Technical Report GIT-CC-04-11

Cache-Related Timing Analysis for Multi-tasking Real-Time Systems with Nested Preemptions

Yudong Tan and Vincent J. Mooney III
Center of Research for Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA

November 8, 2004

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 utilizing 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 task(s) 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 up to 32% (1.4X) 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, many 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 underlying scheduling algorithms.

In this paper, we give an approach to analyze the Worst Case Response Time (WCRT) of each task in a multi-tasking system. We target a single-processor preemptive multi-tasking system with L1 set associative caches. The approach focuses on the cache reload cost 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 by up to 32% over prior state-of-the art.

The rest of this paper is organized as follows. Section II introduces 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 category limits 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 Translation Look-aside Buffers (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 implicit path enumeration method [7]–[9]. 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 [10]–[13]. The precision of time estimation is improved in SYMTA since the overestimate of execution time is reduced. Ermerahl et al. [14] give a clustered calculation approach to reduce the timing overestimate. This approach is similar to SYMTA in removing overestimate of execution time between boundaries of basic blocks. Wilhelm et al. [15]–[17] propose an abstract interpretation methodology to predict cache behavior. Stenstrom et al. [18] give 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. White et al. give a timing analysis approach in [6] for data and wrap-around-fill cache. However, 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 the Rate Monotonic Algorithm (RMA) is used in the system [19], [20]. 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 Worst Case Response Time (WCRT) [21] to analyze schedulability. Busquests-Mataix et al. propose an approach to analyze cache eviction cost in a multi-tasking system [22]. 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. Tomiyama et al. give an approach to calculate Cache Related Preemption Delay (CRPD) by using ILP [23]. However, they only consider direct mapped instruction cache. Lee et al. also give an approach for cache analysis in preemptions [24]. 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. In the approach of Lee et al., the cache reload cost is still the same as the cost to reload all "useful" memory blocks in the preempted task. Lee et al. enhance their approach in [25]. In [25], all preemption scenarios are explored to find the cache reload

cost. The number of preemption scenarios increases exponentially with the number of tasks. Thus, calculating the cache reload cost for each preemption scenario separately is not efficient to compute. Moreover, although Lee et al. mention that the cache reload cost is calculated based on the intersection of cache lines used by the preempting task and the preempted task, no method is given to show how the intersection is calculated. Also, the structure of the program is not considered in their approach [25]. As shown in [1], overestimation may occur if we do not analyze execution paths in the preempting task. Negi et al. [27] refine the approach of Lee et al. in [24] by applying path analysis. However, inter-task cache eviction is not considered. Also, WCRT analysis is not mentioned in [27].

In [1], we propose an approach for inter-task cache eviction analysis. A Cache Index Induced Partition (CIIP) estimates the number of cache lines evicted during a preemption. By using CIIP, we provide a formal method to calculate the intersection of cache lines used by two tasks. This method can be uniformly applied to direct mapped caches and set associative caches. Also, path analysis is applied to the preempted task in order to tighten the estimate. This work is published 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 [24] and [25], only those cache lines used by "useful" memory blocks of the preempted task need to be reloaded.

In this paper, we focus on enhancing our approach in [1] by incorporating "useful" memory block analysis of Lee et al. [24], [25]. In Section VIII we will show examples where we achieve results up to 30% better than the approach of Lee et al. in [24]. We also compare our approach with the enhanced approach of Lee et al. in [25]. The approach in [25] consists of two parts, the estimate of cache reload cost for each preemption and the number of preemptions in the worst case. These two parts are orthogonal. Our approach in this paper focuses on the cache reload cost estimate. Thus, we only compare the cache reload cost estimate derived from the enhanced approach of Lee et al. and our approach. The experimental results show that, as a result of applying path analysis, our approach can achieve up to 32% reduction in WCRT estimation as compared to the enhanced approach of Lee et al. in [25].

III. Overview

In this section, we first state the problem formally. 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.

Suppose that the system contains n tasks represented with $T_0, T_1, ..., T_{n-1}$. A Fixed Priority Scheduling (FPS) algorithm such as the Rate Monotonic Algorithm (RMA) is used in the system. Each task T_i has a fixed priority p_i. If $p_a < p_b$, T_a has a higher priority than T_b. We assume that the tasks are sorted in the descending order of their priorities so that we have $p_0 < p_1 < ... < p_{n-1}$. Tasks are executed periodically. Each task T_i has a fixed period P_i. T_i arrives at the beginning of its period and must be completed by the end of its period. The Worst Case Execution Time (WCET) of task T_i is denoted with C_i. C_i can be estimated with existing analysis tools such as Cinderella [9] and SYMTA [10]. We use SYMTA to derive C_i. We use $T_{i,j}$ to represent the j^{th} run of Task T_i.

