Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-Time Systems

Yudong Tan and Vincent Mooney
{ydtan,mooney}@ece.gatech.edu
Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA

Abstract. In this paper, we propose a timing analysis approach for preemptive multi-tasking real-time systems with caches. The approach focuses on the cache reload overhead caused by preemptions. The Worst Case Response Time (WCRT) of each task is estimated by incorporating cache reload overhead. After acquiring the WCRT of each task, we can further analyze the schedulability of the system. Four 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 44% over prior state-of-the-art.

1 Introduction

When designing a real-time system, it is of great importance that the timing of the system be predictable. While underestimating the execution time of tasks in a real-time system may cause catastrophic disasters, especially in hard real-time systems, overestimating the execution time can also lower the utilization of resources such as processors. However, processors with advanced features such as caching and pipelining are widely used in real-time systems nowadays. Using processors with these complicated architectures in real-time systems makes timing analysis more difficult.

In this paper, we propose a timing analysis approach for preemptive multi-tasking real-time systems. The approach focuses on the cache reload overhead caused by preemptions. The Worst Case Response Time (WCRT) of each task is estimated by incorporating cache reload overhead. After acquiring the WCRT of each task, we can further analyze the schedulability of the system. Four 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 44% over prior state-of-the-art.

The rest of this paper is organized as follows. Section 2 investigates the previous work in this field. Section 3 states the problem and defines some terminology used in the paper. Then, an overview of our approach is given. Sections 4, 5 and 6 elaborate the details of our approach. Experimental results are presented in Section 7. The last section concludes the paper.

2 Previous Work

Lots of work has been done to predict the timing properties of real-time systems with caches. In general, this work can be divided into two categories. First, various methods are proposed to achieve predictable system behavior by changing some features such as cache management policies and memory mapping patterns of systems. For example,

two different cache partitioning schemes are presented in [3, 4]. Each task can only use a limited portion of the cache in these two approaches. Compiler optimization and memory remapping techniques are also used to achieve predictable cache behavior [5,6]. These approaches require either customized hardware support such as a specialized cache controller and TLB or modifications to the compilers and OS. Furthermore, utilization of resources such as cache and memory are compromised.

Static analysis is a second category of methods to predict timing properties. 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. Static analysis methods do not require any changes to the system under consideration. Li and Malik propose a WCET analysis approach by using an implicit path enumeration method [7]. Their approach requires path analysis at the granularity of basic blocks. Wolf and Ernst extend the concept of basic blocks to program segments and develop another framework for timing analysis, SYMTA [8]. The precision of time estimation is improved in SYMTA since the overestimate of execution time is reduced. Wilhelm et al. [11,12] propose an abstract interpretation methodology to predict cache behavior. Stenstrom et al. [13] 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. However, both of the aforementioned approaches only consider a system with a single task. They cannot handle multi-tasking systems with preemptions, in which the timing analysis becomes even more complicated.

The behavior of a multi-tasking system is affected greatly by the scheduling algorithm used in the system. In this paper, we assume that a fixed priority scheduling algorithm such as Rate Monotonic Scheduling (RMS) [14, 15] is used. We further assume a single processor with a unified (instruction plus data) set associative L1 cache and secondary memory (the secondary memory can be either on- or off-chip). However, the same method in our approach can be used to analyze timing properties in systems with more than two levels of memory hierarchy. Multiple processor systems may involve cache coherence problems which are beyond the scope of this paper. In order to analyze the schedulability of a system, we estimate the WCRT [16] of each task in the system. In preemptive multi-tasking systems, cache eviction among tasks may extend the response times of tasks. Busquests-Mataix et al. propose an approach to analyze cache-related WCRT in a multi-tasking system [17]. They conservatively assume that all 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 [10]. However, they only consider direct mapped instruction cache. Lee et al. propose another approach for cache analysis when preemptions occur [18, 19]. This approach counts the number of "useful" memory blocks by performing path analysis on the preempted task. They do not take the program structure of the preempting task into consideration. Also, the number of preemption scenarios used in their approach is exponential with the number of tasks. Negi et al. [9] refine the approach of Lee et al. in [18] by applying path analysis. However, inter-task cache eviction is not considered. Also, WCRT analysis is not mentioned in [9].

