4-2006

Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks

Harini Ramaprasad
Southern Illinois University Carbondale, harinir@siu.edu

Published in Ramaprasad, H., & Mueller, F. (2006). Bounding preemption delay within data cache reference patterns for real-time tasks. Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, 2006, 71 - 80. doi: 10.1109/RTAS.2006.14

Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks *

Harini Ramaprasad, Frank Mueller
Dept. of Computer Science, Center for Embedded Systems Research
North Carolina State University
Raleigh, NC 27695-8206, mueller@cs.ncsu.edu

Abstract

Caches have become invaluable for higher-end architectures to hide, in part, the increasing gap between processor speed and memory access times. While the effect of caches on timing predictability of single real-time tasks has been the focus of much research, bounding the overhead of cache warm-ups after preemptions remains a challenging problem, particularly for data caches.

In this paper, we bound the penalty of cache interference for real-time tasks by providing accurate predictions of the data cache behavior across preemptions. For every task, we derive data cache reference patterns for all scalar and non-scalar references. Partial timing of a task is performed up to a preemption point using these patterns. The effects of cache interference are then analyzed using a set-theoretic approach, which identifies the number and location of additional misses due to preemption. A feedback mechanism provides the means to interact with the timing analyzer, which subsequently times another interval of a task bounded by the next preemption. Our experimental results demonstrate that it is sufficient to consider the n *most expensive preemption points, where* n *is the maximum possible number of preemptions. Further, it is shown that such accurate modeling of data cache behavior in preemptive systems significantly improves the WCET predictions for a task. To the best of our knowledge, our work of bounding preemption delay for* **data caches** *is unprecedented.*

1. Introduction

A data cache is an invaluable architectural feature in today's higher-end processors. The savings it provides in terms of memory latency are immense. Hence, data caches have become indispensable. Nonetheless, caching has one inherent complexity, *i.e.*, the latency of data reference becomes unpredictable. While characterization of data cache behavior for a single task is complex enough, considering a preemptive scheduling system is even more complex.

* This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

In a preemptive system, a task may be interrupted at any time by a task with a higher priority. This implies that some cache blocks may potentially be evicted from cache and would need to be reloaded when the preempted task resumes execution. The main idea of this paper is to bound the delay caused due to preemptions for data caches and to derive an upper bound for the response time of a task.

In previous work, we proposed a method for analyzing data cache behavior for a single task [19]. We extended the concept of Cache Miss Equations to derive exact Miss/Hit patterns for every reference in a loop nest. We integrated this enhanced data cache analysis into our static timing analyzer framework.

In this paper, we further extend the work to consider a multi-tasking preemptive environment. We propose a method to obtain the worst-case data cache related preemption delay for every task in a given task set. This delay is added to the timing analysis results to derive a safe upper bound on the Worst Case Execution Time (WCET) of the task in the light of preemptions. We use the WCET thus obtained in a response time analysis (RTA) equation to calculate the response time of every task. Thus, we perform schedulability analysis on a task set. Any task whose response time is less than or equal to its deadline leads to a schedulable task set.

The fundamental contributions of our work are similar to those studied in instruction caches [20, 21], namely:

1. Preemption delay: Given the preempted task, the set of possible preempting tasks, and the preemption point, calculate the preemption delay that is incurred.
2. Number of preemptions: Calculate n, the maximum number of times a task can be preempted when it is executed as part of a given task set.
3. Worst-case scenario: Identify the placement of the n preemption points in the iteration space such that the worst-case total delay / preemption cost is obtained.

A method for calculating the preemption delay, given the preempted and preempting tasks, was proposed by Lee *et al.* [10]. This method was enhanced and the second and third points were newly contributed by Staschulat *et al.* [20, 21]. The difference between the ideas proposed by Staschulat *et*

Proceedings of the Twelfth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)
0-7695-2516-4/06 $20.00 © 2006 IEEE

al. and this paper lies in the methodology and in the application domain (instruction caches vs. data caches).

In the work by Staschulat *et al.*, the focus is on data-flow analysis to obtain the useful and used cache blocks for a task in order to calculate cache-related preemption delay. This is done by enumeration of possible cache states corresponding to every basic block in the control flow graph for a given task and relates primarily to the *instruction cache*.

