WCRT Analysis for a Uniprocessor with a Unified Prioritized Cache

Yudong Tan

Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
ydtan@ece.gatech.edu

Vincent J. Mooney III

Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
mooney@ece.gatech.edu

Abstract

In this paper, we investigate the problem of inter-task cache interference in preemptive multi-tasking real-time systems. A prioritized cache is used to reduce cache conflicts among tasks by partitioning the cache. Cache partitions are assigned to tasks according to their priorities. We extend a known tool, SYMTA, in order to estimate the Worst Case Execution Time of each task executing on a uniprocessor with a unified prioritized L1 cache. Furthermore, we apply a formal timing analysis approach to estimate the Worst Case Response Time (WCRT) of each task using the prioritized cache. Our WCRT analysis handles nested preemptions. WCRT using a prioritized cache is compared to using a conventional set associative cache of the same size and associativity. Our experiments show that the WCRT estimate can be reduced up to 26% when a prioritized cache is used.

Categories and Subject Descriptors C.3 [*Computer Systems Organization*]: Special-Purpose and Application-Based Systems

General Terms Algorithms, Reliability

Keywords Real-time System, Cache Design, Timing Analysis

1. Introduction

In real-time systems, we have to know in advance if each task can meet its time constraint. Especially for hard real-time systems, disastrous results may occur if tasks miss their deadlines. Usually, in a real-time system where there are no preemptions, we can use the Worst Case Execution Time (WCET) of a task to check if the task can be completed before its deadline [23]. In a multi-tasking real-time system where preemptions are allowed, we have to estimate the Worst Case Response Time (WCRT) for each task in order to check for satisfaction of timing constraints[1, 12, 20, 21]. WCET and WCRT analyses are complicated due to the fact that advanced features such as caching, pipelining and out-of-order execution are used in modern processors. For example, a cache introduces uncertainty in memory access time, which complicates cache-related timing analysis. This problem is worsened when preemptions are allowed in a multi-tasking system; the multi-tasking results in cache conflicts among different tasks. By customizing the cache allocation policy, we can reduce WCRT as well as reduce the complexity of cache-related timing analysis by removing or minimizing cache conflicts among tasks.

LCTES'05, June 15–17, Chicago, Illinois, USA.

Although customized caches can reduce task WCRT by minimizing cache conflicts, we still need to analyze the behavior of customized caches formally in order to safely apply customized caches to real-time systems. This paper explains an approach to evaluate some key trade-offs in cache design when designers are focused on the effect of cache type selection on WCET/WCRT estimation. We apply a known tool, SYMTA [23], to estimate the WCET of each task executing on a uniprocessor with a unified prioritized [16] L1 cache. Then, our formal WCRT approach as proposed in [17, 18] is extended to estimate the WCRT of each task in a preemptive multi-tasking system with a unified prioritized cache.

We also compare the performance of a prioritized cache with a set associative cache in terms of WCRT estimates. Three applications are used in our experiments. The experimental results show that the WCRT estimate can be reduced significantly by using a prioritized cache. A tight WCET/WCRT estimate allows a programmer to utilize computing resources in real-time systems more efficiently. For example, more tasks may be able to be scheduled or more complicated applications can be executed. As a result, the total utilization of CPU and other resources is improved.

2. Problem Statement

A typical embedded system usually includes memory, programmable components, reconfigurable logic and Application Specific Integrated Circuits (ASICs). Software programmable components can be typical RISC embedded processors or Digital Signal Processors (DSPs).

Hardware units such as ASICs and reconfigurable logic have strict timing properties. Their behavior is predictable. A processor provides a platform on which to run software, which is much easier to develop, thus having a short time-to-market period. Software design is more flexible in terms of design changes and product evolution. However, as compared to custom hardware, the execution time of software is more difficult to predict, especially when the target memory hierarchy has several levels.

Software applications can be accelerated significantly by using caches. Caches exploit temporal and spacial locality in memory access patterns. Cache performance is degraded when multiple memory reference streams with different localities compete for the same cache resources.

In a multi-tasking real time system, a set of tasks has to be completed before the corresponding task deadlines. We use the Worst Case Response Time (WCRT) of each task to analyze if each task can meet its deadline.

Definition 1. Worst Case Response Time (WCRT): The WCRT is the time taken by a task from its arrival to its completion of computations in the worst case. □

Notice that WCRT is different from Worst Case Execution Time (WCET) which is often used in timing analysis for single-task systems. WCET only includes the execution time of a task without considering preemptions, interrupts and context switch cost. WCRT, on the other hand, includes both execution time of the task

and additional time caused by preemptions, interrupts and context switches.

The WCRT of a task is affected by cache behavior. In a preemptive multi-tasking real-time system, each task usually has a priority. Low priority tasks can be preempted by high priority tasks. Example 1 shows two types of inter-task cache interference that can possibly degrade the cache performance.

Example 1: Figure 1 shows a scenario in a preemptive multi-tasking system. A low priority task, Task A, is preempted by a high priority task, Task B, at time t_1. During the preemption, Task B uses some cache lines that were used by Task A before the preemption. The memory blocks loaded to these cache lines by Task A are thus evicted. After Task A resumes at time t_2, it again needs to access some of the memory blocks evicted by Task B. Task A has to reload those memory blocks to the cache. Furthermore, after Task A resumes, Task A may evict some cache lines used earlier by Task B as well. When Task B is executed for the second time at time instant t_3, Task B also needs to reload some cache lines. Cache reload caused by inter-task interference extends the response times of Task A and Task B, as shown in Figure 1(B). □

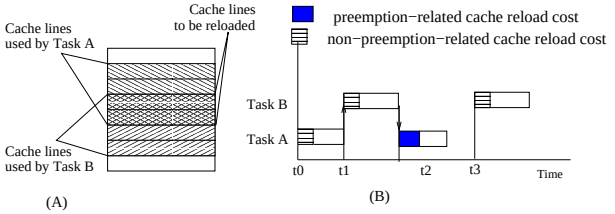

Figure 1. An example of inter-task cache eviction

Example 1 shows that inter-task cache interference can degrade the usefulness of a cache and can introduce additional cache reload cost to the WCRT of a task. Furthermore, cache reload cost increases uncertainty in memory access time, which worsens the timing analysis problem. Thus, we aim to avoid or reduce cache conflicts among tasks.

