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

Yudong Tan and Vincent J. Mooney III, {ydtan, mooney}@ece.gatech.edu
Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering, Georgia Institute of Technology
Atlanta, GA 30332, USA

Abstract

In this paper, we propose an approach to estimate the Worst Case Response Time (WCRT) of tasks in a preemptive multi-tasking single-processor real-time system with a set associative cache. The approach focuses on analyzing the cache reload overhead caused by preemptions. We combine inter-task cache eviction behavior analysis and path analysis of the preempted task to reduce, in our analysis, the estimate of the number of cache lines that can possibly be evicted by the preempting task (thus requiring a reload by the preempted task). A mobile robot application which contains three tasks is used to test our approach. The experimental results show that our approach can tighten the WCRT estimate by up to 73% over prior state-of-the-art.

1. Introduction

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

In this paper, we give an approach to analyze the Worst Case Response Time (WCRT) of tasks. We target a single-processor preemptive multi-tasking system with set associative caches. The approach focuses on the cache reload overhead caused by preemption and imposed on the preempted task. Inter-task cache eviction behavior analysis is combined with path analysis of the preempted task to tighten the estimate of the number of cache lines to be reloaded by the preempted task. A mobile robot application which contains three tasks is used to test the performance of our approach.

The remaining of this paper is organized as follows. Section 2 introduces the previous work in the field of timing analysis. Section 3 defines terminology used in this paper. Section 4 gives the details of our approach. Experimental results are presented in Section 5. The last section concludes the paper.

1530-1591/04 $20.00 (c) 2004 IEEE

2. Previous Work

A cache is one of the main factors complicating timing analysis in real-time systems. Two categories of methods can be applied to predict cache behavior. One is limiting cache usage. This can be implemented by hardware approaches such as cache partitioning [1, 2], or, by software approaches such as compiler optimizations and memory remapping [3, 4]. Usually, these schemes need specialized hardware support in the cache controllers or TLBs as well as custom modifications to the compilers used. Moreover, cache utilization is compromised in these schemes, because either the cache allocation strategy is more strict than conventional caches such as in [1, 2] or the memory-to-cache mapping is more restrictive such as in [3, 4].

The second category of methods to predict cache behavior is to use static analysis methods. Such methods analyze cache behavior and make restrictive assumptions in order to predict Worst Case Execution Time (WCET) or Worst Case Response Time (WCRT) of tasks in real-time systems. Li and Malik contributed to WCET analysis by proposing an explicit path enumeration method [5]. 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 for timing analysis [6]. By extending basic blocks to program segments, the overestimate of execution time in boundaries of basic blocks is reduced, thus improving the precision of timing estimate. Wolf and Ernst developed a framework for timing analysis, SYMTA [6]. However, both of 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 Rate Monotonic Scheduling (RMS) algorithm is used in the system [7, 8]. 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). The purpose of timing analysis is to verify the schedulability of tasks. In this paper, we use the Worst Case Response Time (WCRT) [9] to analyze schedulability. Busquests-Mataix propose an approach to analyze cache eviction cost in a multi-tasking system [10]. They conservatively assume that all the cache lines used by the preempting lines need to be reloaded by the preempted task when the preempted task is resumed. [11] also give an approach for cache analysis in preemptions. This approach counts the number of "useful cache lines" by performing path analysis on the preempted task. However, they assume that all "useful cache lines" of the preempted task are evicted by the preempting task, which might not be true. In our approach, we show that the cache lines to be reloaded are not only determined

by the preempted task, but also by the preempting task. Thus, the intersection of cache lines used by the preempted task and the preempting task needs to be analyzed.

This paper presents the most accurate WCRT method known to the authors to date for a multi-tasking single-processor system using set-associative or direct mapped instruction and data caches. In Section 5 we will show examples where we achieve results up to 73% better than prior art.

3. Terminology

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

We assume that there are n tasks in the system, which are represented with $T_1, T_2, ..., T_n$. Each task T_i has a period P_i. T_i is ready to run at the beginning of its period. The deadline of T_i is at the end of its period. A fixed priority scheduling algorithm is used for scheduling; thus, each task has a fixed priority, p_i. The Worst Case Execution Time (WCET) of task T_i is denoted with C_i. This WCET can be estimated initially with existing analysis tools such as Cinderella [5] and SYMTA [6]. In this paper, we use SYMTA to derive the WCET of tasks. We will discuss later how to estimate WCRT on the basis of WCET in a multi-tasking system. Tasks are executed periodically. We use $T_{i,j}$ to represent the j^{th} run of Task T_i. Note that our approach also works with other scheduling algorithms, e.g., when scheduling tasks with precedence constraints.