The WCET of a task is the execution time of this task in the worst case, assuming there are no preemptions or interruptions. In a preemptive multi-tasking system, WCET alone cannot reflect the schedulability of tasks in the system because of the existence of preemptions. Thus, our goal is to provide an approach to estimate the Worst Case Response Time (WCRT), which is defined as below, for every task in the system.

In a multi-tasking system space, we aim at estimating the Worst Case Response Time (WCRT) of tasks, as defined in [21], 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 cost caused by inter-task cache evictions increases the response time of the preempted task.

Example 1: We have three tasks T_0, T_1 and T_2. T_0 is an Inverse Discrete Cosine Transform (IDCT) extracted from an MPEG2 decoder. T_0 is invoked every 4.5ms. T_1 is an Adaptive Differential Pulse Code Modulation Decoder (ADPCMD). T_2 is an ADPCM Coder (ADPCMC). ADPCMC and ADPCMD are taken from MediaBench [32], [33]. ADPCMC has a period of 50ms. ADPCMD has a period of 10ms. RMS is used for scheduling. T_0 has the highest priority and T_2 has the lowest priority. Figure 1 shows this example. In this example, three tasks arrive at time instant 0. However, T_2 is not executed until there are no instances of T_0 or T_1 ready to run. During the execution of T_2, it could be preempted by T_0 or T_1, which is shown in Figure 1. The response time of T_2 is the time from 0 to the time when T_2 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_2 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 imposes an overhead on the WCRT of the preempted task. Figure 1(B) shows this issue. Obviously, due to cache evictions, the WCRT of T_2 is increased, as shown in Figure 1(B). Note that the cache reload cost occurs only when some memory blocks evicted from the cache during the preemption are required again by the preempted task, which does not necessarily happen for every preemption. □

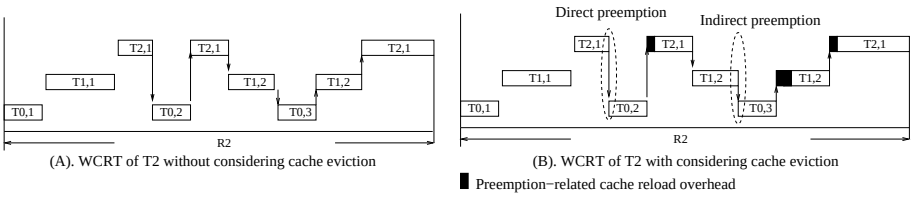


Fig. 1. Example of WCRT

As shown in Example 1, inter-task cache eviction affects the WCRT of a task. 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 we propose in [1] with the approach of Lee et al. in [24]. The approach proposed in this paper can be applied to both instruction caches and data caches, all that is required is a stream of addresses accessed. In our experiment, we use a unified cache without differentiating the instruction cache from the data cache.

In this paper, we perform path analysis on both 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 a 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 [10]. 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 [10]. □

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 [28]. A direct mapped cache can be viewed as a special set associative cache which has only 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. □

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

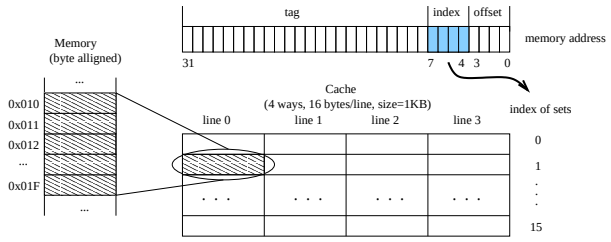


Fig. 2. Cache vs. Memory

When a memory block with an address of a is loaded into 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 an LRU algorithm is used for cache line replacement. However, our approach can also be applied to caches with other replacement algorithms with minor modifications. For example, if a Round-Robin algorithm is used for cache line replacement, we only need to slightly change the intra-task cache eviction algorithm in our approach. The inter-task cache eviction analysis algorithm can be applied to all cache line replacement policies.

B. Overall Approach

Intuitively, we know that the cache lines causing reload overhead after preemption(s) 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. Note that Condition 1 is only a necessary condition for cache eviction. If the preempted task and the preempting task access the same memory block, a cache hit may occur. In this case, there is no cache reload cost for the cache line mapped by that memory block. However, Condition 1 prevents underestimating the cache reload cost. Please note that we extend Condition 1 in a straightforward way to nested preemptions.

