Timing Analysis for Preemptive Multi-tasking Real-Time Systems with Caches

Yudong Tan and Vincent J. Mooney III

{ydtan,mooney}@ece.gatech.edu

Center of Research for Embedded Systems and Technology

School of Electrical and Computer Engineering

Georgia Institute of Technology

Atlanta, GA 30332

Abstract

In this paper, we propose an approach to estimate the Worst Case Response Time (WCRT) of each task in a preemptive multi-tasking single-processor real-time system with an L1 cache. The approach combines inter-task cache eviction analysis and intra-task cache access analysis to estimate the number of cache lines that can possibly be evicted by the preempting task and also be accessed again by the preempted task after preemptions (thus requiring the preempted task to reload the cache line(s)). This cache reload delay caused by preempting tasks is then incorporated into WCRT analysis. Two sets of applications are used to test our approach. Each set of applications contains three tasks. The experimental results show that our approach can tighten the WCRT estimate by 38% (1.6X) to 56% (2.3X) over prior state-of-the-art.

I. Introduction

Timing analysis is critical in a real-time system. Underestimating the execution time of a task may cause deadlines to be missed in practice, which might bring disastrous results. On the other hand, pessimistic estimates of execution times may lower the utilization of resources. However, advanced features in modern processors such as caching and pipelining complicate timing analysis. Lots of work has been performed to analyze the cache behavior in a single task system in order to predict the timing properties of the system. Although single-task based timing analysis can help us acquire insight about timing properties of tasks, lots of factors in a multi-tasking system are not taken into consideration which will definitely affect the accuracy of such timing estimates. In a preemptive multi-tasking system, timing analysis becomes even more difficult because of unpredictability of preemptions, the interaction among tasks such as inter-task cache evictions and the underlining scheduling algorithms.

In this paper, we give an approach to analyze the Worst Case Response Time (WCRT) of tasks. We target the single-processor preemptive multi-tasking system with set associative caches. The approach focuses on the cache reload overhead caused by preemption and imposed on the preempted task. A novel method is proposed to analyze inter-task cache eviction. Inter-task cache eviction behavior analysis is then combined with intra-task cache access analysis of the preempted task to estimate the number of cache lines to be reloaded by the preempted task. Furthermore, path analysis is applied to the preempting task in order to tighten the result. After acquiring the WCRT of each task, we can further analyze the schedulability of the system. Two sets of applications are used to exhibit the performance of our approach. The experimental results show that our approach can reduce the estimate of WCRT up to 56% over prior state-of-the art.

The remaining of this paper is organized as follows. Section II introduces the previous work in the field of timing analysis. Section III introduces the problem and gives an overview of the approach presented in this paper.. Sections IV, V and VII give the details of our approach. Experimental results are presented in Section VIII. The last section concludes the paper.

II. Previous Work

A cache is one of main factors complicating timing analysis in real-time systems. Two categories of methods can be applied to predict cache behavior. One is limiting cache usage. This can be implemented by hardware approaches such as cache partitioning [2], [3], or, by software approaches such as compiler optimizations and memory remapping [4], [5]. Usually, these schemes need specialized hardware support in the cache controllers or TLBs as well as custom modifications to the compilers used. Moreover, cache utilization is compromised in these schemes, because either the cache allocation strategy is more strict than conventional caches such as in [2], [3] or the memory-to-cache mapping is more restrictive such as in [4], [5].

The second category of methods to predict cache behavior is to use static analysis methods. Such methods analyze cache behavior and make restrictive assumptions in order to predict Worst Case Execution Time (WCET) or Worst Case Response Time (WCRT) of tasks in real-time systems. Li and Malik contributed to WCET analysis by proposing an explicit path enumeration method [6]–[8]. They use Integer Linear Programming (ILP) techniques to limit the paths to be evaluated. Path analysis in their work is at the granularity of basic blocks. Wolf and Ernst extend the concept of basic blocks to program segments and developed a framework for timing analysis, SYMTA [9]–[12].The precision of time estimation is improved in SYMTA since the overestimate of execution time is reduced. [13] gave a clustered calculation approach to reduce the timing overestimate. This approach is similar to SYMTA in essence. [14], [15] proposed an abstract interpretation methodology to predict cache behavior. Stenstrom et al. [16] gave another static analysis approach based on symbolic execution techniques. In both Wilhelm's and Stenstrom's approach, WCET of programs can be analyzed without knowing the exact input data. All the aforementioned works focus on single task timing analysis. The problem becomes more complicated in a multi-tasking system, especially when preemption is allowed.

Timing analysis in multi-tasking systems is tightly related to scheduling techniques. In this paper, we assume that a Fixed Priority Scheduling (FPS) algorithm such as Rate Monotonic Algorithm (RMA) is used in the system [17], [18]. We further assume a single processor with a set associative L1 cache and secondary memory (the secondary memory can be either on- or off-chip). The purpose of timing analysis is to verify the schedulability of tasks. In this paper, we use the Worst Case Response Time (WCRT) [19] to analyze schedulability. Busquests-Mataix et al. propose an approach to analyze cache eviction cost in a multi-tasking system [20]. They conservatively assume that all the cache lines used by the preempting task need to be reloaded by the preempted task when the preempted task is resumed. Lee et al. also give an approach for cache analysis in preemptions [21], [22]. This approach counts the number of "useful" memory blocks by performing path analysis on the preempted task. However, they assume that all "useful" memory blocks of the preempted task are evicted from the cache by the preempting task, which might not be true. For example, if there are no dynamic data allocation in tasks and the cache lines used by the preempted task are disjoint with the cache lines used by the preempting task, the cache reload cost induced by preemption will be zero. But in Lee's approach, the cache reload cost is still the same as the cost to reload all "useful" memory blocks in the preempted task.

We proposed an approach for inter-task cache eviction analysis in [1]. This approach assumes that all cache lines used by the preempted task and evicted by the preempting task will be reloaded after the preemption. But, as presented in [21], only those cache lines used by "useful" memory blocks of the preempted task need to be reloaded.