In a multi-tasking system,we aim at estimating the Worst Case Response Time (WCRT) of tasks, as defined in [9], 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. □

Example 1. We have three tasks T_1, T_2 and T_3. T_1 is a Mobile Robot control application (MR). The mobile robot updates its behavior every 3.5ms. The second task, T_2 is an Edge Detection application (ED) and is invoked every 6.5ms to process the images of obstacles detected by the robot. The third task, T_3, which is an OFDM transmitter, is invoked to communicate with other robots every 40ms. Figure 1 shows this example. In this example, three tasks arrive at time instant 0. However, T_3 is not executed until there are no instances of T_1 or T_2 ready to run. During the execution of T_3, it could be preempted by T_1 or T_2, which is shown in Figure 1. The response time of T_3 is the time from 0 to the time when T_3 is completed. We need to estimate the response time of such a task in the worst case. □

When one task is preempted, the cache lines used by the preempted task may be evicted by the preempting task. This forces the preempted task to reload these cache lines when the preempted task is executed and the evicted cache lines are accessed again. The cache evictions cause an overhead in execution time of the preempted task. In order to predict the cache reload overhead caused by inter-task cache eviction, we need to analyze cache eviction behavior first.

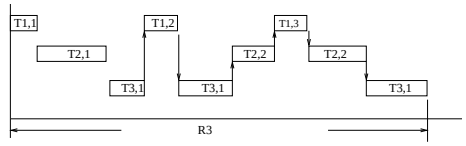

Figure 1. Example of WCRT

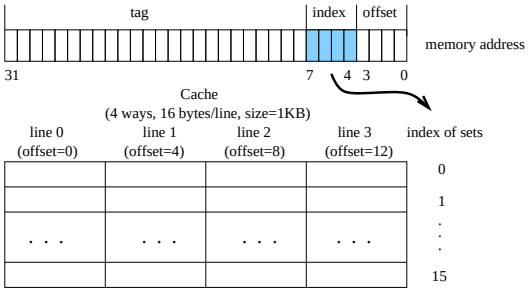

Figure 2. Example of a set associative cache

A cache is defined by three parameters: size, the number of cache lines in a set (i.e., the number of ways) and the number of bytes/words in a cache line. A direct mapped cache can be viewed as a special set associative cache which only has one way. The sets in a cache are indexed sequentially, starting from 0. All the cache lines in a set have the same index. Accordingly, a memory address is divided into three parts: the offset, the index and the tag. When a memory block is loaded to a set associative cache, it can only occupy a cache line in the set which has the same index as the index in the memory block address. We use $idx(a)$ to denote the index of a memory address a.

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. □

Cache eviction only happens among memory blocks that need to be loaded to the same set in a set associative cache. The memory blocks that are loaded into the same set in the cache have the same index. Intuitively, we can divide memory blocks into different subsets according to their index.

Suppose we have a set of r memory block addresses, $M = \{m_0, m_1, ..., m_{r-1}\}$ 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.

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

When the memory blocks in the same subset defined above are accessed, these memory blocks are loaded into the same set in the cache because they have the same index. Thus, cache evictions can happen among these memory blocks (i.e., with the same index).

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

Definition 2. Cache Index Induced Partition (CIIP) of a memory block address set: Suppose we have a set of memory block addresses, $M = \{m_0, m_1, ..., m_{r-1}\}$ 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 lines, which is denoted by $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$. Each $\widehat{m}_i = \{m_k \in M | idx(m_k) = i\}$ is a subset of M. We call $\widehat{M}$ the CIIP of M.□

The CIIP of a memory address set categorizes the memory block addresses according to their indices in the cache. Cache evictions

can only happen among memory blocks that are in the same subset in the CIIP.