Usually, WCET/WCRT analysis assumes a cold cache when a task starts to run. This assumption is reasonable in a system where the cache is shared by all tasks because the cache lines used by one task may be evicted by other tasks. In the worst case, a task has to reload all data and instructions from the memory to the cache after a context switch. This cache reload cost due to the cold cache assumption applies to all multi-tasking systems, either preemptive or non-preemptive. Thus, we call this type of cache reload cost *non-preemption related cache reload cost*; such cost is already included in the WCET estimate for each task. Non-preemption related cache reload cost as shown in Example 1 affects both WCET and WCRT of tasks.

Notice that in a system where the cache is used by a task exclusively, the assumption of a cold cache is too conservative. After the first run of the task, the cache is filled with some data and instructions used by the task. Since the cache is not shared, no inter-task cache conflicts exist. When the task runs again later, the cache is already warmed up. In SYMTA [23], an iterative method is provided to calculate the minimum set of memory blocks guaranteed to reside in the cache because of previous executions.

In preemptive multi-tasking systems, preemptions can cause additional cache reload cost. As shown in Example 1, during Task B preempting Task A, some cache lines used by Task A before preemption are evicted, which causes cache reloading after Task A resumes. This cache reload overhead only happens in preemptive multi-tasking systems. Thus, we call this type of cache reload cost *preemption-related cache reload cost*. We propose an approach in [17, 18, 19] to analyze this type of cache reload cost.

Cache reload cost (including preemption-related and non-preemption related cache reload cost) is caused by cache interference among tasks. Cache interference can be reduced by customizing cache management policy. We propose a prioritized cache in [16]. In the prioritized cache, cache ways ("columns"[14]) are allocated to tasks dynamically according to the priorities of tasks. Each task uses its cache columns exclusively. Thus, cache conflicts among tasks are reduced. As distinct from all other cache partitioning approaches known to the authors, the prioritized cache takes the priorities of tasks into consideration. High priority tasks are given more privileges to use cache resources because it is usually more critical to meet the timing constraints of high priority tasks. Additionally, cache partitions are assigned to tasks adaptively according to the requirements of tasks. Users do not have to allocate cache partitions explicitly. Only minor modifications in the context switch function of the operating system are required to support the prioritized cache [16].

A prioritized cache can be used safely in a real-time system only if we can analyze the impact of a prioritized cache on the WCRT of tasks. In this paper, we aim to analyze the behavior of a prioritized cache formally by appropriately modifying the WCRT analysis approach as proposed in [17, 18]. This WCRT analysis approach is particularly well-tuned in its ability to analyze cache-related preemption delay. We integrate inter- and intra-task cache eviction analysis and give a new WCRT estimate formula. With our new approach, the WCRT estimate is tightened significantly. We also compare the prioritized cache with the conventional set associative cache.

We formally state the problem addressed in this paper. We assume we have a preemptive multi-tasking system consisting of n periodic tasks, T_0, T_1, ..., T_{n-1}. Each task, T_i, $(0 \leq i \leq n-1)$, has a period P_i. We assume that the deadline of task T_i is the same as its period. Each task, T_i, has a unique priority p_i. Task priorities can be derived by using a Fixed Priority Scheduling (FPS) algorithm (e.g., the rate monotonic algorithm) or can be assigned statically by designers. We also assume that tasks are sorted in the descending order of their priorities $p_0 < p_1 < ... < p_{n-1}$. For two tasks T_a and T_b, if $p_a < p_b$, Task T_a has a higher priority than Task T_b. The Worst Case Execution Time (WCET) of Task T_i can be estimated by using existing approaches such as SYMTA [23]. Here, we assume that there is no dynamic memory allocation and all memory addresses accessed by tasks are fixed. We further assume that only instructions residing in cache lines with instructions actually executed are loaded to the cache. In other words, we do not consider the impact of out-of-order execution, branch prediction and pipelining. We use C_i and R_i to denote the WCET and WCRT of Task T_i, respectively. We use a prioritized cache in such a preemptive multi-tasking real-time system. In order to know if Task T_i can meet its deadline, we need to estimate its Worst Case Response Time (WCRT). Such a WCRT estimate has to take into consideration the impact of a prioritized cache.

3. Prioritized Cache

Inter-task cache interference breaks spacial and temporal locality in memory access patterns because multiple tasks compete for the same cache resource. Memory access locality characteristics of different tasks are typically not very similar at all. An intuitive method to address this problem is to allocate cache lines to tasks exclusively so that cache lines used by one task cannot be used by other tasks.

In [16], we propose a prioritized cache. Briefly, a prioritized cache is a variant of a conventional set-associative cache. Figure 2 shows a prototype of the prioritized cache. Each way in an L-way set associative cache is viewed as a "column" in the prioritized cache [14]. The prioritized cache is partitioned and allocated to tasks at the granularity of columns. Each column can be set either "shared" or "not shared." If a column is not shared, it can be allocated to any task upon request. As soon as a column is allocated to a task, that task is called the owner of the column. As shown in Figure 2(A), extra registers are added to save the status of each

column. Memory blocks are mapped to a prioritized cache in the same way as a set associative cache. A memory address is split to three parts, tag, index and offset. The index selects a set in the cache. A set in the cache contains L cache lines. A cache line is selected by the cache replacement algorithm (e.g., LRU). The offset determines the location of the memory block in a cache line. A memory-to-cache mapping example is shown in Figure 2(B) which assumes a 4-way set associative cache with 16 sets. Each cache line has 16 bytes.

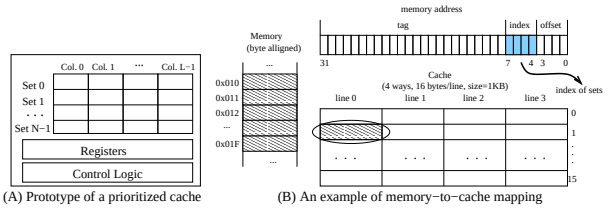

Figure 2. The prioritized cache

The main idea in the prioritized cache is to partition the cache among the the tasks. Cache columns allocated to lower priority tasks can be used by higher priority tasks. But cache columns allocated to higher priority tasks cannot be used by lower priority tasks. In this manner, the high priority tasks can minimize cache reload. As a result, the high priority tasks can be completed more quickly.

In the case of a cache hit, the prioritized cache behaves exactly the same as a conventional set associative cache. In the case of a cache miss, when a memory block needs to be loaded to the cache, a set in the cache is uniquely determined by the address of the memory block. Then, a cache line in that set is chosen from a column in the order as follows:
(1). Columns owned by the current task.
(2). Columns not owned by any tasks.
(3). Columns owned by lower priority tasks.
(4). Shared columns.