We propose an approach for inter-task cache eviction analysis in [1, 2]. 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 [18], only those cache lines used by "useful" memory blocks of the preempted task need to be reloaded.

Both the approach we present in [1, 2] and the approach of Lee et al. in [18] have their pros and cons. However, these two methods are complementary. In this paper, we focus on enhancing our approach in [1, 2] by incorporating "useful" memory block analysis in the work of Lee et al. 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 7 we will show examples where we achieve results up to 44% better than the approach of Lee et al.

3 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.

3.1 Problem Statement

In this paper, we target uniprocessor multi-tasking preemptive real-time systems with caches. A Fixed Priority Scheduling (FPS) algorithm such as the Rate Monotonic Algorithm (RMA) is used in the system. Suppose that the system contains n tasks represented with $T_0, T_1, ..., T_{n-1}$. Each task T_i has a fixed priority p_i. We assume that the tasks are sorted in the descending order of their priorities so that we have $p_0 < p_1 < ... < p_{n-1}$. If $p_a < p_b$, T_a has a higher priority than T_b. 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 [7] and SYMTA [8]. 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 the 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.

Definition 1. Worst Case Response Time (WCRT): 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_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 [23]. 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

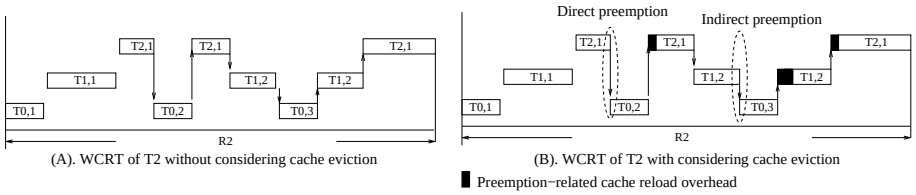

Fig. 1. Example of WCRT

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). □

Inter-task cache eviction(s) caused by preemption(s) affect the WCRT of a task. As shown in Example 1, there are two types of preemption, direct preemption and indirect preemption. For example, T_2 can be preempted directly by T_0 or T_1 because T_2 has the lowest priority. On the other hand, when T_2 is preempted by T_1 and T_1 is running, T_1 can be preempted further by T_0 because T_1 has a lower priority than T_0. Although T_2 is not directly preempted by T_0, T_0 may bring a cache reload overhead to the execution time of T_1, which also extends the response time of T_2. Thus, we need to consider both indirect and direct preemptions caused by T_0 when estimating the WCRT of T_2. Figure 1 illustrates both direct and indirect preemptions.

This paper aims to incorporate inter-cache eviction cost in WCRT analysis by combining the approach of Lee et al. and the approach we presented in [1].

We perform path analysis on the preempted task and the preempting task in order to analyze the cache access pattern of the preempted task. The path analysis is based on a Control Flow Graph (CFG). 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 of the graph G. Each edge $e_i = (v_k, v_j)$ represents the control dependence between two nodes, v_k and v_j. Each node v_i in a CFG represents a Single Feasible Path Program Segment (SFP-PrS) [8].

Definition 2. Single Feasible Path Program Segment (SFP-PrS): SFP-PrS is defined as a hierarchical program segment with exactly one path [8]. □

3.2 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 the work of Lee et al. [18, 19]. We can use the algorithm of Lee et al. in [18] to find the maximum set of these useful memory blocks. The maximum set of useful memory blocks of the preempted task is derived from the program structure and the memory blocks accessed by the preempted task. In summary, we call this type of analysis 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 four steps.

First, we derive the memory trace of each task with the simulation method as used in SYMTA [8]. Here, we assume that there are no dynamic data allocations in tasks and addresses of all the data structures are fixed (e.g., any use of pointers does not result in unpredictable memory accesses). 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. As proposed in [1], path analysis is applied to the preempting task in order to tighten the estimate of cache reload overhead in this step. From the third step, we obtain an estimate of the number of cache lines that need to be reloaded after preemption. Then, we can calculate the cache reload overhead. In the fourth step, we preform WCRT analysis for all tasks based on the results from the third step.