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

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

In this paper, we will also perform path analysis on the preempted task in order to tighten the WCRT estimate. The path analysis is based on a Control Flow Graph (CFG) which describes the control structure of a program. A CFG is represented with a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_m\}$ is the set of nodes and $E = \{e_1, e_2, ..., e_n\}$ is the set of edges. Each edge $e_i = (v_k, v_j)$ represents the control dependence between two nodes, v_k and v_j. Usually, each node v_i in a CFG represents a basic block in a program. Wolf and Ernst extend the basic block concept to Single Feasible Path Program Segment (SFP-Prs) in [6]. A Program Segment can be viewed as a sequence of basic blocks with exactly on entry and one exit.

Definition 3. Single Feasible Path Program Segment (SFP-Prs): SFP-Prs is defined as a hierarchical program segment with exactly one path [6]. □

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 $SFP\_Prs(T_i, v_j)$.

4. WCRT Analysis

The main purpose of timing analysis in a multi-tasking system is to verify the schedulability of all tasks in the system. If we already know the WCRT of tasks, we only need to check if the WCRT of all tasks are earlier than their deadlines. WCRT may be affected by cache behavior, pipelining and scheduling algorithms. In this paper, we assume that the RMS algorithm is used for scheduling and only focus on the cache eviction analysis.

In this section, we give our approach to analyze cache evictions caused by preemption in a multi-tasking real-time system. The approach consists of three steps. First, we calculate the intersection set of cache lines used by the preempting task and the preempted task to find an upper bound of the number of cache lines which will be reloaded by the preempted task. Next, we apply path analysis to remove the cache lines which will not be accessed by the preempted task from the intersection set. Because these cache lines are not accessed after the preempted task resumes, we do not need to consider these cache lines when estimating WCRT of the preempted task. Finally, we use an iterative method to calculate the WCRT.

The architecture we target in this paper is a single processor system with a unified set associative L1 cache. We only consider two levels of memory hierarchy in our approach. For our future work, we plan to extend this to three or more levels of memory hierarchy.

4.1. WCRT Analysis

One approach for schedulability analysis is the Worst Case Response Time (WCRT) approach which can be found in [9]. The approach uses the following recursive equations to calculate the WCRT R_i of a task T_i:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times C_j \quad --(2)$$

where $hp(i)$ is the set of tasks whose priorities are higher than T_i and P_j is the period of Task T_j as defined in Section 3. In this equation, the term $\sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times C_j$ reflects the interference of preempting tasks during the execution time of T_i. This equation can be calculated iteratively. When R_i converges, we obtain WCRT of T_i. Then, R_i is compared with the deadline of T_i to determine if T_i can meet its deadline or not. If the equation diverges, the task cannot meet its deadline.

In this approach, C_j is the WCET estimate of T_j without considering preemption. The WCET of T_j can be derived with existing approaches such as Cinderella [5] and SYMTA [6]. The approaches of Cinderella and SYMTA do not consider the costs of inter-task cache eviction and context switch caused by preemptions. Therefore, the WCRT of tasks might be underestimated in Equation (2). Here, we focus on inter-task cache eviction analysis and assume the cost of a context switch is a constant, C_{cs}, which is equal to the WCET of a context switch. Example 4 gives the context switch cost for our simulation architecture. The context switch function cannot be preempted, so the context switch cost is not affected by inter-task cache eviction. Therefore, it is reasonable to assume the context switch cost is a constant, which is its WCET. The context switch function is called twice in every preemption, once for switching to the preempting task and once for resuming the preempted task.

Example 4. In our simulation architecture, we have an ARM9TDMI processor with two levels of memory, a 32KB 4-way set associative L1 cache and 256MB SRAM. The cache miss penalty is 20 cycles. The Atalanta RTOS developed at Georgia Tech [12] 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. □

The cache lines that should be included in the estimate of cache reload cost imposed on the preempted task by preemptions must satisfy two conditions as follows.

Condition 1. These cache lines are used by the preempting task.

Condition 2. These cache lines are required by the preempted task after the preempted task is resumed.

According to these two conditions, the cache eviction cost is determined by the preempting task and the preempted task together. We use $C_{pre}(T_i, T_j)$ to represent the cache eviction cost imposed on task T_i when T_i is preempted by task T_j. Equation (2) can be

modified as follows to no longer underestimate R_i:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times (C_j + C_{pre}(T_i, T_j) + 2C_{cs}) \quad --(3)$$

Now, we need to determine the cache eviction cost caused by preemption, i.e., the value of $C_{pre}(T_i, T_j)$.