In contrast, our method focuses on *data cache* analysis for loop-nest oriented code. Since the actual data reference (and, hence, memory and data cache locations accessed) is potentially different for every iteration of a loop nest, we cannot characterize data cache behavior in the same way as instruction cache behavior, based on cache states for a certain basic block. Hence, the method proposed by Staschulat *et al.* is not applicable in a framework such as ours.

We present a new technique to achieve the above stated goals. The technique is suited to data cache analysis and may be used in a similar fashion for instruction caches. The instruction cache analysis, however, has not been addressed in this paper and is subject to future work.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to the static timing analysis framework that we use for calculating the WCET for a task. Section 3 gives an overview of our previous work in analyzing data cache behavior. Section 4 explains in detail, the methodology used in our work. This is followed in Section 6 by a discussion of our experimental results. Section 7 contrasts our work to related work. Section 8 summarizes the results and Section 9 discusses future work.

2. Static Timing Analysis

Schedulability tests in real-time systems are generally based on the assumption that the WCET of every task in the task set is known *a priori*. These estimates need to be a *safe* upper bound on the execution times of tasks. As previous work has demonstrated, dynamic analysis by actual execution of the task does not guarantee worst-case performance [25]. Nor is exhaustive testing of the entire input space practical, as shown in the same study. Hence, static timing analysis is a viable approach to obtain WCET of tasks. Static timing analysis traverses all execution paths in a program and, during this process, calculates a conservative (*i.e.*, safe) upper bound on the time for the longest path in the program.

The structure of a program may cause a hurdle in the path of the analyzer due to factors like data-dependent control flow, pointer accesses, etc. Furthermore, *architectural features* also cause unpredictability for a timing analyzer. One such architectural feature, invaluable, but, at the same time, particularly hard to model, is the *data cache*. If data cache behavior cannot be predicted sufficiently accurately, WCET estimates may become highly pessimistic. Such predictions may be counter-productive since it may deem task sets infeasible that would otherwise be schedulable.

Figure 1 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 [19]) to produce data cache reference patterns.

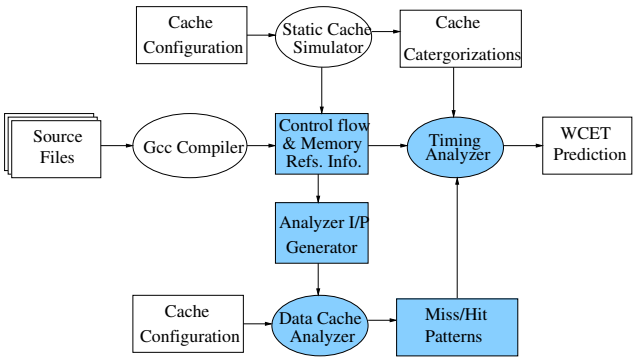


Figure 1. Static Timing Analysis Framework

3. Prior Work

In previous work [19], we enhanced a method by Vera *et al.* [23, 24] that statically analyzes data cache behavior using Cache Miss Equations [8]. This data cache analyzer was integrated into the static timing analysis framework as shown in Figure 1.

The data cache analyzer produces data cache access patterns, in terms of hits and misses, for every scalar and non-scalar memory reference in a given task. It is applicable in loop nest oriented code that adheres to certain constraints as specified elsewhere [19].

These patterns give us an accurate estimate of the number of data cache misses that the task incurs and their positions in the reference stream. In this work, since we only dealt with a single task, it was sufficient to provide the number of misses instead of the actual pattern of misses and hits to the static timing analyzer described in Section 2.

4. Methodology

While our prior work analyzes single tasks with respect to data caches, it does not take multi-task preemptive systems into account. In such a system, a task may be interrupted by higher priority tasks at arbitrary points during its execution. We consider non-partitioned data caches in our work. Hence, cache lines may be shared across tasks resulting in the eviction of a subset of existing memory lines from cache by preempting tasks. Assuming that all cache blocks brought in by the preempted task are evicted from

cache due to preemption (*i.e.*, the cache is effectively empty after every preemption point) leads to a significant overestimation of the Data Cache Related Preemption Delay (D-CRPD). Hence, schedulability of task sets may be adversely affected.

In this paper, we present a method to incorporate D-CRPD during WCET calculation itself. Furthermore, we make the calculation of the delay as accurate as possible by considering only the intersection of the cache blocks that are useful to the preempted task once it is restarted and those that are potentially used by preempting tasks.

4.1. Response Time Analysis

For this study, we constrain ourselves to response time analysis for determining schedulability of a task set [12, 1]. We assume a fixed-priority periodic task set where the deadline of a task is equal to its period. The calculation of response time involves an iterative approach using Equation 1.