Condition 2 reveals that memory blocks causing cache reload cost 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 the work of Lee et al. [24], [25]. We can use the algorithm of Lee et al. in [24] to find the maximum set of these useful memory blocks. The algorithm of Lee et al. 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 [10]. Here, we assume that there are no dynamic data allocations in tasks and addresses of all the data structures are fixed. Also, we assume that we can determine all loop bounds (or the user gives us an upper bound for each loop) so that the memory trace in the worst case can be derived. 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 cost. In the last step, we preform WCRT analysis for all tasks (including nested preemptions) 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 cost must be present in the cache before the preemption and must be accessed by the preempted task again after the preemption. Lee et al. give 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 preemption(s).

For a formal description, we use the notation of *reaching memory blocks (RMB)* and *living memory blocks (LMB)* as defined in [24]. 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 [24], Lee et al. demonstrate 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 [24]. 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 the approach of Lee et al., 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 (e.g., all cache lines used have distinct indices), the preempting task will not evict any cache lines used by the preempted task. In this case, there is no cache reload cost imposed on the preempted task, yet the approach of Lee et al. would indicate significant reload overhead (e.g., the approach of Lee et al. does not distinguish cache lines based on indices).

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 the approach of Lee et al. 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. The approach of Lee et al. 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 have been 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 the approach of Lee et al. 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 $\widehat{m}_i$ 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).

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 $\widehat{m}_i$ in $\widehat{M}$, the CIIP of memory address set M. 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 0. $0x010$, $0x110$ and $0x210$ have the same index 1. 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 subset $\widehat{m}_i$ 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 subsets $\widehat{m}_i$, $\widehat{m_j}$, $i \neq j$, 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 conflicted cache lines in one set of the cache can neither exceed the number of memory blocks that mapped to this set nor exceed the total number of cache lines in this set. In other words, the maximum number of cache lines conflicted in the set with index k_1 (or k_2 because $k_1 = k_2$) in the cache is $min(|\widehat{m}_{a,k_1}|, |\widehat{m}_{b,k_2}|, L)$, 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 an upper bound on the number of cache lines that may conflict 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 due to 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. Note that the memory blocks can be mapped to cache lines in other ways (e.g., $0x100$ can possibly be mapped to line 0 instead of line 1, but in this case $0x100$ would kick out $0x200$ or vice versa). In any case, the mapping given in Figure 3(a) gives a case in which the largest amount of cache line overlaps occurs. Let us consider another case. 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, 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

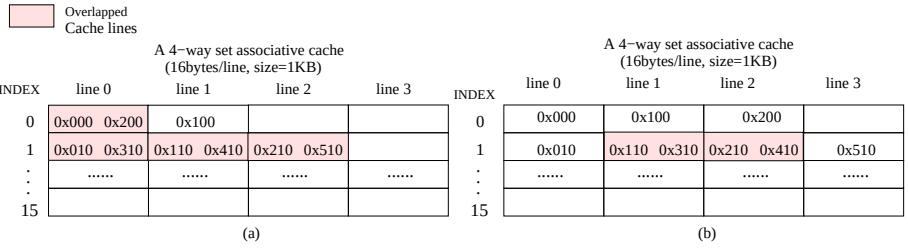


Fig. 3. Conflicts of cache lines in a set associative cache

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 the approach of Lee et al.) 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.

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 the approach of Lee et al. 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 Useful Memory Blocks Set (MUMBS) of the preempted task only depends on the program structure of the preempted task 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 [10]. 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 tighter 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 multiple feasible paths only one of which is executed, some memory blocks included in Equation 5 calculation may in fact not be accessed; thus, there is no need to reload the cache lines mapped from these memory blocks not accessed. 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. □

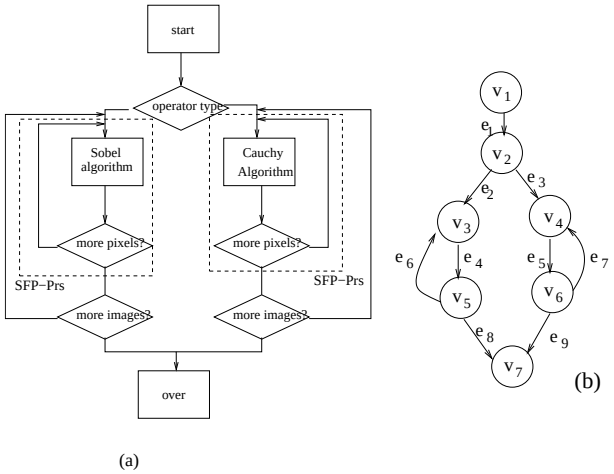


Fig. 4. CFG of ED

The issue presented in Example 5 can be described more generally. Suppose we have two tasks, T_a and T_b, in a system with an L-way set associative cache. The largest index of any cache line in 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 with the method used in SYMTA [10] as explained in Section III-A.