4.2. Inter-task cache eviction analysis

In multi-tasking systems, inter-task cache evictions resulting from preemptions may cause cache reloading when the preempted task resumes. In [10], the authors assume that all the cache lines used by the preempting task have to be reloaded when the preempted task is resumed. This approach implies that all the cache lines used by the preempting task will also be required by the preempted task later on. Obviously, this is not always true. Considering Conditions 1 and 2, it is not difficult to find that the cache lines that will be reloaded by the preempted task must be in the intersection set of the cache lines used by the preempting and the preempted task.

Let us state the problem formally. Suppose we have two tasks T_a and T_b. All 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 that 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:

$$S(T_a, T_b) = \sum_{r=0}^{N-1} min\{|\widehat{m}_{a,r}|,\ |\widehat{m}_{b,r}|,\ L\},$$
$$where\ \widehat{m}_{a,r} \in \widehat{M_a},\ \widehat{m}_{b,r} \in \widehat{M_b} \quad --(4)$$

where $S(T_a, T_b)$ denotes the upper bound on the number of reloaded cache lines caused by T_b preempting T_a.

Example 5: Suppose we have a cache as defined in Example 2 and two tasks T_1 and T_2. The memory block addresses accessed by T_1 and T_2 are contained in $M_1 = \{0x000, 0x100, 0x010, 0x110, 0x210\}$ and $M_2 = \{0x200, 0x400, 0x310, 0x410\}$ respectively. The CIIP of M_1 is $\widehat{M_1} = \{\{0x000, 0x100\}, \{0x010, 0x110, 0x210\}\}$, and the CIIP of M_2 is $\widehat{M_2} = \{\{0x200, 0x400\}, \{0x310, 0x410\}\}$. 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 4. However, if we map the memory blocks in M_1 and M_2 to the cache as shown in the Figure 3(b), only one cache line is overlapped. Therefore, Equation 4 only gives an upper bound for the number of overlapped cache lines.□

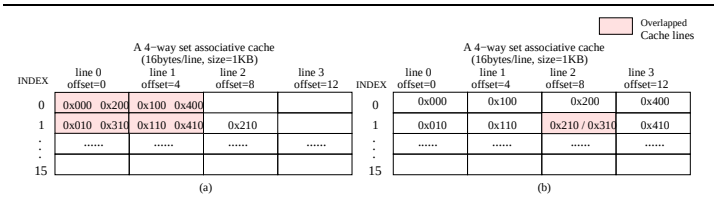

Figure 3. Conflicts of cache lines in a cache

In order to apply Equation 4 to estimate the upper bound for the number of cache lines to be reloaded after a preemption, we need to know the memory block addresses accessed by the preempting task and the preempted task.

Next, we will discuss how to find the memory blocks accessed by a task by using path analysis techniques.

4.3. Path Analysis

The set M_i used in the section above contains all the memory block addresses that can possibly be accessed by the preempted task T_i, if we do not use any path analysis methods. In this case, the result derived from Equation 4 only gives an upper bound for the number of cache lines that could be potentially reloaded by the preempted task. However, since the preempted task might have more than one feasible path and only one path is executed, some memory blocks may not be accessed, in which case there would be no need to reload the cache lines mapped from those memory blocks. Example 6 gives such a case.