Both the approach we presented in [1] and Lee's approach in [21] have their pros and cons. However, these two methods are complementary. Lee's approach calculates the maximum set of memory blocks in the preempted task that can possibly cause cache overload. But the preempting task is not considered. Our approach shows that the intersection of memory blocks accessed by the preempted task and the preempting task influences the cache reload overhead. However, we do not calculate the "useful" memory blocks in the preempted task. Thus, in this paper, we focus on enhancing our approach in [1] by incorporating "useful" memory block analysis in Lee's work. The new approach gives the most accurate WCRT method known to date for a multi-tasking single-processor system using set-associative or direct mapped unified caches. In Section VIII we will show examples where we achieve results up to 56% better than Lee's approach, and 38% better than our previous approach in [1].

III. Overview

In this section, we will state the problem formally first. Some terminology is defined for clarity. Then, we give an overview of the approach proposed in this paper.

A. Terminology

For clarity, we first define terminology we will use throughout the paper.

We assume that there are n tasks in the system, which are represented with $T_1, T_2, ..., T_n$. Each task T_i has a period P_i. T_i is ready to run at the beginning of its period. The deadline of T_i is at the end of its period. A fixed priority scheduling algorithm is used for scheduling; thus, each task has a fixed priority, p_i. The Worst Case Execution Time (WCET) of task T_i is denoted with C_i. This WCET can be estimated initially with existing analysis tools such as Cinderella [8] and SYMTA [9]. In this paper, we use SYMTA to derive the WCET of tasks. We will discuss later how to estimate WCRT on the basis of WCET in a multi-tasking system. Tasks are executed periodically. We use $T_{i,j}$ to represent the j^{th} run of Task T_i.

In a multi-tasking system,we aim at estimating the Worst Case Response Time (WCRT) of tasks, as defined in [19], for schedulability analysis.

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. The WCRT of task T_i is denoted by R_i. □

In a multi-tasking preemptive system, a task with a low priority may be preempted by a task with a higher priority. During a preemption, the preempting task may evict some cache lines used by the preempted task. When the preempted task resumes and accesses an evicted cache line, the preempted task has to reload the cache line from memory. This cache reload overhead caused by inter-task cache evictions increases the response time of the preempted task.

Example 1: We have three tasks T_1, T_2 and T_3. T_1 is a Mobile Robot control application (MR). The mobile robot updates its behavior every 3.5ms. The second task, T_2 is an Edge Detection application (ED) and is invoked every 6.5ms to process the images of obstacles detected by the robot. The third task, T_3, which is an OFDM transmitter, is invoked to communicate with other robots every 40ms. Figure 1 shows this example. In this example, three tasks arrive at time instant 0. However, T_3 is not executed until there are no instances of T_1 or T_2 ready to run. During the execution of T_3, it could be preempted by T_1 or T_2, which is shown in Figure 1. The response time of T_3 is the time from 0 to the time when T_3 is completed. We need to estimate the response time of such a task in the worst case. If we do not consider inter-task cache evictions, the WCRT of T_3 is shown in Figure 1(A). However, because of inter-task cache evictions, the preempted task has to reload some cache lines after preemption which impose an overhead on the WCRT of the preempted task. Figure 1(B) shows this issue. t_1, t_2 and t_3 are cache reload overhead in three preemptions respectively. Obviously, due to cache evictions, the WCRT of T_3 is extended, as shown in Figure 1(B) □

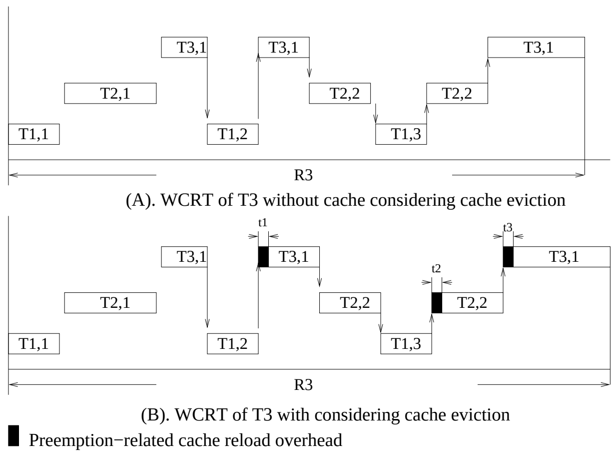

Fig. 1. Example of WCRT

As shown in Example 1, inter-task cache eviction affects the WCRT of tasks. In order to include inter-task cache eviction in the WCRT analysis for multi-tasking preemptive systems, we need to estimate the number of cache lines that need to be reloaded by the preempted task after each preemption. This paper aims to incorporate inter-cache eviction cost in the WCRT analysis by combining the inter-task cache eviction analysis as proposed [1] and Lee's approach in [21]..

In this paper, we will perform path analysis on the preempted task and the preempting task. The path analysis is based on a Control Flow Graph (CFG) which describes the control structure of a program. A CFG is represented with a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_m\}$ is the set of nodes and $E = \{e_1, e_2, ..., e_n\}$ is the set of edges. Each edge $e_i = (v_k, v_j)$ represents the control dependence between two nodes, v_k and v_j.

Usually, each node v_i in a CFG represents a basic block in a program. Wolf and Ernst extend the basic block concept to Single Feasible Path Program Segment (SFP-Prs) in [9]. A Program Segment can be viewed as a sequence of basic blocks with exactly on entry and one exit.

Definition 2. Single Feasible Path Program Segment (SFP-Prs): SFP-Prs is defined as a hierarchical program segment with exactly one path [9]. □

In this paper, each node in a CFG corresponds to a SFP-Prs. The SFP-Prs represented by the node v_j in the

CFG of task T_i is denoted by $SFP\_Prs(T_i, v_j)$.

We also need to clarify some definitions of caches and memory. A set-associative cache is defined by three parameters: the number of cache sets, the number of cache lines in a set (i.e., the number of ways) and the number of bytes/words in a cache line [23]. A direct mapped cache can be viewed as a special set associative cache which only has one way. The sets in a cache are indexed sequentially, starting from 0. All the cache lines in a cache set have the same index. A cache set with an index of i is represented with $cs(i)$. Accordingly, a memory address is divided into three parts: the tag, the index and the offset. We use $idx(a)$ to denote the index of a memory address a.

When a memory address is accessed, it is possible that only one byte or one word at this address is actually used by the program. However, when the byte/word at this address is loaded into the cache, the whole memory block that contains the byte/word requested is loaded into the cache instead of a single byte/word. A memory block has the same size as a cache line. Example 2 shows the relationship between cache and memory.