Example 2: Suppose we have three tasks, a Mobile Robot control application (MR), an Edge Detection (ED) application and an Orthogonal Frequency Division Multiplexing (OFDM) transmitter. MR updates the behavior of a robot periodically. ED processes images detected by the robot, and OFDM is used to communicate among robots. MR has the highest priority, and OFDM has the lowest priority. Also, we assume that a 4-way prioritized cache is used in the system and that one column is shared. Consider the scenario in Figure 3(A). OFDM runs first. Then, ED arrives and preempts OFDM. ED is then itself preempted by MR. When OFDM runs, it uses all columns. Column 0, Column 1 and Column 2 are owned by OFDM as shown in Figure 3(B). Column 3 is shared and cannot be owned by any task. After OFDM is preempted by ED, ED uses Column 0 and Column 1 because ED has a higher priority than OFDM. Now, Column 0 and Column 1 are owned by ED as shown in Figure 3(C). OFDM cannot load memory blocks to cache lines in Column 0 and Column 1. However, OFDM can still read cache lines in Column 0 and Column 1 in the case of cache hit. Similarly, after ED is preempted by MR, MR owns Column 0 as shown in Figure 3(D). □

The prioritized cache can allocate columns to tasks dynamically. The cache is partitioned at the granularity of columns. By changing the number of columns, we can partition the cache at different granularities. High priority tasks are given priority in cache usage. This strategy conforms to the characteristics of real-time systems because usually high priority tasks are more critical and thus have a greater requirement to guarantee their timing constraints.

We divide prioritized cache usage into two stages. In the first stage, the cache columns are allocated to tasks. Cache evictions may happen if columns used by low priority tasks are allocated to high priority tasks. In this stage, tasks run with a cold cache.

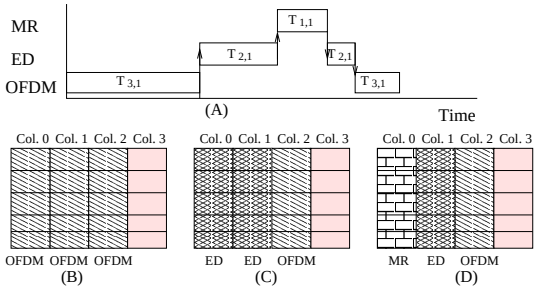

Figure 3. An example use of the prioritized cache

Because a task may have multiple feasible paths, the task may not execute all basic blocks (or equivalently, Single Feasible Path Program Segments as explained in SYMTA [23]) in one run. In other words, a task may request more cache columns after the first run. Thus, the cache columns owed by a task may change dynamically. A high priority task may acquire more cache columns and a lower priority task may lose cache columns subsequently. However, after all basic blocks of a task are executed at least once, cache columns allocated to this task become stable. Therefore, for the purpose of WCRT analysis, we run each task one or more times with carefully selected input data so that every basic block in the task is executed at least once in the first stage. Tasks are executed in this way in the descending order of task priorities. After this stage, cache allocation is completed. So, in the second stage, all tasks are allocated a portion of columns. The prioritized cache works in the second stage for the rest of time. Thus, we call the first stage *transit stage* and the second stage *stable stage*.

4. WCRT Analysis for a Prioritized Cache

In this section, we introduce our cache-related WCRT analysis approach proposed in [17, 18] briefly first. Then we modify this WCRT analysis approach to the prioritized cache.

4.1 Cache-related WCRT analysis

In [17, 18], we propose an approach to estimate the WCRT of each task in a preemptive multi-tasking real-time system. The tasks have properties as stated in Section 2. Our approach focuses on incorporating cache reload overhead into the WCRT estimate.

Suppose we have two tasks, T_a and T_b. T_b has a higher priority than T_a. Thus, T_a can possibly be preempted by T_b. As shown in Example 1, such preemption can bring cache reload overhead to the response time of T_a. Cache reload overhead caused by T_b preempting T_a is analyzed by calculating an upper bound on the number of cache lines to be reloaded by the preempted task T_a. Two conditions must be satisfied for memory blocks to have to be reloaded into the cache.

Condition 1. These memory blocks are used by both the preempted and the preempting task.

Condition 2. The memory blocks mapped to these cache lines are accessed by the preempted task before the preemption and are also required by the preempted task after the preemption (i.e., when the preempted task is resumed).

By using the simulation method as presented in SYMTA [23], we can find all the memory blocks that can be possibly accessed by tasks T_a and T_b. The memory blocks that satisfy Condition 2 are called "Useful Memory Blocks"[4, 5]. Lee et al. propose an intra-task cache eviction analysis approach to find useful memory blocks in the preempted task [4, 5]. We extend this concept to Maximum Useful Memory Block Set (MUMBS) [18, 19]. MUMBS is the maximum set (i.e., the union) of useful memory blocks over all the execution points of a task. Example 3 explains MUMBS.

Example 3: Suppose we have a task as shown in Figure 4. The task is represented with a Program Control Flow Graph as defined in [19]. The memory blocks accessed by the task are also shown in Figure 4. Let us consider the execution point s. Memory block

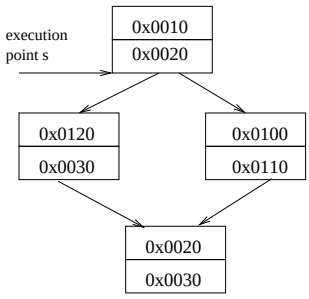

Figure 4. An example of MUMBS

$0x0020$ is accessed by the task before the execution point s as well as after the execution point s. By using the useful memory block analysis approach in [5], we can find that $0x0020$ is a useful memory block. The useful memory block set at the execution point s is $\{0x0020\}$. We perform the same analysis over all the execution points in the task and calculate the union of all useful memory blocks to obtain MUMBS. In this example, the MUMBS is $\{0x0020, 0x0030\}$. □

MUMBS contains all the memory blocks that can possibly cause cache reload cost in a task when this task is preempted at any possible execution point in the task. We can calculate the intersection set of MUMBS and the set of memory blocks that can possibly be accessed by the preempting task. The memory blocks in the intersection set satisfy both Condition 1 and Condition 2. In [17], we propose a Cache Index Introduced Partition (CIIP) of memory block sets. Let us briefly summarize CIIP.