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

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.

4.2. Phase 1: Calculation of Base Time and Data Cache Patterns

In this section, we describe the main process involved in our method, namely, computation of the WCET of all tasks. Our work is unique in that the WCET of each task is ensured to include the *data* cache related preemption delay due to preemptions by higher priority tasks. Since we incorporate the D-CRPD calculation into the calculation of C_i, we do not require an additional term for the delay in Equation 1.

In the first phase of the process, every task in a given task set is individually analyzed (*i.e.*, without considering preemptions) by the data cache analyzer to produce data cache miss/hit patterns for its references. Next, the timing analyzer framework is utilized to build a timing tree for every task in the task set. The timing tree provides information about the timing of individual nodes (functions/loops) in a given task. This phase constructs information required to calculate the WCET of every task. It is to be noted that the base time does not include the D-CRPD. Furthermore, this calculation is only performed *once* for every task in the task set.

4.3. Phase 2: Preemption Delay Calculation

In this phase, the data cache analyzer and the timing analyzer interact repeatedly for every interval between preemption points in a task in order to calculate the WCET of the task in the presence of preemptions. The interaction between the data cache analyzer and the timing analyzer is shown in Figure 2. The timing analyzer times the task up to the first preemption point. At this point, data cache analysis is performed to calculate the number of additional misses incurred due to the preemption. This delay is added to the base time. The timing analyzer is similarly invoked for every interval between preemption points.

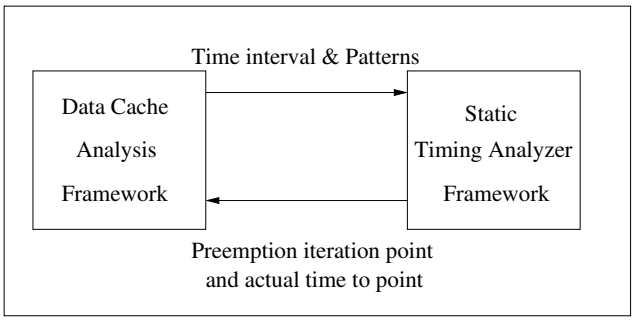


Figure 2. Interaction between Data Cache Analyzer and Static Timing Analyzer

4.3.1. Identification of Preemption Points There are two steps involved in the identification of preemption points.

Step1 : In this step, we calculate the maximum number of times each task may be preempted in the worst case. Consider a task *i*. For every task *j* that has a higher priority than that of task *i*, we subtract the total amount of time for which task *i* may be preempted by task *j* from the time remaining before the deadline of task *i*. The time remaining after this is used for consideration of further higher priority tasks. The formula shown in Equation 2 is repeatedly used for this calculation. Here, T_{rem} is the time remaining at every phase of the calculation, P_j is the period of the higher priority task, *j*, for the current phase of the calculation and C_j is its WCET. The initial value for T_{rem} is the relative deadline of the task *i*, which is equal to the period of the task *i* since we consider systems where the relative deadline is equal to the period for any task.

$$T_{rem} = T_{rem} - \lceil \frac{T_{rem}}{P_j} \rceil \cdot C_j \quad (2)$$

This process converges when no time is left prior to the deadline of task *i* or when there are no more higher priority tasks, whichever occurs first. The number of preemptions is then given by the sum of the $\lceil \frac{T_{rem}}{P_j} \rceil$ terms of each phase of calculation.

Proceedings of the Twelfth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)
0-7695-2516-4/06 $20.00 © 2006 IEEE

Step 2 : Next, we identify the actual placement of these preemption points that result in the worst-case preemption delay for a particular task. For this purpose, we have devised the following method:

All the data cache reference patterns of the task are merged, maintaining the order of access. All memory references in this consolidated pattern that access the same cache set are connected together to form a chain. Since the pattern maintains the order of access, this chain accurately indicates reuse. A chain that represents a particular cache line is colored with a unique color. An example with just three cache line chains is shown in Figure 3.

We identify points in the iteration space where a preemption would result in the largest cost, *i.e.*, by cutting the maximum number of differently colored chains. The n cuts with the largest cost are identified where n is the maximum number of preemption points incurred by the current task, as calculated in phase 1.