Note that Pa_b^k is generic notation for any path in T_b. Among all the paths in T_b, there exists a particular path in T_b which, 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 MUMBS 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 describe this transformation in the following paragraphs.

We define a cost function for a 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_b. Suppose the longest path in task T_b is represented with $Pa_b^{longest}$, the cache lines to be reloaded in the worst case is bounded by the cost of $Pa_b^{longest}$. 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:

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

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.

Note that the cache miss penalty in practice may not be always constant. For example, in a wrap-around-fill cache [28], a cache line is not filled completely at one time. Instead, CPU starts to run as soon as the requested memory contents are fetched from to the cache. The rest of the cache line is filled while the CPU continues execution. In this type of cache, the cache miss penalty varies. An cache miss delay analysis approach is presented in [6] for the wrap-around-fill cache. Our approach can be easily extended to handle the wrap-around-fill cache by replacing the constant C_{miss} with a cache miss penalty function $C_{miss}(m_i)$, where m_i is in the intersection set of the memory blocks used by the preempting task and the preempted task, as found by the approach above.

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 [22]. The approach uses the following recursive equations to calculate the WCRT R_i of the task T_i.

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

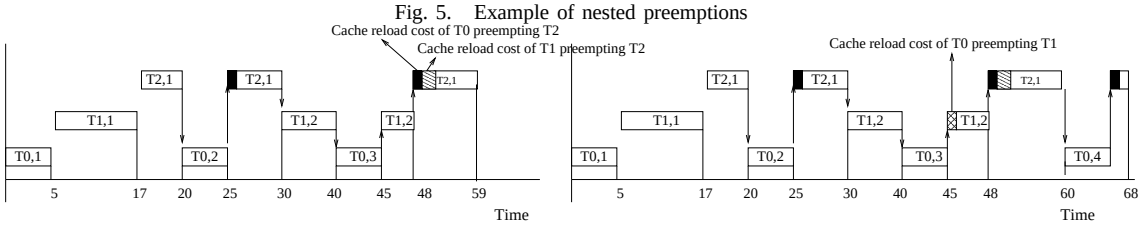
where $hp(i)$ is the set of tasks whose priorities are higher than T_i. Because we assume that all tasks are sorted in the descending order of their priorities in this paper, we have $hp(i) = \{k | 0 \leq k < i\}$. γ_j is the cache reload cost related to preemptions caused by T_j (indirect or direct). Recall that C_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 $\sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times (C_j + \gamma_j)$ 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, then we are not able to schedule task T_i successfully.

TABLE I

TASKS IN EXAMPLE 6

Task	WCET(us)	Period(us)	Preemptions	Cache reload cost (us)
T_0	5	20	T_0 preempting T_1	1
T_1	12	30	T_0 preempting T_2	2
T_2	15	100	T_1 preempting T_2	2

Fig. 5. Example of nested preemptions



(A). WCRT of T2 without considering nested preemptions

(B). WCRT of T2 with considering nested preemptions

In [22], the authors assume that all the cache lines used by the preempting task need to be reloaded after the preemption. With this assumption, there is no need to distinguish indirect preemptions from direct preemptions because the estimate of cache reload cost is only related to the preempting task in this case. However, as we pointed out in Section IV and V, this assumption exaggerates the cache reload cost for each preemption. We can apply inter-task and intra-task cache eviction analysis techniques above to reduce the overestimation in Equation 6. Let us consider Example 6.

Example 6: Consider the tasks in Example 1. We assume that WCET, period and cache reload cost for each task are listed in Table I. Here we ignore the context switch cost. If we do not consider the indirect preemptions, the tasks are scheduled as shown in Figure 5(A). However, the third run of T_0 preempts T_2 indirectly. Thus, we have to include the cache reload cost of T_0 preempting T_1 in this case, which make tasks scheduled as shown in Figure 5(B). Notice that when the nested preemptions are considered, the WCRT of T_2 is 68. □

Example 6 shows the effect of nested preemptions on WCRT. Thus, 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 $C_{pre}(T_a, T_b)$ to the WCRT of T_a. T_a can also be preempted 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 $C_{pre}(T_l, T_b)$ to the WCRT of T_a. In the worst case, T_{a-1} may preempt T_a first, then T_{a-1} is preempted T_{a-2}, ..., T_{b+1} is then preempted by T_b. Thus, there might be $a - b$ nested preemptions in the worst case. In Equation 4, the number of cache conflicts between T_a and T_b results from the intersection of the MUMBS of T_a and 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, Equation 4 is extended as follows:

$$C(Pa_b^k) = S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b^k) = \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \widehat{\tilde{m}}_{l,r}^{\,k}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{7}$$