Suppose we have a memory block set $M = \{m_0, m_1, ..., m_k\}$, where m_i is the address of a memory block. We also assume an L-way set associative cache with N sets. When a memory block is loaded into the cache, the set in which this memory is located is determined by the index in the address of this memory block. We use $idx(m_i)$ to represent the index of m_i. The index of the cache ranges from 0 to $N-1$. We can derive N subsets of M as follows.

$$\widehat{m}_i = \{m_k \in M | idx(m_k) = i\},\ (0 \leq i < N) \qquad (1)$$

We represent the CIIP of M with $\widehat{M}$, defined as below.

$$\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$$

where $\emptyset$ is the empty set and $\widehat{m}_i$ is defined as Equation 1.

Example 4: Suppose we have a memory block set $M_1 = \{0x0010, 0x0210, 0x1100\}$ and a two-way set associative cache. We have 16-bit memory addresses. The cache has 16 sets. Each cache line has 16 bytes, thus, bit 3 to bit 0 is the offset and bit 15 to bit 4 is the tag. In this case, we have $\widehat{m}_{10} = \{0x1100\}$ and $\widehat{m}_{11} = \{0x0010, 0x0210\}$. All the memory blocks in $\widehat{m}_{10}$ have the same index of 0. All the memory blocks in $\widehat{m}_{11}$ have the same index of 1. Note that in this example, $\widehat{m}_{1i} = \emptyset$ for all i such that $1 < i < 16$. In this example, $\widehat{M}_1 = \{\widehat{m}_{10}, \widehat{m}_{11}\} = \{\{0x1100\}, \{0x0010, 0x0210\}\}$. □

When the memory blocks in the same subset $\widehat{m}_i$ are accessed, these memory blocks are loaded into the same set in the cache because they have the same index. Thus, cache evictions can happen among these memory blocks (i.e., with the same index).

CIIP builds a bridge from memory blocks to a set associative cache without requiring knowledge the replacement algorithm in the cache. Based on the CIIP of the intersection set, we can estimate the cache reload cost caused by preemptions.

Suppose we have two tasks, the preempted task T_a and the preempting task T_b. T_a accesses the memory blocks in M_a, where $M_a = \{m_{a,0}, m_{a,1}, ..., m_{a,k_a}\}$. The MUMBS of T_a is $\tilde{M}$. T_b accesses the memory blocks in M_b, where $M_b = \{m_{b,0}, m_{b,1}, ..., m_{b,k_b}\}$. We use $\widehat{\tilde{M}}_a = \{\widehat{\tilde{m}}_{a,0}, \widehat{\tilde{m}}_{a,1}, ..., \widehat{\tilde{m}}_{a,N-1}\}$ to represent the CIIP of $\tilde{M}_a$ and $\widehat{M}_b = \{\widehat{m}_{b,0}, \widehat{m}_{b,1}, ..., \widehat{m}_{b,N-1}\}$ to represent the CIIP of M_b. We can estimate the number of cache conflicts between T_a and T_b by using the following formula.

$$S(\tilde{M}_a, M_b) = \sum_{r=0}^{N-1} min\{|\widehat{\tilde{m}}_{a,r}|,\ |\widehat{m}_{b,r}|,\ L\} \qquad (2)$$

where $\widehat{\tilde{m}}_{a,r} \in \widehat{\tilde{M}_a}$ and $\widehat{m}_{b,r} \in \widehat{M_b}$.

We assume cache miss penalty is fixed, which is represented with C_{miss}. Therefore, CRPD caused by T_b preempting T_a can be estimated as below.

$$CRPD(T_a, T_b) = S(\tilde{M}_a, M_b) \times C_{miss} \qquad (3)$$

CRPD estimate given in Equation 3 can be further tightened by using path analysis as proposed in [18]. However, this estimate does not consider nested preemptions. Let us consider the case in Example 5.

Table 1. Tasks in Example 5

Task	WCET(us)	Period(us)	Preemptions	CRPD(us)
T_0	5	20	T_1 by T_0	5
T_1	11	30	T_2 by T_0	2
T_2	12	100	T_2 by T_1	2

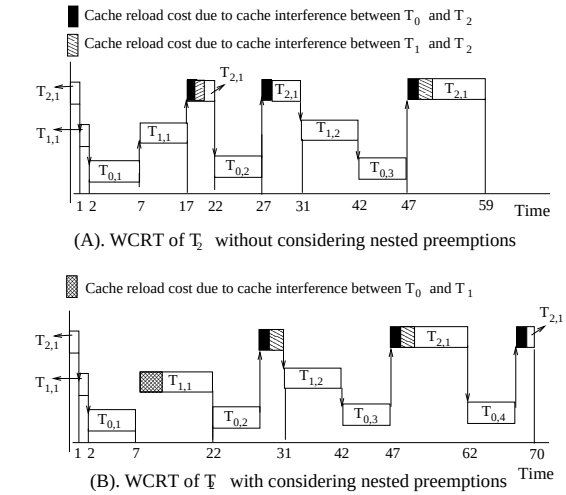

Figure 5. An example of nested preemptions

Example 5: Suppose we have three tasks, T_0, T_1 and T_2. WCET, period and cache reload cost for each task are listed in Table 1. Task T_2 has the has the lowest priority and Task T_0 has the highest priority. Here we ignore the context switch cost. At time instant 1 (measured in clock cycles; e.g., we assume a 100MHz clock), T_2 is preempted by T_1 directly. Then, at time instant 2, T_1 is preempted by T_0. The second preemption is nested in the first preemption. Thus, T_2 is also preempted by T_0, albeit indirectly, at time instant 2. We call this type of preemption an *indirect preemption*. Note that another indirect preemption occurs at time instant 42. In this case, the CRPD caused by T_0 indirectly preempting T_2 consists of two parts, cache reload cost due to cache interference between T_2 and T_0 and cache reload cost due to cache interference between T_1 and T_0 as shown in Figure 5(B). However, by using Equation 3, only cache reload cost due to cache interference between T_2 and T_0 is included in CRPD caused by T_0 indirectly preempting T_2, which is shown in Figure 5(A). Comparing Figure 5(A) with Figure 5(B), we find that cache reload cost due to cache interference between T_1 and T_0 can possibly extend the response time of T_2. Notice that when the cache reload cost due to cache interference between T_0 and T_1 is considered, the WCRT of T_2 is 70 (instead of 59). Thus, we need to include this factor in our WCRT analysis; in this specific case, while T_0 can arrive at most three times in 59 clock cycles, in fact T_0 can arrive up to four times in 70 clocks – as shown in Figure 5(B). □