Weights are assigned to each point in the access chains of a task. The weight at a point is a direct indication of the number of additional data cache misses that would occur due to preemption at that point. The weight at a point is the number of differently colored chains that *cross over* this point. This already eliminates the cache lines which are no longer used after the point under consideration. In order to eliminate more infeasible points, we perform some additional checks while assigning weights.

1. We do not count chains in which the access point on the chain immediately following the current point is a MISS in the pattern. 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 as far as the particular cache set that the chain represents is concerned.
2. We do not count chains that correspond to a cache set that is not used by any task which has a higher priority than the task under consideration. This ensures that only the cache blocks that could potentially be replaced during preemption of the current task are considered.

Our method thus effectively considers only the intersection of useful cache blocks of the preempted task and the used cache blocks of the preempting tasks. Construction of the access chains is only required **once** for any task. The assignment of weights for every point in the access chain of a task is additionally dependent on the tasks that can potentially preempt the current task and, hence, is *task-set* specific rather than just *task* specific.

4.3.2. Actual Calculation of WCET and Response Time

In order to calculate the actual WCET of tasks, we first calculate the maximum preemption delay for every task using the method described above. Once we have the WCET of a task that includes the D-CRPD within it, we use the formula shown in Equation 1 to calculate the response time for the task. Since this formula requires the knowledge of the response times for tasks with higher priority than the current task, we start with the highest priority task and proceed towards the lowest priority task.

Since the highest priority task cannot be preempted, there is no need to calculate an additional delay (in second phase of our method) for that task and hence response time may be calculated directly from the equation. For the next highest priority task, we only need the response time of the highest priority task, and so on. In this way, we calculate response times for all tasks. These values may now be used to perform schedulability analysis on the task-set.

5. Experimental Framework

The tool set that we use in our experiments is the static timing analyzer framework, enhanced with a data cache analyzer that is responsible for producing data cache reference patterns for tasks according to our prior work [19]. We use this framework in conjunction with the generic PISA and SimpleScalar architecture [3].

The assumptions of the data cache analyzer are the same as those stated in prior work [19]. Fundamental among these are as follows. First, the loop bounds must be known at compile-time. Second, array subscript expressions must be affine functions of the loop induction variables. Third, there must be no dynamic or pointer-based memory accesses.

In our experiments, we use a 4KB direct mapped cache. Our experiments use the DSPStone benchmark suite [27]. In order to make the benchmarks statically analyzable by our framework, they were modified to replace pointer-based memory accesses with equivalent array accesses. Abstract inlining [19] was performed on the functions in the benchmarks to make each of them represent data references in only one (main) function. The benchmarks that we use in our task sets are briefly described in Table 1.

6. Experimental Results

The experiments we conducted are two-fold. We first study the behavior of the benchmarks with respect to the placement of preemption points while constructing the worst-case scenario. Next, we study the actual response time results for specific task sets.

6.1. Identification of Preemption Points

In this section, we discuss some observations about the identification of preemption points that would lead to the worst-case preemption delay.

As stated in Section 4, we build access chains for a task and calculate the costs of preemptions at every point. This cost is obtained by counting the number of chains which sat-