Example 6. 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. □

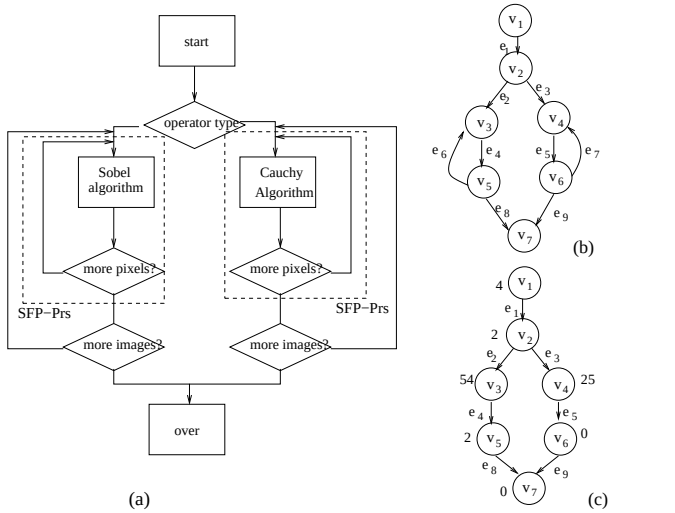

Figure 4. CFG of ED

The issue presented in Example 6 can be described more generally. Suppose we have two tasks in a system with an L-way set associative cache, T_i and T_j. The largest index of the cache is $N-1$. T_j has a higher priority than T_i. Thus, T_j can preempt T_i. We use M_j to represent the set of all memory block addresses that can be possibly accessed by T_j. The CFG of T_i is $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, v_{i,2}, ..., v_{i,n}\}$ and $E_i = \{e_{i,1}, e_{i,2}, ..., e_{i,m}\}$. A path in G_i can be represented with $Pa_i^k = \{v_{i,i_1}, e_{i,i_1}, v_{i,i_2}, e_{i,i_2}, ..., v_{i,i_p}\}$. We use M_i^k to denote the set of memory block addresses accessed by the task T_i when T_i runs along the path Pa_i^k. The CIIP of M_i^k is $\widehat{M}_i^k = \{\widehat{m}_{i,0}^k, \widehat{m}_{i,1}^k, ..., \widehat{m}_{i,N-1}^k\}$. When Pa_i^k is determined, $M_{i,k}$ can be derived from the simulation method as used in SYMTA [6]. We assume that there is no dynamic data allocation and the addresses for all data structures are fixed. Now, we need to find a path in the preempted task T_i. When T_i takes this path, the memory blocks loaded to the cache have the largest overlap with the cache lines that can possibly used by the preempting task. In other words, when T_i takes this path, the number of cache lines evicted by T_j and also used by T_i is the largest. This problem can be transformed to a problem of finding the longest path in a graph.

First, since the cache lines in all SFP-Prs only need to be reloaded once, we can remove all the cycles caused by loops in the CFG. In Example 6, edge e_6 and edge e_7 are removed, which is shown in Figure 4 (c). Next, we define a cost function for the path Pa_i^k in the preempted task T_i.

$$C(Pa_i^k) = S(M_j, M_i^k) = \sum_{r=0}^{N-1} min\{|\widehat{m}_{i,r}^k|,\ |\widehat{m}_{j,r}|,\ L\} \quad --(5)$$

The cost of a path Pa_i^k in the preempted task T_i is defined as the maximum number of cache lines that can be possibly overlapped with the cache lines used by the preempting task T_j, when the preempted task T_i runs along the path Pa_i^k.

By using this cost function, we can apply a Dynamic Programming algorithm to find the longest path in the CFG of T_i. Suppose the longest path is represented with $Pa_{longest}$, the number of cache lines to be reloaded in the worst case is bounded by the cost of $Pa_{longest}$.

Furthermore, we assume that cache miss penalty is a constant, which is represented by C_{miss}. The cache reload overhead caused by T_j preempting T_i, $C_{pre}(T_i, T_j)$ can be estimated with the following formula,

$$C_{pre}(T_i, T_j) = C(Pa_{longest}) \times C_{miss} \quad --(6)$$

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.

4.4. Overall Approach

Putting all the steps described above together, we can derive our WCRT estimate approach. Suppose we want to estimate the WCRT of T_i. All the SFP-Prs of T_i are represented by the nodes in the CFG of T_i, which are $v_1, v_2, ..., v_m$. All the tasks that have higher priorities than T_i are in the set $hp(i) = \{T_{i_1}, T_{i_2}, ..., T_{i_k}\}$. Also, we assume that we have derived the WCET C_i of every task $T_i, i = 1, 2, ..., n$, and the CFG of all tasks except the tasks with the highest priority by using SYMTA. Because the tasks with the highest priority are not preempted, we will not perform path analysis on these tasks.

Step 1. For every task in $hp(i)$, derive the set of all memory block addresses that can be possibly accessed by the task in $hp(i)$ by the simulation approach as used in SYMTA [6]. Here, we assume that there is no dynamic data allocation in the program. For our future work, we plan to extend our research to handle dynamic data allocation.

Step 2. For every task $T_{i_k} \in hp(i)$, use the cost function defined in Equation (5) to estimate the number of cache lines to be reloaded after T_{i_k} preempts T_i. Furthermore, the cache reload overhead caused by T_{i_k} preempting T_i, which is $C_{pre}(T_i, T_{i_k})$, can be derived with Equation (6).

