrate it with past work in instruction cache and pipeline analysis. A framework developed in prior work is enhanced to calculate tight bounds for the data cache related preemption delay for real-time tasks. These bounds are used to calculate tighter bounds on WCETs and hence response times of tasks to determine its schedulability.

The contributions of this paper are:

(1) Calculation of an upper bound on the maximum number of preemptions for a given task;

(2) Calculation of a significantly tighter bound on the maximum number of preemptions using an algorithm that eliminates infeasible preemption points for tasks with given phasing, and an upper bound of the same for tasks with any possible phasing;

(3) Proof that the critical instant for a task set need not occur upon simultaneous release of all tasks when considering data cache related preemption delay;

(4) Construction of a realistic worst-case scenario for the placement of preemption points using the BCET and WCET of tasks for systems with static or dynamic scheduling policies.

We obtain significantly tighter bounds for (a) the number of preemptions, (b) the WCET and (c) the response time of a task as compared with prior methods. The improvements are up to an order of magnitude over two of the prior methods and up to half an order of magnitude over another. To the best of our knowledge, our work is novel in its contribution of a methodology to integrate data caches into preemption delay determination and in the consideration of critical instants for staggered releases of tasks.

As part of future work, we propose to conduct experiments to measure the WCET of a task using a cycle-accurate simulator. These values could then be compared to the results of our analysis in order to validate our results.

In light of the restrictions posed on the tasks and the characteristics of the task-sets being analyzed, it would be worthwhile to investigate the possibility of restricting areas where a task may be preempted. In other words, a task could have a region during its execution where is is non-preemptible. A sensitivity study may be conducted by varying the position and length of this region. We are currently pursuing this line of investigation [Ramaprasad and Mueller 2007].

Currently, if the data layout for a task changes, we can account for it only by recalculating the response times of all tasks. In future work, we could refine our approach to allow incremental changes to the data layout.

REFERENCES

AUDSLEY, A. N., BURNS, A., RICHARDSON, M., AND TINDELL, K. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 284–292.

BASUMALLICK, S. AND NILSEN, K. 1994. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*.

BURGER, D., AUSTIN, T., AND BENNETT, S. 1996. Evaluating future microprocessors: The simplescalar toolset. Tech. Rep. CS-TR-96-1308, University of Wisconsin - Madison, CS Dept. July.

BUSQUETS-MATRAIX, J. V. 1996. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*.

CHATTERJEE, S., PARKER, E., HANLON, P., AND LEBECK, A. 2001. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 286–297.

FRAGUELA, B. B., DOALLO, R., AND ZAPATA, E. L. 1999. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques*.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1997. Cache miss equations: An analytical representation of cache misses. In *Conference Proceedings of the 1997 International Conference on Supercomputing*. ACM SIGARCH, Vienna, Austria, 317–324.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems 21,* 4, 703–746.

JU, L., CHAKRABORTY, S., AND ROYCHOUDHURY, A. 2007. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *IEEE Design Automation and Test in Europe.*

KIM, S., MIN, S., AND HA, R. 1996. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Technology and Applications Symposium*.

LEE, C.-G., HAHN, J., SEO, Y.-M., MIN, S. L., HA, R., HONG, S., PARK, C. Y., LEE, M., AND KIM, C. S. 1998. Analysis or cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers 47(6)*, 700–713.

LEE, C.-G., LEE, K., HAHN, J., SEO, Y.-M., MIN, S. L., HA, R., HONG, S., PARK, C. Y., LEE, M., AND KIM, C. S. 2001. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering 27(9)*, 805–826.

LEHOCZKY, J., SHA, L., , AND DING, Y. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium*. Santa Monica, California.

LEUNG, J. Y.-T. A new algorithm for scheduling periodic, real-time tasks. to appear in *Journal of Algorithmica*.

LI, Y.-T. S., MALIK, S., AND WOLFE, A. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*. 254–263.

LIM, S.-S., BAE, Y. H., JANG, G. T., RHEE, B.-D., MIN, S. L., PARK, C. Y., SHIN, H., AND KIM, C. S. 1994. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*. 97–108.

LISPER, B. AND VERA, X. 2003. Data cache locking for higher program predictability. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 272–282.

LUNDQVIST, T. AND STENSTRÖM, P. 1999. Empirical bounds on data caching in high-performance real-time systems. Tech. rep., Chalmers University of Technology.

NEGI, H. S., MITRA, T., AND ROYCHOUDHURY, A. 2003. Accurate estimation of cache-related preemption delay. *ACM International Symposium on Hardware Software Codesign*.

PUAUT, I. 2006. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems*.

PUAUT, I. AND DECOTIGNY, D. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*.

RAMAPRASAD, H. AND MUELLER, F. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 148–157.

RAMAPRASAD, H. AND MUELLER, F. 2006. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 71–80.

RAMAPRASAD, H. AND MUELLER, F. 2007. Bounding worst-case response time for tasks with non-preemptive regions. Tech. Rep. TR 2007-22, Dept. of Computer Science, North Carolina State University.

STASCHULAT, J. AND ERNST, R. 2004. Multiple process execution in cache related preemption delay analysis. In *International Conference on Embedded Sofware*.