Example 5 shows the effect of nested preemptions on WCRT. Note that in this example, we explain the difference between an *indirect preemption* and a *direct preemption*. When we estimate the WCRT of a task T_a, we need to consider all possible preemptions caused by each task, $T_b,\ 0 \leq b < a$, which has a higher priority than T_a. T_b can preempt T_a directly, which brings a cache reload cost of $CRPD(T_a, T_b)$ to the WCRT of T_a. T_a can also potentially be preempted by T_b indirectly if there exists a task T_l with a priority lower than T_b, but higher than T_a. In this case, when an instance of T_b arrives while T_a is preempted by T_l, T_l is further preempted by T_b. This indirect preemption introduces a cache reload cost of $CRPD(T_l, T_b)$ to the WCRT of T_a. In the worst case, T_{a-1} preempts T_a first, then T_{a-1} is preempted by T_{a-2}, ..., until finally T_{b+1} is then preempted by T_b. Thus, there are $a - b$ nested preemptions in the worst case.

In Equation 2, the number of cache conflicts between T_a and T_b results from a calculation using $\tilde{M}_a$, the MUMBS of T_a, and M_b, the memory blocks that are accessed by T_b. However, when nested preemptions exist, T_b may evict cache lines used by useful memory blocks of all tasks that have higher priorities than T_a but lower priorities than T_b. In order to include nested preemptions, we extend Equation 2 as follows (for additional details/examples, please see [19]).

$$S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b) = \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \widehat{m}_{l,r}|,\ |\widehat{m}_{b,r}|,\ L\} \quad (4)$$

By using Equation 4, we can derive a CRPD estimate formula which considers nested preemptions.

$$CRPD(T_a, T_b) = S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b) \times C_{miss} \quad (5)$$

Now, suppose we have as a known item the preemption related cache reload cost, $CRPD(T_i, T_j)$, then we can use an iterative method to estimate the WCRT of tasks. We use R_i^k to represent the WCRT of task T_i in the k^{th} iteration. The initial WCRT of T_i, R_i^0, is equal to C_i, which is the WCET of T_i.

$R_i^0 = C_i$;

$R_i^1 = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

...

$R_i^k = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

where $CRPD(T_i, T_j)$ is the cache reload overhead caused by T_j preempting T_i. C_{cs} is the context switch cost (number of cycles). We assume a constant upper bound on context switch cost in this paper. There are two context switches for each preemption, one for loading the preempting task and the other for resuming the preempted task.

This iteration terminates when R_i converges or R_i is greater than the deadline of T_i. If the iteration converges, T_i can be scheduled. Otherwise, we cannot find a feasible schedule. In short, using the iterative WCRT calculation approach presented above, we can analyze the schedulability of the system based on the WCRT estimate of each task.

Next, we propose a novel method of adapting the WCRT analysis approach for the prioritized cache. We are not aware of any prior work (other than the thesis of the first author [19]) in WCRT analysis for customized caches.

4.2 WCRT analysis for a prioritized cache

As stated in Section 3, we assume that we can use the behavior of a prioritized cache in the stable stage to determine the timing properties of tasks. Otherwise, to analyze the transit stage, the assumption of a cold cache can be used. In the stable stage, cache eviction only happens in the shared columns. Notice that we assume each task has a unique priority.

Now, let us summarize CRPD analysis for a prioritized cache in the stable stage formally. In a prioritized cache, tasks only conflict in the shared columns. The tasks that do not use shared columns do not have cache interference with other tasks in the stable stage. Thus, all preemptions related to these tasks do not incur CRPD. Since cache columns are allocated to tasks according to task priorities, high priority tasks have higher priority in using cache columns. Suppose we have a set of tasks $T_i,\ (0 \leq i < n)$ sorted in the descending order of their priorities. Here, n is the number of tasks. In other words, if $i < j$, T_i has a higher priority than T_j. In this case, if T_j can use some non-shared cache columns in the stable stage, T_i must own some non-shared cache columns as well. (Otherwise, T_i will occupy cache columns used by T_j since T_j has a lower priority.) Thus, in the stable stage, we divide tasks into two groups. In the first group, we have tasks T_0, T_1, ..., T_q, where $0 \leq q < n$. We use $\psi_1 = \{T_l | 0 \leq l \leq q\}$ to represent this group of tasks. All the tasks in this group do not use any shared columns. Thus, each task in ψ_1 does not conflict with any other task. In the second group, we have tasks T_{q+1}, ...,T_{n-1}. We use $\psi_2 = \{T_l | q < l < n\}$ to represent this group of tasks. The tasks in ψ_2 use shared cache columns; thus, each task in ψ_2 may conflict with other tasks in ψ_2.

Suppose we have two tasks, T_a and T_b. T_b has a higher priority than T_a. Let us consider two cases.

1. T_b is in the task group set ψ_1. In this case, T_b owns cache columns exclusively. As a result, the cache lines used by T_b do not overlap with the cache lines used by T_a; thus, $CRPD(T_a, T_b) = 0$.

Note that T_a has a lower priority than T_b. If T_a is in ψ_1, T_b must be also in ψ_1. Thus, we do not need to consider the case where T_a is in ψ_1, since the case of $T_a \in \psi_1$ is already covered by Case 1 above.

2. Neither T_a nor T_b is in ψ_1. In other words, both T_a and T_b use shared columns. In this case, cache eviction only happens in the shared columns. We can modify the CIIP-based approach proposed in [17, 18] to estimate the number of cache lines to be reloaded after T_a resumes from a preemption. The necessary modifications are shown in Equation 6 below, which is based on Equations 4 and 5.

$$CRPD(T_a, T_b) = \begin{cases} 0 & T_b \in \psi_1 \\ S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b) \times C_{miss} & T_b \in \psi_2 \end{cases} \quad (6)$$

We use Example 6 to explain how a prioritized cache affects cache interference among tasks.

Example 6: Suppose we have three tasks as stated in Example 2 and a 32KB 8-way prioritized cache. Each cache line has 16 bytes. Thus, each column has 256 cache lines. MR uses Column 0 to Column 4 and ED uses the rest of columns. The last two columns are set as shared. Thus, OFDM can only use the last two columns. OFDM has no conflicts with MR, but OFDM and ED may conflict in the shared columns in the prioritized cache. In this example, the estimate of an upper bound on the number of cache conflicts between OFDM and ED is 160.