In Equation 8, we show the combination of Equation 7 with Equation 5 to estimate the cache reload cost caused by T_b preempting T_a, where T_b has a higher priority than T_a.

$$C_{pre}(T_a, T_b) = C_{miss} \times S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b^{longest}) = C_{miss} \times \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \widehat{\tilde{m}}_{l,r}^{\,k}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{8}$$

where $\tilde{M}_l$ is the MUMBS of task T_l and $M_b^{longest}$ is the set of memory blocks accessed by task T_b when task T_b runs along the longest path.

In Equation 6, C_j is the WCET estimate of T_j without considering preemption. We use SYMTA [10] to estimate WCET. Note that the cost of context switch caused by preemptions is not included in Equation 6. Here, we focus on cache reload cost analysis and assume the cost of a context switch is a constant, C_{cs}, which is equal to the WCET of a context switch. 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 7: 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 [29] 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 cost. We use $C_{pre}(T_i, T_j)$ to represent the cache reload cost 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 cost, Equation 6 can be modified as follows:

$$R_i^k = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + C_{pre}(T_i, T_j) + 2C_{cs}) \tag{9}$$

By comparing Equation 9 with Equation 6, we can find that $\gamma_j = C_{pre}(T_i, T_j) + 2C_{cs}$ in our new WCRT analysis equation. This new equation includes inter- and intra-task cache analysis as proposed in this paper.

Based on Equation 9, we can estimate the WCRT for each task T_i with the following iteration:

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

...

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

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. Note that this iterative formula only requires that the period of each task is fixed. Also, the cache reload cost gives an upper bound of all possible preemption scenarios, no matter when the preemption happens. Thus, we do not require tasks arriving at the beginning of every period. In other words, this method can handle a system with jitter.

In Equation 9, every preemption is tied to an invocation of a task. Thus, no infeasible preemptions are introduced to the WCRT estimate. A preemption is included in our estimate only when a task with a higher priority than the running task arrives (i.e., the condition for preempting is satisfied). However, in the approach of Lee et al., the number of preemptions are estimated separately from the number of invocations of tasks. Due to this separate estimation of the number of preemptions, Lee et al. [25] suffer from a problem that our approach as presented in this paper does not have: infeasible preemptions that cannot happen in any real case could potentially be included in the WCRT estimate. To eliminate this possibility, Lee et al. use an ILP formulation to remove infeasible preemptions.

Now, let us consider the computational complexity of this iteration procedure. Because we conclude that R_i converges and $R_i = R_i^k$ if R_i^k is equal to R_i^{k+1}, R_i has to increase monotonically before the iteration is terminated. R_i has to be increased by $min_{j=0}^{j=i-1}(C_j)$ at least in each iteration. On the other hand, R_i cannot exceeds P_i. Thus, the number of iterations is limited by $\frac{P_i}{min_{j=0}^{j=i-1}(C_j)}$. This implies that the number of iterations has a constant upper bound when the periods and the WCET of tasks are determined. When considering nested preemptions, we have to calculate all cache reload cost $C_{pre}(T_b, T_a)$, where $a < b$, $0 \leq a \leq n-2$ and $1 \leq b \leq n-1$. n is the number of tasks. So the number of cache reload cost is $O(n^2)$, where n is the number of tasks. Note that in [25], in order to estimate the WCRT for one task, all the preemption scenarios have to be investigated. The total number of preemption scenarios scales exponentially with the number of tasks. Thus, our method is more feasible and scalable when there are a large amount of tasks in the system.

VIII. Experimental Results

First, two experiments, each with an application consisting of three tasks, are used to investigate the performance of our approach. The applications run on an ARM9TDMI processor with a 4-way set associative unified 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 [31]. The tasks are supported by Atalanta RTOS developed at Georgia Tech [29]. The whole system is integrated with Seamless CVE provided by Mentor Graphics [30]. The simulation environment is shown in Figure 6.