Example 2: Suppose we have a 4-way set associative cache with each line in the cache having 16 bytes. The size of the cache is 1KB. Thus, the maximum index of the cache is 15. If a memory address has 32 bits, we can derive each part (i.e., offset, index and tag) of the address for this cache as shown in Figure 2. When a memory address, $0x011$, is accessed and the byte at this address is not in the cache, the whole memory block that contains the byte at $0x011$ is loaded. The size of the memory block is also 16 bytes, starting from the address with an offset of 0. □

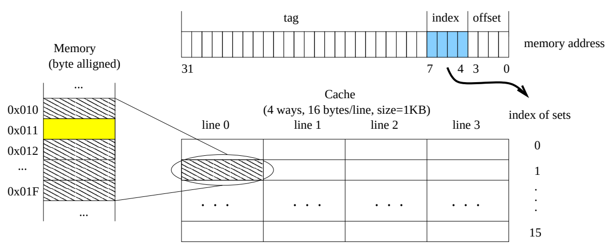

Fig. 2. Cache vs. Memory

In the rest of this paper, when we refer to a cache operation such as cache load and cache eviction, we always imply that the operation is performed on a unit of memory block by default. We do not distinguish the notation of “byte/word at a memory address” and “memory block” explicitly.

When a memory block with an address of a is loaded to a set associative cache, it can only occupy a cache line in the set with an index of $idx(a)$. In this paper, we assume that LRU algorithm is used for cache line replacement. However, our approach can also be applied to the caches with other replacement algorithms with minor modifications.

B. *Overall Approach*

Intuitively, we know that the cache lines causing reload overhead after preemptions need to satisfy two conditions.

Condition 1. These cache lines 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).

Condition 1 implies that memory blocks accessed by the preempting task conflict in the cache with memory blocks accessed by the preempted task. Thus, some of the memory blocks loaded to the cache by the preempted task before the preemption are evicted from the cache by the preempting task during the preemption. This cache eviction involves memory access patterns of both the preempted task and the preempting task. Thus, we call this type of cache eviction an inter-task cache eviction.

Condition 2 reveals that memory blocks causing cache reload overhead must have been present in the cache prior to the preemption. Furthermore, these memory blocks must be accessed again by the preempted task after the preemption, thus requiring reload to the cache. These memory blocks are called “useful memory blocks” in Lee’s work [21], [22]. We can use Lee’s algorithm in [21] to find the maximum set of these useful memory blocks. Lee’s algorithm does not consider the interaction between the preempting task and the preempted task. The maximum set of useful memory blocks of the preempted task is derived from the program structure of the preempted task and the memory blocks accessed by the preempted task. Thus, we call this type of analysis an intra-task cache eviction analysis.

Based on the two facts above, we can give an overview of our approach presented in this paper. Our approach has five steps.

First, we derive the memory trace of each task with the simulation method as used in SYMTA [9]. Here, we assume that there are no dynamic data allocations in tasks and addresses of all the data structures are fixed. Second, we perform intra-task cache access analysis on the preempted task to find the maximum set of useful memory blocks accessed by the preempted task. Only the memory blocks in this set can possibly cause cache reload delay. Third, we use the maximum set of useful memory blocks of the preempted task to perform inter-task cache eviction analysis with the preempting tasks (i.e., all the tasks that have higher priorities than the preempted task). A low priority task might be preempted more than once by a higher priority task, depending on the period of the low priority task as compared to the period of the high priority task. Fourth, we apply path analysis to the preempting task in order to tighten the estimate of the number of cache lines to be reloaded. After the fourth step, we can calculate the cache reload overhead. In the last step, we preform WCRT analysis for all tasks based on the results from the fourth step.

IV. Intra-task cache access analysis

According to Condition 2 in Section III-B, the memory blocks of the preempted task that can possibly cause cache reload overhead must be present in the cache before the preemption and must be accessed by the preempted task again after the preemption. Lee gives an approach to calculate the maximum set of such memory blocks.

As we mentioned in Section III-A, a task can be represented with a CFG. Each node in a CFG is an SFP-Prs. A task can be preempted at any point, which is called an execution point. When a preemption happens, a task can be viewed as two parts, one part before the preemption and the other part after preemption. The pre-preemption part of the preempted task loaded memory blocks to the cache. Some of these memory blocks might be accessed again by the post-preemption part of the preempted task. These memory blocks are called useful memory blocks. Only useful memory blocks of the preempted task can possibly cause cache reload after preemptions.

For a formal description, we use the notation of *reaching memory blocks (RMB)* and *living memory blocks (LMB)* as defined in [21]. The set of *reaching memory blocks* of a cache set $cs(i)$ at an execution point s of a task is denoted by RMB_s^i. RMB_s^i contains all possible memory blocks that may reside in cache set $cs(i)$ when the task reaches execution point s. Suppose a cache set has L cache lines (i.e., a L-way set associative cache). If a memory block can reside in $cs(i)$, this memory blocks must have an index of i. Moreover, in order to be contained in RMB_s^i, this memory block is one of the last L distinct references to the cache set $cs(i)$ when the task runs along some execution path reaching execution point s. Otherwise, this memory would have been evicted from the cache by other memory blocks. Similarly, the set of *living memory blocks* of cache set $cs(i)$ at execution point s, denoted by LMB_s^i, contains all possible memory blocks that may be one of the first L distinct references to cache set $cs(i)$ *after* execution point s.

In [21], Lee demonstrates that the intersection of RMB_s^i and LMB_s^i can be used to find a superset of the set of memory blocks in the preempted task that may cause cache line reload(s) due to preemption. These memory blocks are called “useful memory blocks”. The details of their algorithm can be found in [21], [22]. Of course, whether those memory blocks will really cause cache line reloading still depends on the actual path the preempted task takes and the cache lines used by the preempting task. In Lee’s approach, he conservatively assumes that all the useful memory blocks in the preempted task will be reloaded. Consider an extreme counter example for this assumption: if the cache lines used by the preempted task and the preempting task are completely disjoint, the preempting task will not evict any cache lines used by the preempted task. In this case, there is no cache reload overhead imposed on the preempted task, yet Lee’s approach would indicate significant reload overhead.