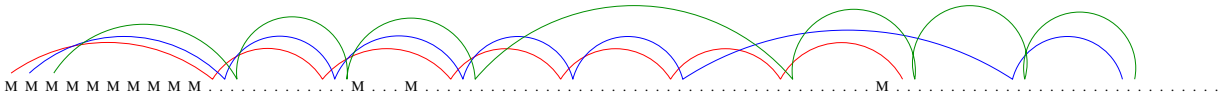
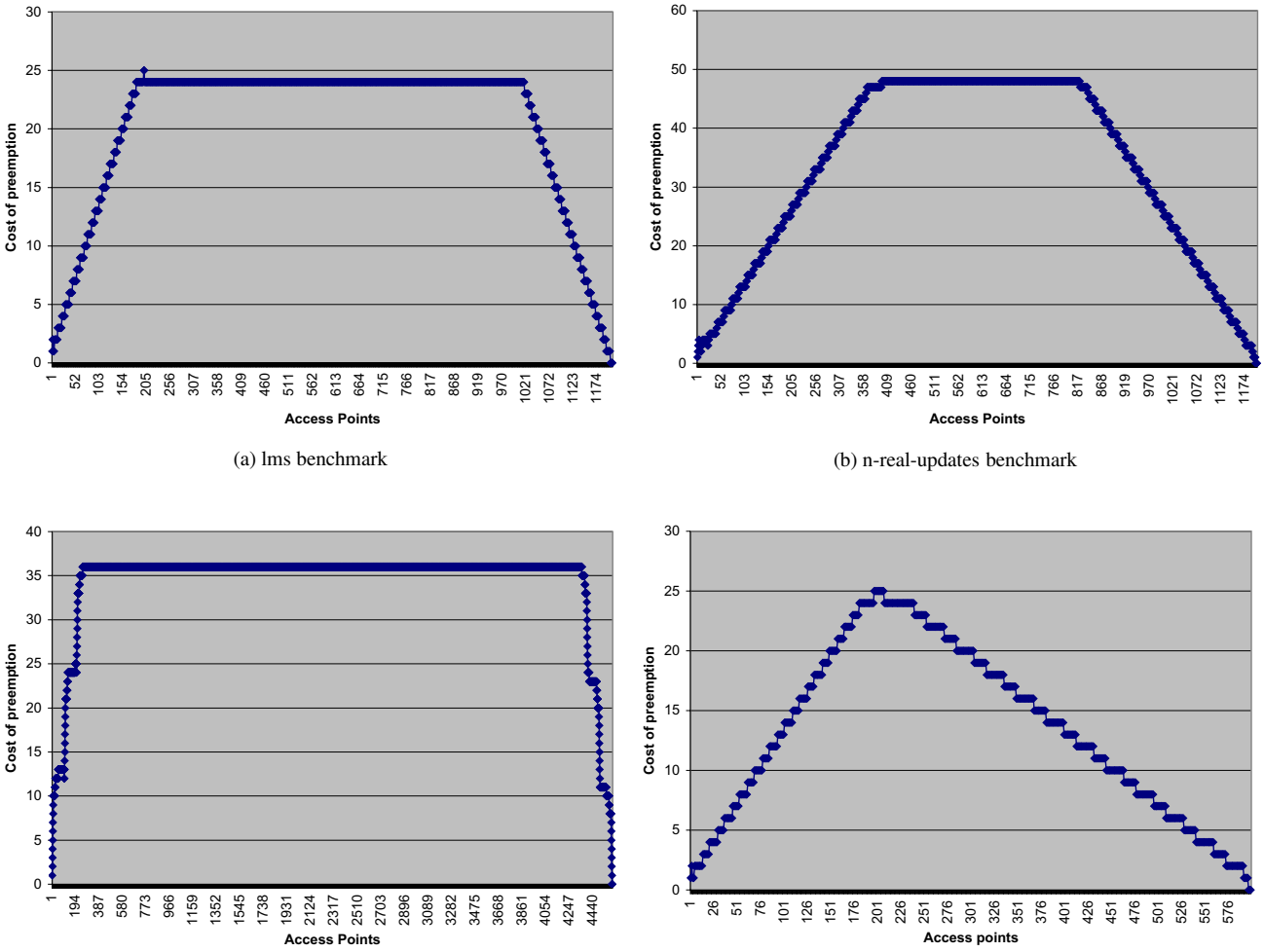


Figure 3. Cache Line Access Chains for Lines 1, 2 and 3



(a) lms benchmark

(b) n-real-updates benchmark

(c) matrix1 benchmark

(d) fir benchmark

Figure 4. Distribution of preemption costs across the iteration space

isfy the checks stated in Section 4. Among these costs, we currently choose the most expensive n points without placing a constraint on the interval between any two preemption points. The reason we do this is because of an observation we made regarding the usage of cache lines in most of our benchmarks.

The distribution of the costs of preemptions at access points for the second, third, fourth and fifth tasks in the second task set (see Table 3) are depicted in Figure 4. The X-axis shows the access points and the Y-axis shows the cost of preemption. The distribution proceeds in time order.

We can see that the number of access points with the highest cost is large and it is concentrated in consecutive access points for the benchmarks in Figures 4(a), 4(b) and 4(c). This means that a preemption at any of these consecutive access points is equally expensive. Hence, picking the n most expensive preemption costs irrespective of the distance between them gives reasonably tight estimates of the worst-case preemption delay.