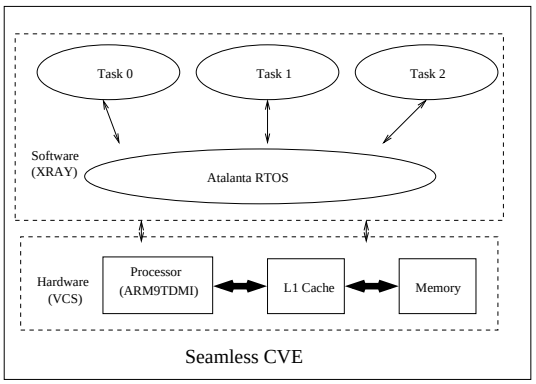


Fig. 6. Simulation Architecture

The tasks in the first experiment, OFDM, ED and MR, are described in Example 1. The tasks in the second experiment are Adaptive Differential Pulse Code Modulation Coder (ADPCMC), ADPCM Decoder (ADPCMD) and Inverse Discrete Cosine Transform (IDCT). ADPCMC and ADPCMD are taken from MediaBench [32], [33]. 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 II.

TABLE II

TASKS

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

In the experiments, we compare five approaches to estimate cache reload cost caused by preemptions.

Approach 1 (A1): All cache lines used by preempting tasks are reloaded for a preemption. Note that this approach is proposed by [22].

Approach 2 (A2): Only lines in the intersection set of lines used by the preempting task and the preempted task are reloaded after a preemption. Our inter-task cache eviction method proposed in [1] is used here.

Approach 3 (A3): 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 et al. in [24] is used here.

Approach 4 (A4): Both inter-task cache eviction analysis and intra-task cache access analysis are used to estimate the cache reload cost. Path analysis is not used in this approach. Note that this approach is used by Lee et al. in [25] to estimate the cache reload cost for each preemption. ILPs as proposed in Lee's approach are constructed

to estimate to WCRT in Approach 4. This approach can potentially include infeasible preemptions in the WCRT estimate which is not existing in our approach. As a verification of this, we have carefully checked and do not observe any infeasible preemptions in the worst case scenarios of any experiments presented in this paper.

Approach 5: 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 five approaches are listed in Table III.

TABLE III

NUMBER OF CACHE LINES TO BE RELOADED

Experiment I						Experiment II					
Preemptions	A1	A2	A3	A4	A5	Preemptions	A1	A2	A3	A4	A5
OFDM by MR	245	134	187	118	88	ADPCMC by IDCT	249	68	98	64	56
OFDM by ED	254	172	187	135	98	ADPCMC by ADPCMD	220	114	98	92	64
ED by MR	245	87	106	85	81	ADPCMD by IDCT	183	58	89	55	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 memory access pattern of the preempted task. Thus, for a certain preempted task, the estimate of cache reload cost 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 III.

The WCRT of OFDM and ED can be calculated based on the results shown in Table III. Notice that MR has the highest priority so that it can never be preempted. So, the WCRT of MR is just equal to its WCET. We also vary C_{miss} from 10 cycles to 40 cycles to investigate the influence of cache miss penalty on the WCRT. The estimate results (Approach 1 thorough Approach 5) and the Actual Response Times (ART) which is the WCRT as observed in simulations are listed in Table IV. We obtain the ART by using the worst case scenarios in experiments. For example, we choose input data specifically so that the longest path in each task is taken in the experiment. We also consider jitter in experiments. We assume that the tasks can arrive earlier or later than their periods. In this case, more preemptions can possibly happen. For example, if two tasks arrive at the same time, the high priority task is

executed. Thus, there is no preemption. However, if the low priority task arrives earlier because of jitter, the low priority task is executed and then possibly preempted by the high priority task when the high priority task arrives. Jitter can affect the WCRT. In our experiments, because the number of tasks is not large and the control flow in each task is not complicated, we can set the input data and choose the preemption scenarios carefully so that the worst case response time can be observed in simulation. However, generally speaking, when there is a large number of tasks and the control flow of each tasks is complicated, there is no guarantee that the worst case scenario can be covered with simulations. Therefore, WCRT cannot be observed with simulations in general cases.

The same results of the second experiment are also listed in Table IV.