4 Intra-task cache access analysis

According to Condition 2 in Section 3.2, 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 et al. give an approach to calculate the maximum set of such memory blocks.

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 per cache line. A direct mapped cache can be viewed as a special set associative cache which has only one way. We assume that 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 block with an address

of a is loaded to a set associative cache, it can only occupy a cache line with an index of $idx(a)$. In this paper, we assume that the LRU algorithm is used for cache line replacement. However, our approach can also be applied with minor modifications to caches with other replacement algorithms. 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 used in the approach of Lee et al. The inter-task cache eviction analysis algorithm can be applied to all cache line replacement policies.

As we mentioned in Section 3.1, 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 [18]. 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 block 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 [18], 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. The details of their algorithm can be found in [18,19]. 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.

5 Inter-task cache eviction analysis

In [1,2], 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. show that this is not always true. In this paper, we focus on incorporating Lee's intra-task cache access analysis with the approach in [1,2] in order to give a tighter estimate of cache-related preemption delay in multi-tasking preemptive systems.

As stated in Condition 1, the cache lines that may need to be reloaded must be accessed by both the preempted and 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.

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 [2]. Based on this conclusion, we define the Cache Index Induced Partition 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 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].

The definition of CIIP provides a formal representation useful to analyze inter-task cache evictions. 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 $\widehat{m}_{a,r} \in \widehat{M_a},\ \widehat{m}_{b,r} \in \widehat{M_b}$.

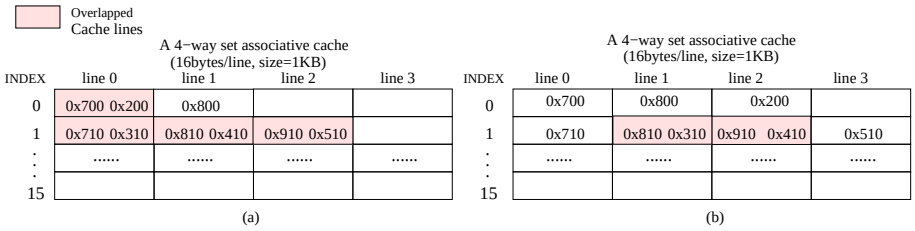

Fig. 2. Conflicts of cache lines in a set associative cache

$S(M_a, M_b)$ denotes an upper bound on the number of cache lines that 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 2. Suppose we have a a 4-way set associative cache with 16 sets. Each cache line has 16 bytes. 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 $M_1 = \{0x700, 0x800, 0x710, 0x810, 0x910\}$ and $M_2 = \{0x200, 0x310, 0x410, 0x510\}$ respectively. The CIIP of M_1 and M_2 are $\widehat{M_1} = \{\{0x700, 0x800\}, \{0x710, 0x810, 0x910\}\}$ 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 2(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., $0x800$ can possibly be mapped to line 0 instead of line 1, but in this case $0x800$ would kick out $0x700$ or vice versa). In any case, the mapping given in Figure 2 gives a case in which the largest amount of cache line overlaps occur. 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 2(b), only two cache lines overlap. Therefore, Equation 2 only gives an upper bound (as opposed to an exact count) for the number of overlapped cache lines. □

Now, we can 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 estimate the cache reload overhead.

Suppose we have two tasks, T_a and T_b. T_b has a higher priority than T_a, thus, T_b can preempt T_a. In the case that T_b preempts T_a, we want to know the number of cache lines that need to be reloaded by T_a after T_a resumes from the preemption.

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 T_a is called the maximum set of useful memory blocks of this task. It is represented with $\tilde{M}_a$. $\widehat{\tilde{M}_a}$ is the CIIP of $\tilde{M}_a$.

We use the approach of Lee et al. to calculate the MUMBS of the preempted task. Only the memory blocks in this set can possibly be reloaded 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 [8]. 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. Note that we can apply path analysis techniques proposed in [1, 2] to the preempting task in order to tighten the estimate of cache reload overhead by reducing the number of memory blocks in M_b.

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 gives an upper bound on the number of cache lines that can possibly need to be reloaded after T_b preempts T_a.