If we use a conventional cache, all three tasks, MR, ED and OFDM conflict with each other in the cache. The estimate of cache conflicts between MR and OFDM is 88. The estimate of the number of cache conflicts between ED and OFDM is 98.

Assuming cache miss penalty is 10 clock cycles, the CRPD caused MR preempting OFDM is zero in a prioritized cache. As a comparison, the CRPD caused MR preempting OFDM is 880 clock cycles in a conventional set-associative cache. Therefore, a prioritized cache can be quite effective in preventing high priority tasks conflicting with low priority tasks. □

In Example 6, although the number of cache conflicts between ED, MR and OFDM is reduced, OFDM cannot use the full cache. Thus, the WCET of OFDM is expected to increase. We examine this impact in our experiments.

Now, let us consider how to adapt our WCRT analysis approach to a prioritized cache. Based on the CRPD given in Equation 6, we can modify our WCRT analysis approach for the prioritized cache.

For each task T_i, if $T_i \in \psi_1$, T_i does not conflict with any other tasks in the cache. For each preemption, we only need to consider the context switch cost and the WCETs of preempting tasks. Thus, the WCRT analysis formula introduced in Section 4.1 can be modified as follows.

$R_i^0 = C_i$;

$R_i^1 = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + 2C_{cs})$

...

$R_i^k = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + 2C_{cs})$

On the other hand, if $T_i \in \psi_2$, T_i may conflict with any other task in the task group ψ_2. But T_i does not conflict with any task in ψ_1. Thus, if T_i is preempted by a task T_j in ψ_1, for overhead (i.e., over and above actual worst-case task execution time C_j of T_j) we need only consider the context switch costs. If T_i is preempted by a task in ψ_2, we need to consider both CRPD and the context switch costs. Therefore, we use the formula below to estimate the WCRT of T_i.

$R_i^0 = C_i$;

$R_i^1 = C_i + \sum_{j=0}^{q} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + 2C_{cs}) + \sum_{j=q+1}^{i-1} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

...

$R_i^k = C_i + \sum_{j=0}^{q} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + 2C_{cs}) + \sum_{j=q+1}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

In this section, we adapt our WCRT analysis approach to apply to a prioritized cache. By using this new WCRT analysis, we can predict the worst case timing properties of a prioritized cache, which allows us to safely use a prioritized cache in a real-time system.

5. Previous Work

Inter-task cache conflict complicates WCET/WCRT analysis because of uncertainty in memory access time. This problem can be ameliorated by partitioning the cache and allowing each task to use a portion of the cache exclusively. Cache partitioning can be achieved with hardware approaches [3, 7, 14] or software approaches [11, 24].

Although customized caches show benefits in accelerating executions of applications in multi-tasking environments, a connection between formal WCET/WCRT analysis and customized caches is still missing. The authors of this paper have been unable to find any published work on WCET/ WCRT analysis for customized caches. Instead, the effectiveness of cache partitioning methods are evaluated with experiments [3, 7, 11, 14, 24]. Some typical benchmark applications are executed with cache partitioning approaches. The average execution time or the cache miss rate is used to measure the performance. In [13], Suh et al. give an analytical cache model to analyze the cache miss rate of a partitioned cache. This model predicts the overall cache miss rate of general applications. The worst case is not considered. Dropso et al. [2] compare some existing customized caches by using an analytical cache model. Again, the analysis targets the average performance of general applications.

Customized caches can be demonstrated to be effective in eliminating inter-task cache conflicts by running benchmarks or evaluating average performance (e.g., cache miss rate). However, lack of worst case analysis is not acceptable in real-time systems. Benchmark applications cannot cover all situations. Average performance cannot guarantee the same performance in the worst case. Thus, in order to apply customized cache in real-time systems safely, we need to analyze task WCET/WCRT formally.

Much work has been done in WCET analysis for a single task with a conventional cache (such as set associative or direct mapped) [6, 23]. In this paper, we use a WCET analysis tool, SYMTA[23]; however we can substitute any other WCET analysis approach for SYMTA.

In preemptive multi-tasking systems, a task may be preempted by another task. Thus, WCET cannot reflect the completion time of a task. Instead, we use a task's WCRT to evaluate if the task can meet its deadline.

Tindell et al. [20] propose a generic framework for WCRT analysis by using an iterative equation. Busquests-Mataix et al. extend Tindell's WCRT analysis approach by including cache eviction cost in a multi-tasking system [1]. They conservatively assume that all the cache lines used by the preempting task need to be reloaded by the preempted task upon resuming execution[1]. Other ILP-based WCRT analysis approaches can be found in [4, 5, 12, 17, 18, 21].

The WCET/WCRT analysis approaches introduced above only consider the behavior of conventional caches such as direct mapped caches or set associative caches. For customized caches, these approaches may be difficult to use directly (i.e., without significant modifications) because the behaviors of customized caches are not the same as conventional caches. For example, in conventional set associative caches that are shared by all tasks in a multi-tasking system, the WCRT of each task is heavily affected by inter-task cache conflicts. However, in some partitioned caches, inter-task cache conflicts possibly do not exist at all. Additionally, some customized caches are not easy to analyze formally. For example, in a data-replace-controlled cache presented in [7], specific instructions are inserted into the source code of applications to lock/release individual cache lines used by some data. Because data lock/release is done in the source code level and cache lines used by specific data is also related to the cache replacement algorithm, designers may not be able to know physically which cache lines are actually locked. As a result, inter-task cache conflict analysis is not straightforward. In [22], a cache locking approach is proposed to assist static timing analysis. However, this approach only addresses data caches while instruction caches are not considered [22].

Our prioritized cache is a variant of a set associative cache. The cache is partitioned at the granularity of columns. After the cache partitions are allocated to tasks, the prioritized cache behaves as a group of small size set associative caches. Thus, as explained in Section 4.2, the WCRT analysis approach designed for conventional set associative caches can be adapted easily to analyze the behavior of the prioritized cache.