Therefore, in order to estimate the number of cache lines to be reloaded, we also need to find the cache lines used by the preempted task that may also be evicted by the preempting task during preemptions.

V. INTER-TASK CACHE EVICTION ANALYSIS

In [1], we propose an approach to calculate the intersection of cache lines that are used by both the preempted task and the preempting task. In that paper, we assume that all memory blocks used by the preempted task when the preempted task runs along the longest path are useful. However, the results from Lee’s approach shows that this is not always true. In this paper, we focus on incorporating Lee’s intra-task cache access analysis with the approach we presented in [1] in order to give a tighter estimate of cache-related delay caused by preemptions in multi-tasking preemptive systems.

Let us go back to the two conditions in Section III-B. Lee’s approach only considers Condition 2. His approach gives all memory blocks that can potentially cause cache reload in the preempted task. However, if these memory blocks need to be reloaded after preemption, they must be evicted from the cache by the preempting task. This implies that we need to calculate the intersection of cache lines used by the memory blocks found in Lee's approach and the memory blocks accessed by the preempting task. This is stated in Condition 1.

Memory blocks that are mapped to different cache sets will never conflict in the cache. In other words, only memory blocks that have the same index can possibly evict each other because these memory blocks are loaded to the same cache set. Intuitively, we can divide memory blocks into different subsets according to their index.

Suppose we have a set of q memory block addresses, $M = \{m_0, m_1, ..., m_{q-1}\}$, and an L-way set associative cache. The index of the cache ranges from 0 to $N-1$. We can derive N subsets of M as follows.

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

When the memory blocks in the same subset defined above 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).

If we denote $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$, where $\emptyset$ is the empty set and $\widehat{m}_i$ is defined as Equation 1, then $\widehat{M}$ is a partition of M. Based on this conclusion, we define the Cache Index Induced Partition (CIIP) of a memory block address set as follows.

Definition 3. Cache Index Induced Partition (CIIP) of a memory block address set: Suppose we have a set of memory block addresses, $M = \{m_0, m_1, ..., m_{q-1}\}$, and an L-way set associative cache. The index of the cache ranges from 0 to $N-1$. We can derive a partition of M based on the mapping from memory blocks to cache sets, which is denoted by $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$. Each $\widehat{m}_i = \{m_k \in M | idx(m_k) = i\}$ is a subset of M. We call $\widehat{M}$ the CIIP of M.□

The CIIP of a memory address set categorizes the memory block addresses according to their indices in the cache. Cache evictions can only happen among memory blocks that are in the same subset in the CIIP. We first defined and introduced CIIP in [1].

Example 3: Suppose we have a set of memory block addresses $M = \{0x000, 0x100, 0x010, 0x110, 0x210\}$. Also, we have a set associative cache as defined in Example 2. Therefore, $0x000$ and $0x100$ have the same index $0x0$. $0x010$, $0x110$ and $0x210$ have the same index $0x1$. So, the CIIP of this memory block address set is $\widehat{M} = \{\widehat{m}_0, \widehat{m}_1\}$, where $\widehat{m}_0 = \{0x000, 0x100\}$ and $\widehat{m}_1 = \{0x010, 0x110, 0x210\}$. Any block in $\widehat{m}_0$ will be loaded into the cache set with index 0 when the memory block is accessed. Any block in $\widehat{m}_1$ will be loaded into the cache set with index 1 when the memory block is accessed. Cache eviction can only happen among memory blocks in $\widehat{m}_0$ or memory blocks in $\widehat{m}_1$. A memory block in $\widehat{m}_0$ can never be replaced by a memory block in $\widehat{m}_1$ and vice versa because the memory blocks in $\widehat{m}_0$ and the memory blocks in $\widehat{m}_1$ are loaded into different sets in the cache. □

The definition of CIIP provides us a formal representation to analyze inter-task cache evictions. The memory block addresses in the same element of the CIIP have the same index. Therefore, when these memory blocks are loaded into the cache, they might conflict with each other. Memory blocks in different elements of the CIIP can never conflict in the cache.

Suppose we have two tasks T_a and T_b. All the memory blocks accessed by T_a and T_b are in the set $M_a = \{m_{a,0}, m_{a,1}, ..., m_{a,k_a}\}$ and $M_b = \{m_{b,0}, m_{b,1}, ..., m_{b,k_b}\}$ respectively. T_b has a higher priority than T_a. An L-way set associative cache with a maximum index of $N-1$ is used in the system. In the case T_a is preempted by T_b, the cache lines to be reloaded when T_a resumes are used by both the preempting task and the preempted task. Thus, we can look for the conflicting memory blocks accessed by the preempting task and the preempted task in order to estimate the number of reloaded cache lines. We can use the CIIPs of M_a and M_b to solve this problem.

We use $\widehat{M_a} = \{\widehat{m}_{a,0}, \widehat{m}_{a,1}, ..., \widehat{m}_{a,N-1}\}$ to represent the CIIP of 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. For $\widehat{m}_{a,k_1} \in \widehat{M_a}$ and $\widehat{m}_{b,k_2} \in \widehat{M_b}$, only when $k_1 = k_2$ can memory blocks in $\widehat{m}_{a,k_1}$ possibly conflict with memory blocks in $\widehat{m}_{b,k_2}$ in the cache. Also, when the memory blocks in $\widehat{m}_{a,k_1}$ and $\widehat{m}_{b,k_2}$ are loaded into the cache, the number of conflicts in the cache cannot exceed $min(|\widehat{m}_{a,k_1}|, |\widehat{m}_{b,k_2}|, L)$, where L is the number of ways of the cache. Therefore, we can conclude that the following formula gives an upper bound for the number of cache lines that could be reloaded after Task T_a resumes following a preemption by Task T_b:

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

where $\widehat{m}_{a,r} \in \widehat{M_a},\ \widehat{m}_{b,r} \in \widehat{M_b}$.

$S(M_a, M_b)$ denotes the upper bound on the number of cache lines that conflicts when the memory blocks in M_a and M_b are loaded into the cache. This number can be used to estimate the cache lines to be reloaded by T_b preempting T_a.