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

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

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}. Then, $C_{pre}(T_a, T_b)$ can be calculated with the following equation:

$$C_{pre}(T_a, T_b) = S(\tilde{M}_a, M_b) \times C_{miss} \tag{4}$$

6 WCRT analysis

We can use the Worst Case Response Time (WCRT) to analyze schedulability of a multi-tasking real-time analysis as shown in [17]. The approach uses the following recursive equations to calculate the WCRT R_i of the task T_i.

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times (C_j + \gamma_j) \tag{5}$$

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 3.1. 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, task T_i cannot be guaranteed to be scheduled successfully.

In [17], the authors assume that all the cache lines used by the preempting task need to be reloaded after the preemption. As we pointed out in Section 4 and 5, 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 5.

When we estimate the WCRT of task T_i, we need to consider all possible preemptions caused by each task, $T_k,\ 0 \leq k < i$, which has a higher priority than T_i. T_k can preempt T_i directly, which brings a cache reload overhead of $C_{pre}(T_i.T_k)$ to the WCRT of T_i. T_i can also be preempted by T_k indirectly. Let us consider Example 3.

Example 3. Consider two cases in Figure 1(B). T_2 is preempted by T_0 twice. At the first time, T_2 is preempted by T_0 directly, thus, the cache reload overhead is $C_{pre}(T_2, T_0)$. In the second case, T_2 is preempted by T_1 first, then T_1 is preempted by T_0 further. Thus, T_2 is preempted by T_0 indirectly in this case. The cache reload overhead caused by this indirect preemption is $C_{pre}(T_1, T_0)$. In practice, it is difficult to know if T_0 preempts T_2 directly or indirectly. In order to avoid underestimating the WCRT of T_2, we use

$max(C_{pre}(T_1, T_0), C_{pre}(T_2, T_0))$ as the cache reload overhead caused by T_0 preempting T_2 (i.e., either indirectly or directly).□

Example 3 shows that when one or more than one task has a priority higher than T_i and lower than T_k, the cache reload overhead caused by T_k preempting T_i depends on the actual preemption scenarios, which is difficult to predict in practice. Thus, in order to avoid underestimating WCRT, we use an upper bound, $max^i_{l=k+1}\{C_{pre}(T_l, T_k)\}$, to uniformly estimate the cache reload cost caused by T_k preempting T_i.

In Equation 5, C_j is the WCET estimate of T_j without considering preemption. We use SYMTA [8] to estimate WCET. Note that the cost of context switch caused by preemptions is not included in Equation 5. 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. 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.

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. By considering the cache reload overhead, Equation 5 can be modified as follows:

$$R_i = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i}{P_j} \rceil \times (C_j + max^i_{l=j+1}\{C_{pre}(T_l, T_j)\} + 2C_{cs}) \quad (6)$$

Based on Equation 6,we can estimate the WCRT for each task T_i with the following iteration:

$R_i^0 = C_i$;

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

...

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

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. Only if R_i is less than the deadline of T_i can T_i be guaranteed to be successfully scheduled. Hence, we can analyze the schedulability of the system based on the WCRT estimate of each task.

In Equation 6, 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. [19] 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 exceed 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. In each iteration, we have to calculate $max_{l=j+1}^{i}\{C_{pre}(T_l, T_j)\}$. This can be done by calculating all possible preemption scenarios $C_{pre}(T_b, T_a)$, 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 preemption-related cache reload cost is $O(n^2)$, where n is the number of tasks. Note that in [19], in order to estimate the WCRT for one task, all the preemption scenarios have to be investigated. The total number of preemption scenarios is exponential in the number of tasks. Thus, our method is more feasible and scalable than [19] when there are a large amount of tasks in the system.

7 Experimental Results

Our experiments are 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 [22]. The whole system is integrated with Seamless CVE provided by Mentor Graphics [21]. The tasks are supported by the Atalanta RTOS developed at Georgia Tech [20].