STASCHULAT, J. AND ERNST, R. 2006. Worst case timing analysis of input dependent data cache behavior. In *Euromicro Conference on Real-Time Systems*.

STASCHULAT, J., SCHLIECKER, S., AND ERNST, R. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*.

TOMIYAMA, H. AND DUTT, N. D. 2000. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. *ACM International Symposium on Hardware Software Codesign*.

VERA, X., LLOSA, J., GONZÁLEZ, A., AND BERMUDO, N. 2000. A fast and accurate approach to analyze cache memory behavior (research note). *Lecture Notes in Computer Science 1900*, 194–198.

VERA, X. AND XUE, J. 2002. Let's study whole-program cache behavior analytically. In *International Symposium on High Performance Computer Architecture*. IEEE.

WEGENER, J. AND MUELLER, F. 2001. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems 21,* 3 (Nov.), 241–268.

WHITE, R. T., MUELLER, F., HEALY, C., WHALLEY, D., AND HARMON, M. G. 1999. Timing analysis for data and wrap-around fill caches. *Real-Time Systems 17,* 2/3 (Nov.), 209–233.

ZIVOJNOVIC, V., VELARDE, J., SCHLAGER, C., AND MEYR, H. 1994. Dspstone: A dsp-oriented benchmarking methodology. In *Signal Processing Applications and Technology*.

Received March 2007;

```
/* 1. Description and initialization of variables used */
n: number of tasks
release_points: array of release points
timeline: array of tasks released at every release point
interval: time interval between two preemption points
bcet_rem, wcet_rem: array 1..n of remaining BC/WCET (init val=0)
curr_job: array 1..n of current job of every task
bcet_sum, wcet_sum: var. to sum up BC/WCET in interval
no_work_done, no_count, restart: array 1..n of bool (init to false)
num_p: array 1..n of max. # of preemptions for tasks
curr_priorities: array 1..n of current priorities
bneedtobreak, wneedtobreak: boolean values
/* 2. Loop iterating over intervals between release points */
for all rp in release_points up to hyper-period {
  for all tasks in system /* 3. Initialization of boolean variables for every interval */
    no_count[task], no_work_done ← false
  /* 4. Get tasks released at beginning of interval */
  tasks ← timeline[release_points[rp]]
  interval ← release_points[rp+1] - release_points[rp]
  for each element task of array of tasks released at current point { /* 5. Initialize released tasks */
    curr_job[task] ← curr_job[task] + 1
    restart ← true
    bc/wcet_rem[task] ← bc/wcet[task]
  }
  /* 6. Calculation of current priorities */
  curr_priorities ← calculate current priorities
  bcet_sum, wcet_sum ← 0
  bneedtobreak, wneedtobreak ← false
  for every task in order of curr_priorities { /* 7. Loop iterating over tasks in order of priority */
    if (restart[task]) {
      /* 8. Best case scenario calculation */
      if (bcet_rem[task] > 0) {
        bcet_sum ← bcet_sum + bcet_rem[task]
        if (bcet_sum ≥ interval) {
          for each lower priority task, lp_task
            no_count[lp_task], no_work_done[lp_task] ← true
          insert into array max_exec_times[task][curr_job[task]-1]
            value bcet_rem[task] - (bcet_sum - interval)
          bcet_rem[task] ← bcet_sum - interval
          bneedtobreak ← true
        } else {
          insert into array max_exec_times[task][curr_job[task]-1]
            value bcet_rem[task]
          bcet_rem[task] ← 0
        }
        no_work_done[task] ← false
      } else {
        insert into array max_exec_times[task][curr_job[task]-1]
          value 0
        no_work_done[task] ← true
      }
```

Fig. 12. Algorithm to Eliminate Infeasible Preemption Points

```
         /* 9. Worst case scenario calculation */
         wcet_sum ← wcet_sum + wcet_rem[task]
         if (wcet_sum ≥ interval) {
            for each lower priority tasks, lp_task {
               no_count[lp_task] ← true
               no_work_done[lp_task] ← true
            }
            if (wcet_rem[task] > (wcet_sum - interval)) {
               insert into array min_exec_times[task][curr_job[task]-1]
                  value wcet_rem[task] - (wcet_sum - interval)
               wcet_rem[task] ← wcet_sum - interval
               no_work_done ← false
            } else {
               insert into array min_exec_times[task][curr_job[task]-1]
                  value 0
               if (no_work_done[task] = true)
                  no_count[task] ← true
            }
            wneedtobreak ← true
         } else {
            wcet_rem[task] ← 0
            no_count[task] ← true
         }
         if (bneedtobreak AND wneedtobreak)
            break
      }
   }
   /* 10. Check if point marking end of interval is a
      feasible preemption point */
   for every task in order of curr_priorities {
      if (!no_count[task] AND !no_work_done[task]) {
         if (restart[task]) {
            if task released at end of interval has
               higher priority {
               num_p[task] ← num_p[task] + 1
               insert into preempting_tasks[task]
                  value curr_priorities[0]
            }
         }
      }
   }
   /* 11. If current job is done, reset variables for next job */
   for every task in order of curr_priorities {
      if (restart[task] AND ! wcet_rem[task]) {
         num_p[task] ← 0
         restart[task] ← 0
      }
   }
}
```

Fig. 13. Algorithm (cont.) to Eliminate Infeasible Preemption Points