The reason for this behavior stems from the general nature of programs. In most programs, ninety percent of the time is spent in ten percent of the code. Within this section of the code, there are usually very repetitive reuse-patterns and hence, lots of temporal and spatial reuse. At any point

Benchmark	Period (=deadline)	Stand alone WCET	Response time without delay	WCET with preemption delay	Response time with delay
dot-product	50000	750	750	750	750
convolution	62500	7491	8241	12491	13241
fir	125000	9537	17778	22037	35278
lms	125000	14536	32314	29136	77655
n-real-updates	250000	16738	48692	79138	235198
matrix1	250000	54168	111851	104568	greater than period

Table 2. Task set 1 - Characteristics and response times

Benchmark	Period (=deadline)	Stand alone WCET	Response time without delay	WCET with preemption delay	Response time with delay
convolution	62500	7491	7491	7491	7491
fir	125000	9537	17028	14537	22028
lms	125000	14536	31564	21936	43964
n-real-updates	250000	16738	48302	55138	106593
matrix1	250000	54168	109961	86568	244616

Table 3. Task set 2 - Characteristics and response times

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 1. Description of benchmarks in the DSPStone suite

during this section of code, all data that is used within the code is already in the data cache. Hence, preemption at any such point would result in more or less the same cache lines from being evicted, hence causing the same preemption delay.

In the graph in Figure 4(d), however, we observe a gradual increase up to some point and then a decrease in the cost between adjacent access points. Hence, we considered it beneficial to device a method for tightening the worst-case preemption delay bound for such distributions. This conceptual idea has not been implemented yet. The nature of the distribution is dependent on both the task itself and the position and priority of the task in the task set since that affects the cache lines replaced when the task is preempted.

As a first step, we identify the *n* most expensive preemption points. Then, we spread these points out into the rest of the iteration space based on certain constraints. Let us assume a simple task set with two tasks, T_x and T_y, where T_x has a shorter period (higher priority). Let P_x and P_y be the periods of the two tasks and R_x be the response time of the task T_x. Since T_x cannot be preempted, calculating its response time is straightforward. The concept behind an algorithm to identify the placement of preemption points is depicted in Figure 5. First, we pick one of the most expensive preemption points. This point is labeled as preemption point 1 in the figure. Once this preemption point is fixed, there cannot be any more preemptions of T_y by T_x for a time interval equal to the difference $P_x - R_x$. This is because new instances of T_x are released only at intervals equal to its period. Another constraint is that the next preemption point should be no later than a distance of P_x from the first preemption point. Hence, we place the next preemption point of T_y beyond the uninterrupted interval $P_x - R_x$, but before the end of interval P_x from the first preemption point. This range for the placement of the next preemption points is indicated in Figure 5. We choose the point within this range that causes the maximum preemption delay to be the next preemption point.

Consider a cluster of preemption points in a particular region of the access space for task T_y. While attempting to

Proceedings of the Twelfth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)
0-7695-2516-4/06 $20.00 © 2006 IEEE

move a preemption point to the space on the left of the cluster (*i.e.*, before the first preemption point in the cluster), we calculate the distances discussed above starting from the last preemption point in the cluster, moving backwards in the access space. Similarly, while moving preemption points to the right of the cluster, we calculate distances from the first preemption point in the cluster and moving forward in the access space.

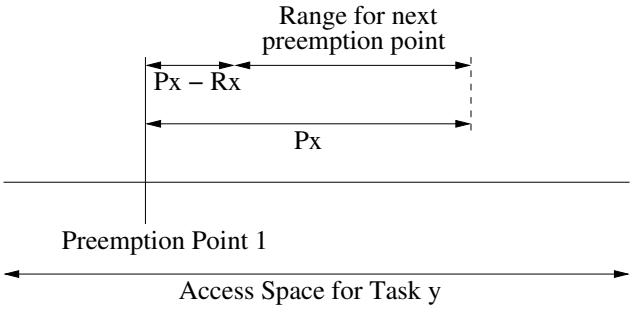


Figure 5. Distance criterion between preemption points

6.2. Calculation of Response Times

We performed experiments by constructing task sets of benchmarks to derive preemption delays for data caches. We use the Earliest Deadline First (EDF) scheme in our experiments. Further, we assume that the period of a task is equal to its relative deadline and that the phase of every task is 0. Our results in Tables 2 and 3 list the tasks used in our task sets in decreasing order of priority.

In our first experiment, we use the task set depicted in Table 2. The third column provides the worst-case execution time of the task as calculated by our static timing analyzer without considering preemptions (stand alone). The fourth column shows the response time calculated for each task using this worst-case execution time. The fifth column shows the WCET of the task with preemption delay added in and the last column shows the response time calculated using this new WCET.