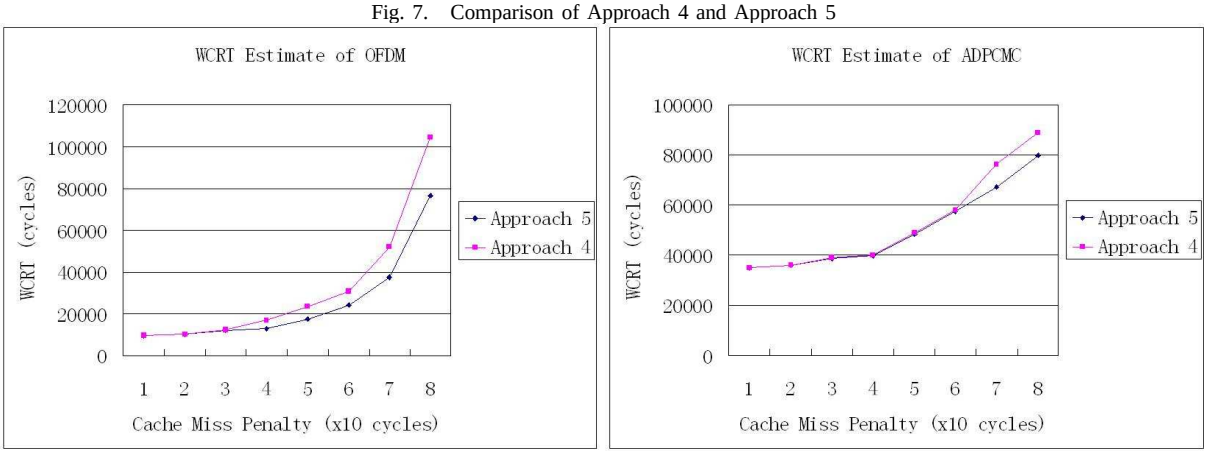
TABLE IV

COMPARISON OF WCRT ESTIMATE

Experiment I								Experiment II						
C_{miss}	Task	A1	A2	A3	A4	A5	ART	Task	A1	A2	A3	A4	A5	ART
10	OFDM	9847	9771	9789	9764	9684	8405	ADPCMC	35743	35701	35071	35027	35863	24186
	ED	2567	2409	2428	2407	2403	2381	ADPCMD	6565	6315	6377	6309	6291	6219
20	OFDM	12510	12242	12378	10424	10264	8516	ADPCMC	48528	38687	37987	35983	34967	24504
	ED	2812	2496	2534	2492	2484	2397	ADPCMD	6931	6431	6555	6419	6383	6242
30	OFDM	23501	19249	17244	12468	12258	8590	ADPCMC	88606	39555	39055	38911	38779	24865
	ED	3057	2583	2640	2577	2565	2434	ADPCMD	7297	6547	6733	6529	6475	6308
40	OFDM	45216	31284	30532	16952	12966	8652	ADPCMC	359239	48714	47722	39931	39755	25173
	ED	3302	2670	2746	2662	2646	2525	ADPCMD	7663	6663	6911	6639	6567	6390

Further, we compare the results of A4 and A5 in Figure 7. The cache miss penalty is changed from 10 cycles to 80 cycles.

Fig. 7. Comparison of Approach 4 and Approach 5



As shown in Figure 7, when the cache miss penalty is small, there are not much difference between Approach 5 and Approach 4. However, when the cache miss penalty becomes larger, our approach (Approach 5) performs better

that the approach of Lee et. al (Approach 5). When the cache miss penalty is 80 clock cycles, the WCRT estimate derived with Approach 5 is 28% less than the WCRT estimate derived from Approach 4. Two factors affect the accuracy of WCRT estimates, the estimate of the number of preemptions and the estimate of cache reload cost for each preemption. The approach of Lee et al. explores all preemption scenarios in order to tight the cache reload cost estimate for each preemption, while our approach (A5) excludes infeasible preemptions. For example, when the cache penalty is 80 cycles, OFDM is preempted by MR 29 times as estimated with Approach 4. However, the number of MR preempting OFDM is only is 21 if estimated with Approach 5. The WCRT estimate is tightened in our approach due to reduction of the estimate of the number of preemptions.

We also executed a third experiment with a larger number of tasks. In this experiment, we have six tasks, OFDM, ADPCMC, ADPCMD, IDCT, ED and MR. The priority and period of each task is listed in Table V. Note that, in order to satisfy the necessary condition of schedulability of a real-time system (i.e., the total utilization of all tasks must be less than 100% [19], [20]), we increase the periods of some tasks as compared to the same tasks in Experiment I and II. OFDM has the lowest priority and MR has the highest prority. The WCET of each task keeps the same.

TABLE V

TASKS IN EXPERIMENT III

	T_1(MR)	T_2(IDCT)	T_3(ED)	T_4(ADPCMD)	T_5(OFDM)	T_6(ADPCMC)
Period(us)	7,000	9,000	13,000	20,000	40,000	50,000
Priority	2	3	4	5	6	7
WCET(us)	830	1580	1392	2839	2830	7675

We use the five different approaches described earlier to estimate the WCRT of the two tasks with the lowest priorities, OFDM and ADPCMC, which may be preempted more frequently than other tasks. Table VI gives the WCRT estimates of OFDM and ADPCMC with different approaches.