First, we have two experiments each with three tasks. The tasks in the first experiment, IDCT, ADPCMD and ADPCMC, are described in Example 1. The tasks in the second experiment are a Mobile Robot control program (MR), an Edge Detection algorithm (ED) and an OFDM transmitter (OFDM). We use SYMTA to estimate the WCET of each task in the experiments. The periods, priorities and WCET of tasks in each experiment are listed in Table 1.

Table 1. Tasks

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

In the experiment, we compare four approaches to estimate cache reload overhead caused by preemptions. Furthermore, we calculate the WCRT of each task by using Equation 6.

Approach 1 (A1): All cache lines used by preempting tasks are reloaded for a preemption. Note that this approach is proposed by [17].

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

Approach 3 (A3): Intra-task cache access analysis for the preempted task proposed by Lee et al. in [19] is used here. Note that no path analysis is applied in this approach. This approach can potentially include infeasible preemptions in the WCRT estimate (which does not exist in our approach).

Approach 4 (A4): Both inter-task cache eviction analysis and intra-task cache access

Table 2. Number of cache lines to be reloaded

Experiment I					Experiment II				
Preemptions	A1	A2	A3	A4	Preemptions	A1	A2	A3	A4
ADPCMC by IDCT	249	68	64	56	OFDM by MR	245	134	118	88
ADPCMC by ADPCMD	220	114	92	64	OFDM by ED	254	172	135	98
ADPCMD by IDCT	183	58	55	46	ED by MR	245	87	85	81

analysis are used to estimate the cache reload cost. Also, path analysis proposed in [1] is applied to the preempting task. Approach 4 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 2.

Table 3. Comparison of WCRT estimate

C_{miss}	Experiment I						Experiment II					
	Task	A1	A2	A3	A4	ART	Task	A1	A2	A3	A4	ART
10	ADPCMC	35743	29392	29172	28836	23512	OFDM	9847	9350	9279	6456	6113
	ADPCMD	6565	6315	6309	6291	6190	ED	2567	2409	2407	2403	2382
20	ADPCMC	48528	35607	35079	29420	23867	OFDM	12510	10096	9954	9524	6211
	ADPCMD	6931	6431	6419	6383	6223	ED	2812	2496	2492	2484	2400
30	ADPCMC	88606	38997	38139	35175	24101	OFDM	23501	12174	11964	99844	6255
	ADPCMD	7297	6547	6529	6475	6278	ED	3057	2583	577	2565	2426
40	ADPCMC	359239	48146	39335	35843	24353	OFDM	45216	16700	12774	10444	6362
	ADPCMD	7663	6663	6639	6567	6354	ED	3302	2670	2662	2646	2525

We use SYMTA [8] 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.

In the first experiment, the WCRT of ADPCMC and ADPCMD can be calculated based on the results shown in Table 2. Notice that IDCT has the highest priority and thus cannot ever be preempted. As a result, the WCRT of IDCT 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 (Approach 1 through Approach 4) and the Actual Response Times (ART) – which is the WCRT as observed in simulation – are listed in Table 3. (Please note that as we did not exhaust all possible input data in our simulations, the observed ART may underpredict WCRT; nonetheless, for our experiments the ART should be quite close to the WCRT considering the fact that the tasks in the experiments do not have complicated control flows and the size of the input data is fixed). Table 4 lists the improvement of our approach (Approach 4) over all other approaches (Approach 1, Approach 2 and Approach 3) in these first two experiments.

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. Inter-task cache evictions are also considered in this approach. However, there is no path analysis in this approach. As compared with Approach 3, Approach 4, in which we apply path analysis techniques on the preempting task, achieves a significant reduction of up to 30% in the WCRT estimate of OFDM.

Table 4. Comparison of results for different approaches

Comparison	Experiment I					Experiment II				
		Cache Penalty (cycles)					Cache Penalty (cycles)			
	Task	10	20	30	40	Task	10	20	30	40
A4 vs. A1	ADPCMC	19%	39%	60%	92%	OFDM	34%	23%	57%	77%
	ADPCMD	4%	8%	11%	14%	ED	6%	12%	16%	20%
A4 vs. A2	ADPCMC	2%	17%	10%	26%	OFDM	31%	6%	18%	38%
	ADPCMD	1%	1%	1%	1%	ED	0.2%	0.5%	1%	1%
A4 vs. A3	ADPCMC	1.4%	16%	8%	9%	OFDM	30%	4%	17%	18%
	ADPCMD	0.2%	0.3%	0.5%	0.6%	ED	0.3%	0.6%	0.8%	1%

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 5. 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% [14, 15]), we increase the periods of some tasks as compared to the same tasks in experiment 1 and experiment 2. ADPCMC has the lowest priority and MR has the highest prority. The WCET of each task stays the same.