6. Experimental Results

We use three applications to compare the prioritized cache with a conventional set associative cache. All caches used have size 32KB and have eight ways. A cache line has 16 bytes. In the prioritized cache, two ways are set as shared. Only the cache line replacement algorithms are different (i.e., one is prioritized while the other is LRU). Notice that in our experiments, a single cache is used for both instructions and data, which means the cache is a unified cache. We do not need to distinguish an instruction cache from a data cache because our WCRT analysis is based on simulation and memory footprints. There is no need to differentiate memory accesses caused by loading instructions or data. The same approach explained in this paper can, however, be applied to an instruction cache or a data cache; all that is required is separation into a memory footprint for instructions and a memory footprint for data.

In our experiments, an ARM9TDMI processor with a clock frequency of 100MHz is integrated with a specific cache (prioritized or set-associative). Hardware is simulated with Synopsys VCS [15]. The instruction set simulator is XRAY [8]. The whole system including software and hardware is co-simulated with Seamless CVE [10] provided by Mentor Graphics. The simulation environment is shown in Figure 6.

The first experiment uses a mobile robot application as described in Example 2. Task periods and priorities are listed in Ta-

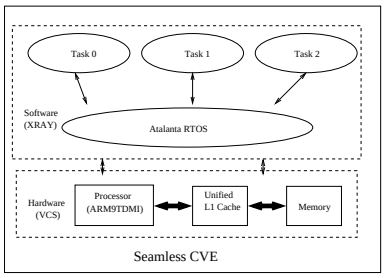

Figure 6. Simulation Architecture

ble 2. We use SYMTA to estimate the WCET of each task. The WCET of each task in the set associative cache (SA) and the WCET of each task in the stable stage of the prioritized cache (PC) are listed in Table 3. Note that the unit for all WCET/WCRT values presented in this paper is a clock cycle.

Table 2. Tasks in Experiment I

Task	Period(ms)	Priority
MR	3.5	2
ED	6.5	3
OFDM	40	4

Table 3. WCET with different caches

Tasks	MR	ED	OFDM
WCET in SA	842	1892	2830
WCET in PC	626	1676	4210

Three types of preemptions can happen in this system, MR preempting ED, MR preempting OFDM and ED preempting OFDM. The number of cache lines to be reloaded in these three preemptions are estimated in Table 4. Table 4 shows that no cache conflicts occur between ED and MR in the prioritized cache. This is also true for OFDM and MR. This means MR is assigned a partition of the prioritized cache exclusively. Because MR has the highest priority, OFDM and ED cannot use the cache assigned to MR. Thus, there are no cache conflicts among MR and other tasks (i.e., ED and OFDM). In this experiment, three columns are assigned to MR exclusively. It turns out that all cache lines required by MR fit into these columns. MR uses 80% of the SRAM available in these three cache columns. The other three non-shared columns are assigned to ED exclusively. ED also uses the two shared columns, which are also used by OFDM. Thus, there are cache conflicts between ED and OFDM. ED uses over 90% of the cache columns which ED can access (i.e., the three columns owned by ED and the two shared columns). OFDM uses 100% of the shared columns.

Table 4. Estimate of cache lines to be reloaded

Cache Type	Preemptions		
	ED by MR	OFDM by MR	OFDM by ED
SA	81	88	98
PC	0	0	160

Based on the WCET and the number of cache lines to be reloaded, we can apply the WCRT approach as proposed in [17, 18]. Because the impact of cache on the WCRT depends on not only the number of cache conflicts but also the cache miss penalty, we change the cache miss penalty from 10 cycles to 40 cycles. The comparison of WCRT of OFDM running with the set associative cache and the prioritized cache is shown in Table 5.

Two facts in these experimental results show the advantages of the prioritized cache as compared to the conventional cache. First, non-preemption related cache reload costs in high priority tasks are reduced. High priority tasks such as MR do not share any cache resources with other tasks; thus, we do not need to assume a cold cache for WCET analysis of these tasks after the first execution of the task upon startup/reboot. As a result, the WCET estimates of high priority tasks are tightened. For example, the WCET of MR – which is never preempted since MR has the highest priority – is reduced by 26% according to the WCET estimate results in Table 3.

Table 5. WCRT of OFDM with different caches

Cache Type	C_{miss}			
	10	20	30	40
PC	10260	11306	11626	11946
SA	9684	10264	12558	12966

In short, by using the prioritized cache, non-preemption related cache reload costs of high priority tasks are reduced. However, as we notice, the WCET of the low priority task, OFDM, is extended significantly because OFDM is restricted to use a limited portion of the cache; in our case, OFDM can only use the shared columns of the hot cache (i.e., after initial runs of tasks due to startup/reboot).

Second, the prioritized cache can also tighten the WCRT estimates of tasks because preemption-related cache reload overhead is minimized. As we can see from Table 4, there are no cache conflicts between ED and MR; neither are there any cache conflicts between OFDM and MR. Thus, CRPD caused by MR preempting ED and CRPD caused by MR preempting OFDM are both zero. However, both ED and OFDM use the shared columns. Thus, there is still CRPD caused by ED preempting OFDM.

We also compare the performance of the prioritized cache and the set associative cache with another application in which there are three tasks, Adaptive Differential Pulse Modulation Coder (ADPCMC), ADPCM Decoder (ADPCMD) and Inverse Discrete Cosine Transform (IDCT). This application is derived from MediaBench [9]. The periods, priorities and WCET for each task in listed in Table 6. As a result of eliminating cache sharing, the WCET of high priority task is reduced as well. For instance, the WCET of IDCT is reduced by 5%.

Table 6. Tasks in Experiment II

Tasks	IDCT	ADPCMD	ADPCMC
Periods(ms)	4.5	10	50
Priority	2	3	4
WCET in SA	1580	2839	7675
WCET in PC	1498	2830	11182

Table 7. Estimate of cache lines to be reloaded

Cache Type	Preemptions		
	ADPCMD by IDCT	ADPCMC by ADPCMD	ADPCMC by IDCT
SA	46	64	56
PC	0	0	0

Table 8. WCRT of ADPCMC with different caches

Cache Type	C_{miss}			
	10	20	30	40
PC	35686	35873	36001	36349
SA	34676	34967	38779	39775

The number of cache lines to be reloaded in each cache is listed in Table 7. Recall, as stated earlier, that we use caches with eight "ways" or "columns." In this experiment, IDCT uses two columns. ADPCMD uses three columns and ADPCMC uses the remaining three columns available, two of which the user preset as shared. IDCT uses 70% of the available memory in the two cache columns used by IDCT. ADPCMD and ADPCMC use 90% of the memory available in the cache columns each uses. In this application, there are no cache conflicts among tasks in the prioritized cache. This means both IDCT and ADPCMD use cache columns exclusively and do not require any shared columns. Only ADPCMC uses shared columns. The overall result is that there are no cache conflicts among these three tasks. The WCRT estimate of ADPCMC is shown in Table 8.