TABLE VI

WCRT ESTIMATES IN EXPERIMENT III

	WCRT estimates of ADPCMC					WCRT estimates of OFDM				
C_{miss}	A1	A2	A3	A4	A5	A1	A2	A3	A4	A5
10	51572	76196	77101	75741	75140	16901	16551	17050	16496	16330
20	75585	138608	135610	115342	95387	25904	17199	17242	17001	16757
30	258814	203341	191132	161034	138900	50831	17847	17750	17699	17184
40	6837328	311838	289041	264114	201358	116464	34694	27718	25615	17611

App. 5 and App. 4 are compared in Table VII. By applying path analysis, the WCRT estimate is reduced by up to 32%. Thus, we demonstrate that our approach can tighten WCRT estimate significantly by using path analysis

TABLE VII

COMPARISON OF APP.4 AND APP.5 FOR WCRT ESTIMATES

	C_{miss}			
Task	10	20	30	40
ADPCMC	1%	17%	14%	24%
OFDM	1%	1.4%	2.9%	32%

technique, which is missing in the enhanced approach of Lee et al. [25].

As stated in Section VII, the approach of Lee et al. cannot guarantee removal of all infeasible preemptions. We have one more experiment to show the effect of infeasible preemptions. For example, consider the following scenario based on the incomplete description of task specification of the experiment in Lee et al. [25].

Four tasks as listed in Table VIII are used in the experiment in [25]. When the cache reload penalty is 100 cycles, the WCRT of FIR (i.e., the task with the lowest priority) given by the approach of Lee et al. is 5,323,620 cycles. However, the WCRT estimate resulting from the iteration we proposed in Section VII is 3,778,075 cycles, which shows a reduction of 29%. Note that we use the preemption related cache reload cost as reported in [25]. Since we use the same cache reload cost for each preemption, the difference in WCRT estimate is caused by the the number of preemptions used in WCRT estimate. Apparently, the approach of Lee et al. cannot remove all the infeasible preemptions. Also, note that the cache reload cost here is derived without applying path analysis on the preempting task. We can expect more reduction in WCRT estimate if the path analysis technique proposed in our approach is used.

TABLE VIII

TASKS IN THE PAPER OF LEE ET AL. [25]

Task	Period	WCET
FFT	$320,000$	$60,234 + 280 \times C_{miss}$
LUD	$1,120,000$	$255,998 + 364 \times C_{miss}$
LMS	$1,920,000$	$365,893 + 474 \times C_{miss}$
FIR	$25,600,000$	$557,589 + 405 \times C_{miss}$

IX. CONCLUSION

We propose a WCRT analysis approach in this paper. The cache reload cost caused by preemptions is considered in our approach. Inter-task cache eviction analysis is combined with useful memory block analysis of the preempted task. Compared to previous work, we make these contributions. First, a formal method is proposed to calculate the intersection of cache lines used by the preempting and the preempted task. The method can be uniformly applied to the all types of cache (e.g., direct-mapped, set associative and full associative caches). Second, path analysis

is applied to reduce the estimate of the number of cache lines in the intersection set. Furthermore, the WCRT estimate formula is improved so that the computational complexity of estimating WCRT for one task is linear with the number of tasks in the system, as compared with the exponential complexity in [25]. The experiment shows that our approach can reduce the estimate of WCRT by up to 32%, compared with prior approaches.

For our future work, we plan to expand our analysis approach to work with more than two levels of memory hierarchy. We also aim to improve WCRT analysis for middle priority tasks, e.g., priority 10 out of 20. Finally, we will further examine WCRT analysis in multiprocessor systems with snoopy caches where cache evictions may occur due to behavior on a processor other than the processor which owns the cache.

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)*, pp. 1034-1039, 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] R. White, F. Mueller, C. Healy, D. Whalley and M. Harmon, "Timing Analysis for Data and Wrap-Around Fill Caches," *Real-Time Systems*, Volume 17, Number 2-3, pp. 209-233, 1999,

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

[15] 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).

[16] H. Theiling, C. Ferdinand and R. Wilhelm, "Fast and Precise WCET Prediction by Seperate Cache and Path Analyses," *Real-Time Systems*,Volume 18, Number 2/3, May 2000.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] K. Tindell, A. Burns and 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] C. Lee, J. Hahn, Y.-M. 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," *Proceedings of the Seventeenth IEEE Real-Time Systems Symposium*, pp. 264-274, December 1996.

[27] H. Negi, T. Mitra and A. Roychoudhury, "Accurate Estimation of Cache-related Preemption Delay," *Proceedings of ACM Joint Symposia CODES+ISSS*, pp. 201-206, October 2003.

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

[29] 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, http://www.cc.gatech.edu/tech_reports/index.02.html.

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

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

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

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