Step 3. Use the following iterations to calculate WCRT of T_i.

$R_i^0 = C_i$;

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

...

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

This iterative procedure terminates when R_i converges. The final value of R_i is the WCRT of T_i.

In order to analyze the schedulability of the system, we compare the WCRT of each task with its deadline. Because we assume that the deadlines of tasks are equal to their periods, we can conclude that a task cannot meet its deadline if the WCRT of a task is longer than its period or if the iteration procedure in Step 3 diverges.

Task	WCET(us)	Period(us)	Priority
T_1(OFDM)	2830	40,000	4
T_2(ED)	1392	6,500	3
T_3(MR)	830	3,500	2

Table 1. Tasks

5. Experimental Results

We use the application described in Example 1 to test our approach. Table 1 lists the WCET, periods and priorities of three tasks: OFDM, ED and MR. For WCET, we use SYMTA, which is a single-task based WCET estimate approach [6] as mentioned in Section 4.1. MR has the highest priority and OFDM has the lowest priority.

The applications are run on an ARM9TDMI processor with a 32KB 4-way set associative cache. Each line in the cache is 16 bytes; thus, there are 512 lines in each "way" of the cache. The instruction set is simulated with XRAY [14]. The tasks are supported by the Atalanta RTOS developed at Georgia Tech [12]. The whole system is simulated with Seamless CVE provided by Mentor Graphics [13].

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

Approach 1: All cache lines used by preempting tasks are reloaded for a preemption. Note that this approach is proposed by [10].

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

Approach 3: Path analysis for the preempted task is added to Approach 2 to reduce the estimate of the number of cache lines to be reloaded, as explained in this paper.

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

	App. 1	App. 2	App. 3
OFDM by MR	245	134	111
OFDM by ED	254	172	135
ED by MR	245	82	77

Table 2. Number of cache lines to be reloaded

cache lines to be reloaded by the preempted task, as shown in Approach 2. On the other hand, due to the existence of multiple feasible paths in the preempted task, the preempted task cannot access all the memory blocks in one run after it is resumed. Hence, we use path analysis techniques to find the longest path of the preempted task in terms of the cost function defined in Equation (5). Approach 3 integrates such path analysis based on Approach 2. The results show that performing path analysis on the preempted task can further reduce the estimate of the number of cache lines to be reloaded.

C_{miss}(cycles)	Task	App. 1	App. 2	App. 3	ART
10	OFDM	9847	9350	9207	6113
	ED	2567	2404	2399	2382
20	OFDM	12510	10096	9810	6211
	ED	2812	2486	2476	2400
30	OFDM	23501	12174	10413	6255
	ED	3057	2568	2553	2426
40	OFDM	45216	16700	12390	6362
	ED	3302	2650	2630	2525

Table 3. Comparison of WCRT estimate

We also vary C_{miss} from 10 cycles to 40 cycle to investigate the influence of cache miss penalty on WCRT estimate. The estimate results and the Actual Response Times (ART) are listed in Table 3. Table 4 lists the improvement of our approach (Approach 3) over Approach 1.

Compared with Approach 1, Approach 3 achieves a reduction of 73% in WCRT estimate of OFDM when the cache penalty is 40 cycles. The WCRT estimate of ED is also reduced by 20.4% in this case.

6. Conclusion

We propose a method to analyze the preemption cost caused by cache eviction in a multi-tasking real-time system. The method analyzes the inter-task cache eviction behavior by calculating the intersection set of cache lines used by the preempting task and the preempted task. Furthermore, path analysis is used to eliminate cache lines that will not be accessed in a task from being used in the estimate. By combining these two approaches, we can achieve up to 73% reduction in WCRT estimate in our experiment.

Currently, we assume that all cache lines evicted in the preemption will be reloaded by the preempted task after the preemption. However, only a subset of cache lines evicted during the preemption will be accessed again. By applying this fact, we can expect a even tighter estimate of preemption-related cache reloading cost. We will continue our work in this aspect in future.

	Cache Penalty (cycles)			
Task	10	20	30	40
OFDM	6.5%	21.6%	55.7%	73%
ED	6.5%	11.9%	16.5%	20.4%

Table 4. Comparison of results

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

[2] 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*, August, pp.116-127, September 2001.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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