Example 4: Suppose we have a cache as defined in Example 2. Two tasks T_1 and T_2 run with this cache. The memory block addresses accessed by T_1 and T_2 are contained in $M_1 = \{0x000, 0x100, 0x010, 0x110, 0x210\}$ and $M_2 = \{0x200, 0x310, 0x410, 0x510\}$ respectively. The CIIPs of M_1 and M_2 are $\widehat{M_1} = \{\{0x000, 0x100\}, \{0x010, 0x110, 0x210\}\}$ and $\widehat{M_2} = \{\{0x200\}, \{0x310, 0x410, 0x510\}\}$ respectively.

If we map the memory blocks in M_1 and M_2 to the cache as shown in Figure 3(a), we find that the maximum number of overlapped cache lines, which is 4, is the same as the result derived from Equation 2. However, if we map the memory blocks in M_1 and M_2 to the cache as shown in the Figure 3(b), only two cache lines overlap. Obviously, the actual number of overlapped cache lines is related to the cache replacement policy and memory access pattern of the preempted task and the preempting task. However, we can guarantee that Equation 2 gives an upper bound of the number of overlapped cache lines. □

In Equation 2, we assume that M_a contains all memory blocks that can possibly be accessed by the preempted task, T_a. However, as we point out above, only useful memory blocks in M_a can possibly cause cache line reload no matter what memory blocks are accessed by the preempting task. Thus, we need to calculate the intersection of useful memory blocks of the preempted task as derived from Lee's approach and the memory blocks used by the preempting task in order to tighten the estimate of the number of cache lines to be reloaded as derived from Equation 2.

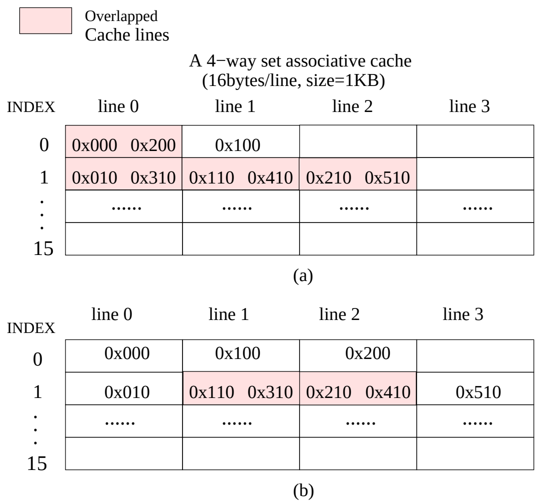

Fig. 3. Conflicts of cache lines in a set associative cache

Definition 4. The Maximum Useful Memory Blocks Set (MUMBS) The maximum intersection set of LMB and RMB over all the execution points of a task is called the maximum useful memory blocks set of this task. We represent the set of useful memory blocks of task T_a with $\tilde{M}_a$. $\widehat{\tilde{M}}_a$ is the CIIP of $\tilde{M}_a$. $\tilde{M}_a$ is a subset of M_a. □

We use Lee's approach to calculate the maximum useful memory blocks set of the preempted task. Only the memory blocks in this set can possibly need to be reloaded by the preempted task. The maximum set useful memory blocks of the preempted task only depends on the structure and the memory accessed by the preempted task.

The simulation method in SYMTA is used to obtain all the memory blocks that can possibly accessed by the preempting task [9]. All these memory blocks are contained in a set M_b. $\widehat{M_b}$ is the CIIP of M_b. Only the memory blocks in M_b can possibly evict the cache lines used by the preempted task.

Then, we apply Equation 2 to calculate the intersection of memory block set $\tilde{M}_a$ and M_b, which is shown in Equation 3. This result also gives an upper bound of the number of cache lines that can possibly need to be reloaded after T_b preempts T_a. Since $\tilde{M}_a$ is a subset of M_a, the estimate given in Equation 3 can be less than the estimate in Equation 2. Hence, we can expect a tighten WCRT estimate.

$$S(\tilde{M}_a, M_b) = \sum_{r=0}^{N-1} min\{|\widehat{\tilde{m}}_{a,r}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{3}$$

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

VI. Path analysis for the preempting task

The set M_b used in the section above contains all the memory block addresses that can possibly be accessed by the preempting task T_b, if we do not use any path analysis methods. In this case, the result derived from Equation 3 only gives an upper bound of the number of cache lines that could be potentially reloaded by the preempted task.

However, since the preempting task might have more than one feasible path and only one path is executed, some memory blocks may not be accessed, thus, there is no need to reload the cache lines mapped from those memory blocks. Example 5 gives such a case.

Example 5: Figure 4 shows the CFG of ED which has four SFP-Prs. When the image size is fixed (i.e. the number of pixels to be processed is fixed), the loop bounds in the dashed-line rectangles are fixed. There are no other branches depending on the input data in these two loops. Thus, these two loops can be viewed as SFP-Prs. The CFG of ED can be simplified as the graph shown in Figure 4 (b). Each node in this graph represents an SFP-Prs in the ED program. According to the parameter selected by the user, the program can only take either the path $(v_1, e_1, v_2, e_2, v_3, e_4, v_5)$ or the path $(v_1, e_1, v_2, e_3, v_4, e_5, v_6)$; thus, only one of two SFP-Prs, v_3 or v_4, can be accessed in one run. In this case, the evicted cache lines to be used by v_3 and the evicted cache lines to be used by v_4 do not need to be reloaded at the same time in one run. □