In this application, because all inter-task cache conflicts in the prioritized are eliminated, the preemption-related cache reload cost is zero. Thus, the WCRT of ADPCMC is not affected by cache miss penalty. As it turns out, when the cache miss penalty is big enough so that the preemption-related cache reload cost cannot be ignored in the conventional cache, the prioritized cache shows better performance in the WCRT even of low priority tasks. For example,

when the cache miss penalty is 40, the WCRT of ADPCMC with the prioritized cache is reduced by 8% as compared to an equivalent set-associative cache.

The third experiment contains six tasks: OFDM, ADPCMC, ADPCMD, IDCT, ED and MR. Table 9 lists the priority and period of each task. Note that in order to satisfy the necessary condition of schedulability of a real-time system (i.e., the total CPU utilization of all tasks must be less than 100%), we increase the periods of some tasks as compared to the same tasks in the first two experiments. ADPCMC has the lowest priority and MR has the highest prority.

Table 9. Tasks in Experiment III

	T_1 MR	T_2 IDCT	T_3 ED	T_4 ADPCMD	T_5 OFDM	T_6 ADPCMC
Period(ms)	7	9	13	20	40	50
Priority	2	3	4	5	6	7

In this experiment, we set the cache miss penalty to 30 clock cycles. Figure 7 compares the WCRT of each task with a set associative cache and a prioritized cache.

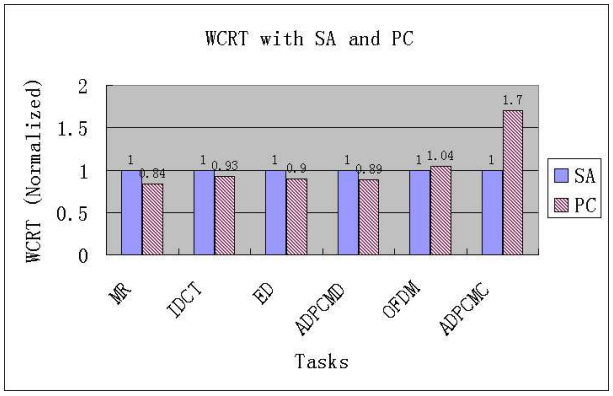

Figure 7. Comparison of task WCRT with a SA and a PC

Apparently, by using a prioritized cache, the WCRT of high priority tasks can be reduced because high priority tasks are allocated cache columns exclusively. On the contrary, low priority tasks have to use shared columns. Thus, a prioritized cache improves the performance of high priority tasks at the cost of the performance of low priority tasks. As shown in Figure 7, the WCRTs of MR, IDCT and ED are reduced by between 7% and 26%. However, the WCRT of ADPCMC is increased by nearly 70%.

7. Conclusion

In this paper, we compare the set associative and the prioritized cache in terms of their impact on the Worst Case Response Times of tasks in a preemptive multi-tasking real-time system. As best we can tell, we are the first authors to publish a formal WCRT analysis approach to analyze the behavior of a customized cache (specifically, the prioritized cache). The prioritized cache reduces inter-task cache interference. As a result, the WCRT is tightened significantly. Our experiments show a reduction up to 26% in WCRT estimate by using the prioritized cache versus using conventional set-associative cache of the same size and associativity. With a tighter WCRT estimate, we can schedule more tasks in a real-time system and thus increase the utilization of computing resources.

For future work, we need to consider the case in which more than one task have the same priority. In this case, tasks with the same priority share a cache partition. Due to preemptions among tasks with the same priority, cache reload overhead caused by preemption may be different. Also, more experiments will be performed in the future to investigate the impact of cache size on WCRT estimation of tasks executing on a processor utilizing a prioritized cache.

References

[1] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium*, pages 204–212, June 1996.

[2] S. Dropso. Comparing caching techniques for multitasking real-time systems. Technical report, University of Massachusetts, Amherst, UM-CS-1997-065, November 1997.

[3] D. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proceedings of the Real-Time Systems Symposium*, pages 229 –237, December 1989.

[4] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Enhanced analysis of cache-related preemption delay in fi xed-priority preemptive scheduling. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 187–198, December 1997.

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

[6] Y. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transaction on Design Automation of Embedded Systems*, 4(3):257–279, July 1999.

[7] N. Maki, K. Hoson, and A. Ishida. A data-replace-controlled cache memory system and its performance evaluations. In *Proceedings of the IEEE Region 10 Conference*, pages 471–474, April 1999.

[8] Mentor Graphics, XRAY® Debugger. http://www.mentor.com/xray/.

[9] MediaBench, http://cares.icsl.ucla.edu/MediaBench/.

[10] Mentor Graphics, Hardware/Software Co-Verifi cation: Seamless. Avaliable HTTP: http://www.mentor.com/seamless/.

[11] F. Muller. Compiler support for software-based cache partitioning. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compliers and Tools for Real-Time Systems*, pages 125–133, June 1995.

[12] H. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of ACM Joint Symposia CODES+ISSS*, 2003.

[13] G. Suh, S. Devadas, and L. Rudolph. Cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, pages 1 – 12, June 2001.

[14] G. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 116–127, September 2001.

[15] Synopsys, http://www.synopsys.com/products/simulation/simulation.html.

[16] Y. Tan and V. Mooney. A prioritized cache for multi-tasking real-time systems. In *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'03)*, pages 168–175, April 2003.

[17] Y. Tan and V. Mooney. Timing analysis for preemptive multi-tasking real-time systems. In *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pages 1034–1039, February 2004.

[18] Y. Tan and V. Mooney. Integrate inter- and intra- cache eviction anlaysis for preemptive multi-tasking real-time systems. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES'04)*, pages 200–206, September 2004.

[19] Y. Tan. *Cache Design and Timing Analysis for Preemptive Multitasking Real-Time Uniprocessor Systems*. PhD Thesis, Georgia Institute of Technology, April 2005. Available HTTP: http://etd.gatech.edu.

[20] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fi xed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, March 1994.

[21] H. Tomiyama and N. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the Eighth International Workshop on Hardware/software Codesign*, pages 67–71, May 2000.

[22] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pages 272–282, June 2003.

[23] F. Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

[24] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of the 3rd Workshop on Responsive Computer Systems*, September 1993.