As we can see from Table 2, the response time calculated using the stand alone WCET of each task suggests that this task set is schedulable since all tasks have response times shorter than their deadline. However, when we perform preemption delay calculation and incorporate that value into the calculation of WCET, we see that the task *matrix1* has a response time that is greater than its period. This indicates that the task-set is actually not schedulable and this stresses the importance of the preemption delay calculation.

A similar experiment with different task set characteristics yielded a schedulable task set. The inputs and results of this task set are shown in Table 3. All the values in Tables 2 and 3 are in number of cycles.

We also calculated the factors by which the response time increases with respect to the WCET with and without preemption delay incorporated. Column two in Tables 4 and 5 show the ratio of response time without preemption delay to the WCET without preemption delay. The third column in the same tables show the ratio of response time with preemption delay to the WCET with preemption delay. We see from the second and third columns in Tables 4 and 5 that there is, a very insignificant increase or decrease in the factor obtained while considering preemption delays as compared to the one without preemption delay. Hence, by adding preemption delay, we are not increasing the response time by a significantly different factor as compared to calculations without preemption delay.

However, when we calculate the ratios between the WCET with delay and the WCET without delay (shown in the fourth column of Tables 4 and 5), we observe significantly larger factors. While the ratio is high in general, the benchmark n-real-updates has particularly high ratios in both task sets. The reasons for this general and specific behavior are discussed below.

1. We calculate the maximum number of preemptions without taking indirect preemption effects (preemption by a higher priority task can affect tasks other than the task with second highest priority task only once until the completion of the task with second highest priority) into account. Hence, the value is pessimistic.

2. We choose the n most expensive preemption points without regard to the minimum distance between them. In most cases, this turns out to be n consecutive points that have the highest cost. The discussion in Section 6.1 already showed that, by spreading preemption points, a more realistic worst-case scenario can be obtained. This would reduce the preemption delay added for certain benchmarks.

3. The benchmark n-real-updates has a large fraction of temporal reuse that is adversely affected by repeated preemptions.

Obtaining tighter bounds for the maximum number of preemptions and constructing a more realistic worst-case scenario are part of future work.

7. Related Work

Several methods that bound data cache behavior have been proposed. Lim *et al.* [14] propose a method that takes data caching into account while computing the WCET for tasks for static memory references. Kim *et al.* [9] propose a method that classifies data references as static or dynamic. Data flow analysis is used by Li *et al.* [13] to analyze data cache behavior. White *et al.* [26] propose a method for direct-mapped caches based on static cache simulation.

Benchmark	Resp. time w/o delay / stand alone WCET	Resp. time with delay / WCET with delay	WCET with delay / stand alone WCET
dot-product	1	1	1
convolution	1.1	1.06	1.67
fir	1.87	1.6	2.31
lms	2.22	2.6	2
n-real-updates	2.91	2.97	4.73
matrix1	2.06	-	1.93

Table 4. Task set 1 - Ratios

Benchmark	Resp. time w/o delay / stand alone WCET	Resp. time with delay / WCET with delay	WCET with delay / stand alone WCET
convolution	1	1	1
fir	1.79	1.52	1.52
lms	2.17	2	1.51
n-real-updates	2.9	1.93	3.3
matrix1	2.03	2.8	1.6

Table 5. Task set 2 - Ratios

Lundqvist *et al.* [16] present a study that shows to what extent data cache accesses are predictable. They conclude that a majority of data cache accesses can be predicted.

Recently, some analytical methods for predicting data cache behavior have been proposed. They include the Cache Miss Equations by Ghosh *et al.* [8], a probabilistic analysis method proposed by Fraguella *et al.* [7] and another analytical method by Chatterjee *et al.* [5]. The common idea behind these methods is to characterize data cache behavior by means of a set of mathematical equations. In prior work [19], we have extended the cache miss equations framework to produce exact data cache patterns for references.

The above methods only deal with analyzing a single task and do not discuss multi-task preemptive scenarios. Some techniques that make data caches more predictable and can be applied in preemptive systems are cache partitioning and cache locking.

In cache partitioning [18], the cache is divided into smaller portions and the portions are used by individual tasks. Since each task has a dedicated cache portion, the question of a preemption replacing its cache blocks does not arise. The method has the disadvantage of having a smaller cache area at the disposal of individual tasks.