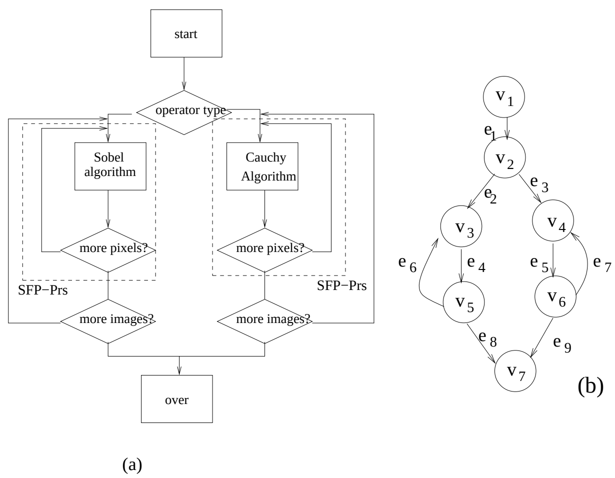

Fig. 4. CFG of ED

The issue presented in Example 5 can be described more generally. Suppose we have two tasks in a system with an L-way set associative cache, T_a and T_b. The largest index of the cache is $N-1$. T_b has a higher priority than T_a. Thus, T_b can preempt T_a. We use M_a to represent the set of all memory block addresses that can be possibly accessed by T_a. $\tilde{M}_a$ is the maximum set of useful memory blocks of the preempted task, as given in Section IV. The CFG of T_b is $G_b = (V_b, E_b)$, where $V_b = \{v_{b,1}, v_{b,2}, ..., v_{b,n}\}$ and $E_b = \{e_{b,1}, e_{b,2}, ..., e_{b,m}\}$. A path in G_b can be represented with $Pa_b^k = \{v_{b,i_1}, e_{b,i_1}, v_{b,i_2}, e_{b,i_2}, ..., v_{b,i_p}\}$. We use M_b^k to denote the set of memory block addresses accessed by the task T_b when T_b runs along the path Pa_b^k. The CIIP of M_b^k is $\widehat{M}_b^k = \{\widehat{m}_{b,0}^k, \widehat{m}_{b,1}^k, ..., \widehat{m}_{b,N-1}^k\}$. When Pa_b^k is determined, $M_{b,k}$ can be derived from simulation as stated in Section III-A. Now, we need to find a path in the preempting task T_b. When T_b takes this path, the memory blocks loaded to the cache have the largest overlap with the cache lines used by memory blocks in the maximum useful memory blocks set of the preempted task T_a. In another words, when T_b takes this path, the number of cache lines evicted by T_b and also used by T_a is the largest. This problem can be transformed to a problem of finding the longest path in a graph.

We define a cost function for the path Pa_b^k in the preempting task T_b.

$$C(Pa_b^k) = S(\tilde{M}_a, M_b^k) = \sum_{r=0}^{N-1} min\{|\widehat{\tilde{m}}_{a,r}^{k}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{4}$$

The cost of a path Pa_b^k in the preempting task T_b is defined as the maximum number of cache lines that can be possibly overlapped with the cache lines mapped by useful memory blocks of the preempted task T_a, when the preempting task T_b runs along the path Pa_b^k.

By using this cost function, we search all the paths of the preempting to find the longest path in the CFG of T_i. Suppose the longest path is represented with $Pa_{longest}$, the cache lines to be reloaded in the worst case is bounded by the cost of $Pa_{longest}$. This algorithm potentially needs to calculate over all paths. However, in practice, many embedded programs have control flow graphs with a reasonably small number of paths. Thus, our approach can still apply to many such systems.

Compared to Equation 3, the estimate given in Equation 4 is reduced further, because only a part of memory blocks in M_b are considered in the calculation of intersection by using Equation 4.

We use $C_{pre}(T_a, T_b)$ to represent the cache reload cost imposed on task T_a when T_a is preempted by task T_b. Suppose the penalty for a cache miss is a constant, C_{miss}, $C_{pre}(T_a, T_b)$ can be calculated with the following equation:

$$C_{pre}(T_a, T_b) = C(Pa_{longest}) \times C_{miss} \tag{5}$$

This equation gives an estimate of the cache eviction cost induced by T_b preempting T_a. By incorporating the cache eviction cost, we can derive a new approach to estimate the WCRT of each task in a preemptive multi-tasking system.

VII. WCRT ANALYSIS

We can use the Worst Case Response Time (WCRT) to analyze schedulability of a multi-tasking real-time analysis as shown in [19]. The approach uses the following recursive equations to calculate the WCRT R_i of the task T_i.

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times C_j \tag{6}$$

where $hp(i)$ is the set of tasks whose priorities are higher than T_i. Recall thatC_j is the WCET of T_j and P_j is the period of Task T_j as defined in Section III-A. In this equation, the term $\sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times C_j$ reflects the interference of preempting tasks during the execution time of T_i. This equation can be calculated iteratively. The iteration can be terminated when R_i converges or R_i is greater than the deadline of T_i. If R_i is greater than its deadline, task T_i cannot be scheduled successfully.

Note that C_j is the WCET estimate of T_j without considering preemption. We use SYMTA [9] to estimate WCET. However, the costs of cache reload and context switch caused by preemptions are not included in Equation 6.

Therefore, Equation 6 may underestimate the WCRT of a task. Here, we focus on cache reload overhead analysis and assume the cost of a context switch is a constant, C_{cs}, which is equal to the WCET of a context switch. Example 6 gives the context switch cost for our simulation architecture. The context switch function cannot be preempted, so the context switch cost is not affected by inter-task cache eviction. Therefore, it is reasonable to assume the context switch cost is a constant, which is its WCET. The context switch function is called twice in every preemption, once for switching to the preempting task and once for resuming the preempted task.

Example 6: An ARM9TDMI processor with two levels of memory, a 32KB 4-way set associative L1 cache and 256MB SRAM, is used in our experiment. The cache miss penalty is 20 cycles. The Atalanta RTOS developed at Georgia Tech [24] is used for task management. We use SYMTA to obtain the WCET of a context switch, which implies that the instructions of the context switch function and the memory blocks where contexts of the preempted and the preempting tasks are saved are not in the L1 cache when the context switch function is called. In this case, the WCET of a single context switch estimated with SYMTA is 1049 cycles. □

When preemptions are allowed in a multi-tasking system, the WCRT of tasks that can be preempted may be increased because of cache reload overhead. We use $C_{pre}(T_i, T_j)$ to represent the cache reload overhead imposed on task T_i when T_i is preempted by task T_j. $C_{pre}(T_i, T_j)$ is defined in Equation 5. By considering the cache reload overhead, Equation 6 can be modified as follows to no longer underestimate R_i:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times (C_j + C_{pre}(T_i, T_j) + 2C_{cs}) \quad (7)$$

Based on Equation 7,we can estimate the WCRT for each task T_i with the following iteration:

$R_i^0 = C_i$;

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

...

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

This iteration terminates when R_i converges or R_i is greater than the deadline of T_i. After the iteration is terminated, we compare the value of R_i with the deadline of T_i. If R_i is less than the deadline of T_i, T_i can be scheduled. Otherwise, T_i cannot be scheduled. Hence, we can analyze the schedulability of the system based on the WCRT estimate of each task. We need to perform such iteration for each task except the task with the highest priority. Thus, the computational complexity of WCRT estimate with the above equation is directly proportional to the number of tasks.