Table 5. 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 four 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 6 gives the WCRT estimates of OFDM and ADPCMC with the different approaches.

Table 6. WCRT estimates in Experiment III

WCRT estimates of ADPCMC						WCRT estimates of OFDM				
C_{miss}	A1	A2	A3	A4	A4 vs.A3	A1	A2	A3	A4	A4 vs. A3
10	51572	34837	34336	33781	2%	16901	16217	15948	15643	2%
20	75585	58646	51990	38235	27%	25904	17531	16993	16383	4%
30	258814	75673	69025	57496	18%	50831	25756	24697	17123	31%
40	6837328	152023	76729	68599	11%	116464	33690	31834	17863	44%

Approach 3 and Approach 4 are compared in Table 6. By applying path analysis, the WCRT estimate is reduced by up to 44%. Thus, we demonstrate that our approach can tighten WCRT estimate significantly by using path analysis technique, which is missing in the enhanced approach of Lee et al. [19].

As stated in Section 6, the approach of Lee et al. cannot guarantee removal of all infeasible preemptions. We have one last (actually, fourth) experiment to show the

effect of infeasible preemptions. For example, consider the following scenario based on the experiment in Lee et al. [19].

The four tasks listed in Table 7 are used in the experiment in [19]. When the cache reload penalty is 100 cycles (this is the penalty used by Lee et al. in [19]), 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 6 is 3,297,383 cycles, which shows a reduction of 38%. Note that we use the preemption related cache reload cost as reported in [19]. 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.

Table 7. Tasks in the paper of Lee et al. [19]

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	$25,600,000$	$557,589 + 405 \times C_{miss}$

8 Conclusion

We propose a method to analyze the preemption cost caused by cache eviction in a multi-tasking real-time system. The method first analyzes the maximum set of memory blocks in the preempted task that can possibly cause cache reload. Then, the method incorporates the inter-task cache eviction behavior by calculating the intersection set of cache lines used by the preempting task and the preempted task. By combining these two approaches, we achieve over prior state-of-the-art up to 44% reduction in WCRT estimate in our experiments.

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

9 Acknowledgment

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, Sun and Synopsys. We also thank Jan Staschulat and Prof. Dr. Rolf Ernst for their help in using SYMTA.

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. Y. Tan and V. Mooney,"Timing Analysis for Preemptive Multi-tasking Real-time Systems with Caches," Technical Report, GIT-CC-04-02, Georgia Institute of Technology, February 2004.
3. D. Kirk, "SMART (Strategic Memory Allocation for Real-Time) Cache Design", *Proceedings of IEEE 10th Real-Time System Symposium*, pp. 229-237, December 1989.
4. 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.

5. 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.
6. 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.
7. Y. Li and S. Malik, *Performance Analysis of Real-Time Embedded Software*, Kluwer Academic Publishers, Boston, 1999.
8. F. Wolf, *Behavioral Intervals in Embedded Software*, Kluwer Academic Publishers, Norwell, MA, 2002.
9. H. Negi, T. Mitra and A. Roychoudhury, "Accurate Estimation of Cache-related Preemption Delay," *Proceedings of ACM Joint Symposium CODES+ISSS*, pp. 201-206, October 2003.
10. 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.
11. 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).
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. Mentor Graphics, Seamless Hardware/Software Co-Verification, http://www.mentor.com/seamless/.
22. Mentor Graphics XRAY Debugger, http://www.mentor.com/embedded/xray/.
23. MediaBench, http://cares.icsl.ucla.edu/MediaBench/.
24. Berkeley MPEG2 decoder, http://bmrc.berkeley.edu/frame/research/mpeg/.