In cache locking [15, 6], selected data is loaded into cache and locked in place so that it may not be replaced until the cache is explicitly unlocked. During the locked interval, since the cache contents are known, cache behavior is predictable. This approach has the disadvantage that locking and unlocking introduce some overheads. Furthermore, if one task has locked certain cache lines, no other task that also uses those cache lines can take advantage of them. Finally, if some data is too large to fit into cache, it has to be completely unloaded from cache to make sure that the cache behavior is still predictable. This leads to a performance loss.

There are also several techniques that 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, but 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 [2]. 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) [4]. 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 [10, 11]. 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.

The work by Lee *et al.* was enhanced by Staschulat *et al.* [20, 21]. The authors propose a complete framework for the calculation of response time for tasks in a given task set. They address three issues involved in this calculation:

1. Calculation of preemption delay using a method very similar to that in the work by Lee *et al.* [10, 11].

2. Calculation of the maximum number of preemptions for a task.

3. Identification of the position of preemption points that would lead to the worst-case delay.

Their focus is, however, not on data caches, but on instruction caches. Hence, their methodology for performing the above calculations is completely different from ours.

Another approach by Tomiyama *et al.* calculates cache related preemption delay for the program path that requires the maximum number of cache blocks [22]. This path is determined by an integer linear programming technique. In this paper, only single preemptions are considered while multiple preemptions are not. Further, an empty cache is assumed at the beginning of a task. Negi *et al.* combined the techniques proposed by Tomiyama *et al.* [22] and by Lee *et al.* [10, 11] to develop an enhanced framework [17]. Once again, however, multiple preemptions are not considered and an empty cache is assumed at the beginning of a task.

8. Conclusion

This work provides a method to calculate cache related preemption delay that is specifically suited to data caches. We propose a framework that calculates bounds for the preemption delay within data cache reference patterns for real-time tasks. Using these bounds to calculate tighter estimates of the WCET of tasks, we perform response time analysis on all tasks in a task set to determine its schedulability.

We have devised a method that involves the following:

1. Derivation of data cache reference patterns for all scalar and non-scalar memory references in a task to analyze single-task data cache behavior.
2. Construction of data cache access chains from these patterns to calculate the delay due to preemption at a certain point in the execution of a task.
3. Determination of the maximum number of preemptions, n, for a given task in the context of a task set.
4. Identification of the n worst-case scenarios of preemptions. Currently, we choose the n most expensive preemption points for this purpose.

9. Future Work

As part of future work, we intend to enhance two aspects of the work proposed in this paper.

Currently, we calculate the maximum number of preemptions possible by checking how many higher priority tasks can be activated in the period of time between the release and the deadline (equal to period) of a lower priority task. This number is pessimistic since it does not take indirect preemptions into account. Thus, in our current analysis, the same instance of a preempting task may be considered in the preemption delay calculation of more than one task. We intend to tighten the bound on the maximum number of preemptions possible for a given task.

A second step involves construction of a more realistic worst-case scenario for preemption by spreading out preemption points in the iteration space while maintaining safety. This would further tighten the estimated worst-case delay and, hence, the WCET bound of tasks.

References

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

[2] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[3] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin - Madison, CS Dept., July 1996.

[4] J. V. Busquets-Matraix. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*, June 1996.

[5] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, June 2001.

[6] D. Decotigny and I. Puaut. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, page 114, dec 2002.

[7] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[8] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[9] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Technology and Applications Symposium*, June 1996.

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

[11] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, Nov. 2001.

[12] J. Lehoczky, L. Sha, , and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium*, Santa Monica, California, Dec. 1989.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1996.

[14] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

[15] B. Lisper and X. Vera. Data cache locking for higher program predictability. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, Mar. 06 2003.

[16] T. Lundqvist and P. Stenstrm. Empirical bounds on data caching in high-performance real-time systems. Technical report, Chalmers University of Technology, 1999.

[17] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. *ACM International Symposium on Hardware Software Codesign*, Oct. 2003.

[18] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, 2002.

[19] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 148–157, Mar. 2005.

[20] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *ACM International Conference on Embedded Software*, 2004.

[21] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*, 2005.

[22] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. *ACM International Symposium on Hardware Software Codesign*, 2000.

[23] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior (research note). *Lecture Notes in Computer Science*, 1900:194–198, 2000.

[24] X. Vera and J. Xue. Let's study whole-program cache behavior analytically. In *International Symposium on High Performance Computer Architecture*. IEEE, Feb. 2002.

[25] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[26] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[27] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Signal Processing Applications and Technology*, 1994.