VIII. Experimental Results

Two groups of applications are used in experiments. The applications are run on an ARM9TDMI processor with a 4-way set associative cache, the size of which is 32KB. Each line in the cache is 16 bytes, thus, there are 512 lines in each "way" of the cache in total. The instruction set is simulated with XRAY [26]. The tasks are supported by Atalanta RTOS developed at Georgia Tech [24]. The whole system is integrated with Seamless CVE provided by Mentor Graphics [25]. The simulation environment is shown in Figure 5.

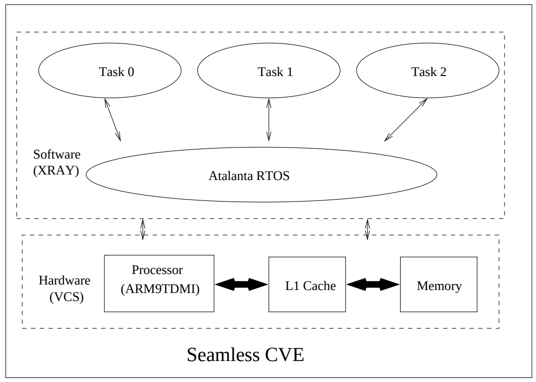

Fig. 5. Simulation Architecture

The first set of tasks, OFDM, ED and MR are described in Example 1. The second set of tasks are Adaptive Differential Pulse Code Modulation Coder (ADPCMC), ADPCM Decoder (ADPCMD) and Inverse Discrete Cosine Transform (IDCT). ADPCMC and ADPCMD are taken from MediaBench [27], [28]. IDCT is extracted from MPEG2 decoder. We use SYMTA, which is a single-task based WCET estimate approach as mentioned in Section III-A, to estimate the WCET of each task in the experiment. The periods, priorities and WCET of tasks in each experiment are listed in Table I.

TABLE I

TASKS

Tasks in Experiment I			
Task	WCET(us)	Period(us)	Priority
T_1(OFDM)	2830	40,000	4
T_2(ED)	1392	6,500	3
T_3(MR)	830	3,500	2
Tasks in Experiment II			
Task	WCET(us)	Period(us)	Priority
T_1(ADPCMC)	7675	50,000	4
T_2(ADPCMD)	2839	10,000	3
T_3(IDCT)	1580	4,500	2

In the experiment, we compare four approaches to estimate cache reload overhead caused by preemptions.

Approach 1: All cache lines used by preempting tasks are reloaded for a preemption. Note that this approach is proposed by [20].

Approach 2: Only lines in the intersection set of lines used by the preempting task and the preempted task are reloaded after a preemption. Inter-task cache eviction method proposed in [1] is used here.

Approach 3: Only useful memory blocks in the preempted task are used to estimate the cache reload delay. Intra-task cache access analysis for the preempted task proposed by Lee in [21] is used here.

Approach 4: Both inter-task cache eviction analysis and intra-task cache access analysis are used to estimate the cache reload cost. Also, path analysis proposed in Section VI is applied to the preempting task. This is the approach described in this paper.

The estimates of the number of cache lines to be reloaded in each type of preemption derived with these fours approaches are listed in Table II.

TABLE II

NUMBER OF CACHE LINES TO BE RELOADED

Experiment I				
Preemptions	App. 1	App. 2	App. 3	App. 4
OFDM by MR	245	134	187	88
OFDM by ED	254	172	187	98
ED by MR	245	87	106	81
Experiment II				
Preemptions	App. 1	App. 2	App. 3	App. 4
ADPCMC by IDCT	249	68	98	56
ADPCMC by ADPCMD	220	114	98	64
ADPCMD by IDCT	183	58	89	46

Approach 1 assumes that all cache lines used by the preempting task will be accessed by the preempted task after the preempted task is resumed. Obviously, this may not be true. Some cache lines will never be used by the preempted task no matter which path the preempted task takes. Thus, by calculating the set of cache lines that can possibly be accessed by both the preempting and the preempted task, we can further reduce the estimate of the number of cache lines to be reloaded by the preempted task, as shown in Approach 2.

Approach 3 calculates the maximum set of memory blocks in the preempted task that can potentially cause cache reload. This approach only relates to the structure and memory access pattern of the preempted task. Thus, for a certain preempted task, the estimate of cache reload overhead is always the same. Obviously, this approach ignores the differences among preempting tasks and only assumes that all “useful” memory blocks in the preempted task will be evicted by the preempting task which might not be true. By considering the preempting tasks and incorporating inter-task cache eviction analysis, the estimate of the number of cache lines that need to be reloaded is significantly reduced, as shown in Table II.

The WCRT of OFDM and ED can be calculated based on the results shown in Table II. Notice that MR has the hight priority so that it can never be preempted. So, the WCRT of MR is just equal to its WCET. We also vary the C_{miss} from 10 cycles to 40 cycles to investigate the influence of cache miss penalty on the WCRT. The estimate results and the Actual Response Times (ART) are listed in Table III. Table IV lists the improvement of our approach (Approach 4) over all other approaches (Approach 1, Approach 2 and Approach 3). The same results of the second experiment are listed in Table V and Table VI.

TABLE III

COMPARISON OF WCRT ESTIMATE (EXPERIMENT I)

C_{miss}	Task	App. 1	App. 2	App. 3	App. 4	ART
10	OFDM	9847	9350	9539	6456	6113
	ED	2567	2409	2428	2403	2382
20	OFDM	12510	10096	10474	9524	6211
	ED	2812	2496	2534	2484	2400
30	OFDM	23501	12174	12900	9984	6255
	ED	3057	2583	2640	2565	2426
40	OFDM	45216	16700	23536	10444	6362
	ED	3302	2670	23746	2646	2525

TABLE IV

COMPARISON OF RESULTS IN EXPERIMENT I

		Cache Penalty (cycles)			
	Task	10	20	30	40
App.4 vs. App.1	OFDM	34%	24%	58%	77%
	ED	6%	12%	16%	20%
App.4 vs. App.2	OFDM	31%	6%	18%	38%
	ED	0.2%	0.5%	1%	1%
App.4 vs. App.3	OFDM	32%	9%	23%	56%
	ED	1%	2%	3%	4%

Compared with Approach 2 and Approach 3, our approach (App. 4) achieves a reduction of from 38% to 56% in WCRT estimate of OFDM when the cache penalty is 40 cycles. Thus, combining inter-task cache evition analysis with intra-task cache access analysis can significatly tighten the estimate of cache reload cost caused by preemptions in multi-tasking systems, which in turn allow us to obtain a more precise estimate of WCRT.

IX. CONCLUSION

We propose a WCRT analysis approach in this paper. The cache reload overhead caused by preemptions are considered in our approach. Inter-task cache eviction analysis is combined with useful memory block analysis of

TABLE V

COMPARISON OF WCRT ESTIMATE (EXPERIMENT II)

C_{miss}	Task	App. 1	App. 2	App. 3	App. 4	ART
10	ADPCMC	35743	29070	29232	28836	23512
	ADPCMD	6565	6315	6377	6291	6190
20	ADPCMC	48528	29888	35223	29420	23867
	ADPCMD	6931	6431	6555	6383	6223
30	ADPCMC	88606	35871	38373	34983	24101
	ADPCMD	7297	6547	6733	6475	6278
40	ADPCMC	359239	38823	39647	30588	24353
	ADPCMD	7663	6663	6911	6567	6354

TABLE VI

COMPARISON OF RESULTS IN EXPERIMENT II

		Cache Penalty (cycles)			
	Task	10	20	30	40
App.4 vs. App.1	ADPCMC	19%	39%	60%	92%
	ADPCMD	4%	8%	11%	14%
App.4 vs. App.2	ADPCMC	1%	2%	3%	21%
	ADPCMD	1%	1%	1%	1%
App.4 vs. App.3	ADPCMC	2%	17%	9%	23%
	ADPCMD	1%	3%	4%	5%

the preempted task. The experiment shows that our approach can reduce the estimate of WCRT by 38% to 77%, compared with prior to approachs.

For future work, we plan to expand our analysis approach for systems with more than two-level memory hierarchy. Also, we will research on the cache eviction problem in multi-processor systems.

REFERENCES

[1] Y. Tan and V. Mooney, "Timing Analysis for Preemptive Multi-tasking Real-Time Systems," *Proceedings of Design, Automation and Test in Europe (DATE'04)*, February 2004.

[2] D. Kirk, "SMART (Strategic Memory Allocation for Real-Time) Cache Design", *Proceedings of IEEE 10th Real-Time System Symposium*, pp. 229-237, December 1989.

[3] G. Suh, L. Rudolph and S. Devadas, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems," *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 116-127, September 2001.

[4] J. Liedtke, H.Härtig and M. Hohmuth, "OS-Controlled Cache Predictability for Real-Time Systems," *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pp. 213-227, June 1997.

[5] F. Muller, "Compiler Support for Software-based Cache Partitioning," *Proceedings of ACM SIGPLAN Workshop on Languages, Compliers and Tools for Real-Time Systems*, pp. 125-133, June 1995.

[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*, Vol. 4, No. 3, pp. 257-279, July 1999.

[7] Y. Li, S. Malik and A.Wolfe, "Efficient Microarchitecture Modeling and Path Analysis for Real-time Software," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 298-397, December 1995.

[8] Y. Li and S. Malik, *Performance Analysis of Real-Time Embedded Software*, Kluwer Academic Publishers, Boston, 1999.

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

[10] F. Wolf, Jan Staschulat and Rolf Ernst, "Hybrid Cache Analysis in Running Time Verification of Embedded Software," *Design Automation for Embedded Systems*, Vol. 7, No. 3, pp. 271-295, October 2002.

[11] F. Wolf, J. Staschulat and R. Ernst, "Associative Caches in Formal Software Timing Analysis," *Proceedings of the IEEE/ACM Design Automation Conference*, June 2002.

[12] F. Wolf, R. Ernst, and W. Ye, "Path Clustering in Software Timing Analysis," *IEEE Transactions on VLSI Systems*, Vol.9, No.6, December 2001.

[13] A. Ermerahl, F. Stappert and J. Engblom, "Clustered Calculation of Worst-Case Execution Times," *Proceedings of Compilers, Architecture and Synthesis for Embedded Systems*, pp. 51-62, October 2003.

[14] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing and R. Wilhelm, "Reliable and Precise WCET Determination for a Real-Life Processor," *Proceedings of the First International Workshop on Embedded Software, (EMSOFT 2001)*, pp. 469-485, Volume 2211 of LNCS, Springer-Verlag (2001).

[15] M. Alt, C. Ferdinand, F. Martin and R. Wilhelm, "Cache behavior prediction by abstract interpretation," *Proceedings of Static Analysis Symposium (SAS'96)*, pp. 52-66, September 1996.

[16] T. Lundqvist and P. Stenstrom, "An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution,"*Real-Time Systems*,Volume 17, Issue 2-3, pp. 183-207, November 1999.

[17] J. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm:Exact Characterization and Average Case Behavior," *Proc. IEEE 10th Real-Time System Symposium*, pp. 166-171, 1989.

[18] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of ACM*, Vol. 20, No. 1, pp. 26-61, January 1973.

[19] K. Tindell, A. Burns, A. Wellings, "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks," *Real-Time Systems* Vol.6, No.2, pp. 133-151, March 1994.

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

[21] 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 Fixed-priority Preemptive Scheduling," *IEEE Transactions on Computers*, Vol. 47, No. 6, pp. 700-713, 1998.

[22] C. Lee, J. Hahn, Y.-M. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee and C. Kim, "Enhanced Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, pp. 187-198, December 1997.

[23] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (3rd edition), Morgan Kaufmann Publishers, Menlo Park, CA, 2002.

[24] D. Sun, D. Blough and V. Mooney, "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications," Technical Report GIT-CC-02-19, Georgia Institute of Technology, April 2002.

[25] Mentor Graphics, Seamless Hardware/Software Co-Verification, http://www.mentor.com/seamless/.

[26] Mentor Graphics XRAY Debugger, http://www.mentor.com/embedded/xray/.

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

[28] C. Lee, M. Potkonjak and W. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems,” *Proceedings of International Symposium on Microarchitecture*, pp. 330-335, 1997.