In Proceedings of the 15th Real-Time Systems Symposium, 1994.

An Accurate Worst Case Timing Analysis Technique for RISC Processors*

Sung-Soo Lim† Young Hyun Bae† Gyu Tae Jang†
Byung-Do Rhee† Sang Lyul Min† Chang Yun Park‡
Heonshik Shin† Kunsoo Park† Chong Sang Kim†

Abstract

An accurate and safe estimation of a task's worst case execution time (WCET) is crucial for reasoning about the timing properties of real-time systems. In RISC processors, the execution time of a program construct (e.g., a statement) is affected by various factors such as cache hits/misses and pipeline hazards, and these factors impose serious problems in analyzing the WCETs of tasks. To analyze the timing effects of RISC's pipelined execution and cache memory, this paper proposes extensions of the original timing schema [26] where the timing information associated with each program construct is a simple time-bound. We associate with each program construct what we call a WCTA (Worst Case Timing Abstraction), which contains detailed timing information of every execution path that might *be the worst case execution path of the program construct. This extension leads to a revised timing schema that is similar to the original timing schema except that concatenation and pruning operations on WCTAs are newly defined to replace the* `add` *and* `max` *operations on time-bounds in the original timing schema. Our revised timing schema accurately accounts for the timing effects of pipelined execution and cache memory not only within but also across program constructs. This paper also reports on preliminary results of WCET analyses for a pipelined processor. Our results show that up to 50 % tighter WCET bounds can be obtained by using the revised timing schema.*

1 Introduction

In real-time computing systems, tasks have timing requirements (i.e., deadlines) that must be met for correct operation. Thus, it is of utmost importance to guarantee that tasks finish before their deadlines. Various scheduling techniques, both static and dynamic, have been proposed to ensure this guarantee. These scheduling algorithms generally require that the WCET (Worst Case Execution Time) of each task in the system be known a priori. Therefore, it is not surprising that much research has focused on the estimation of the WCETs of tasks.

In a non-pipelined processor without cache memory, it is relatively easy to obtain a tight bound on the WCET of a sequence of instructions. One just has to sum up their individual execution times that are usually given by a table. The WCET of a program can then be calculated by traversing the program's syntax tree bottom-up and applying formulas for calculating the WCETs of various language constructs. However, for RISC processors such a simple analysis may not be appropriate because of their pipelined execution and cache memory. In RISC processors, an instruction's execution time varies widely depending on many factors such as pipeline stalls due to hazards and cache hits/misses. One can still obtain a safe WCET bound of a program by assuming the worst case execution scenario (e.g., each instruction suffers from all kinds of hazards and every memory access results in a cache miss and so on). However, such a pessimistic approach would yield an extremely loose WCET bound resulting in severe under-utilization of machine resources.

Our goal is to predict tight and safe WCET bounds of tasks for RISC processors. Achieving this goal would permit RISC processors to be widely used in real-time systems. Our approach is based on an extension of the *timing schema* [26]. The timing schema is a set of formulas for computing execution time bounds of language constructs. In the original timing schema, the timing information associated with each program construct is a simple time-bound. This choice of timing information facilitates a simple and accurate timing analysis for processors with fixed execution times. However, for RISC processors, such timing information is not sufficient to accurately account for timing variations resulting from pipelined execution and cache memory.

This paper proposes extensions of the original timing schema to rectify the above problem. We associate with each program construct what we call a WCTA (Worst Case Timing Abstraction). The WCTA of a program construct contains timing information of every execution path that *might* be the worst case execution path of the program

*This work was supported in part by KOSEF (Grant KOSEF-93-01-00-06) and ADD (Contract ADD-91-4-4).

†Dept. of Computer Engineering, Seoul National University, Seoul 151-742, Korea

‡Dept. of Computer Engineering, Chung-Ang University, Seoul 156-756, Korea

construct. Each timing information includes information about the factors that may affect the timing of the succeeding program construct. It also includes the information that is needed to refine the execution time of the program construct when the timing information of the preceding program construct becomes available at a later stage of WCET analysis. This extension leads to a revised timing schema that accurately accounts for the timing variation resulting from history sensitive nature of pipelined execution and cache memory.

The proposed approach has the following advantages. First, the approach makes possible an accurate analysis of combined timing effects of pipelined execution and cache memory, which was not possible in previous approaches. Second, the timing analysis using the proposed approach is more accurate than that of any other approach we are aware of. Third, the proposed approach is applicable to most RISC processors since it does not make any machine specific assumption. Finally, the proposed approach is extensible in that its general rule can be used to model the timing behavior of other machine features. For example, its underlying general rule can be used to model the timing variation due to TLB and write buffers.

This paper is organized as follows. In Section 2, we survey the related work. Section 3 explains the problems in accurately estimating the WCETs of tasks in pipelined processors and presents our analysis method. In Section 4, we describe an accurate timing analysis technique for instruction cache memory and explain how this technique can be combined with the pipeline timing analysis technique given in Section 3. Section 5 identifies the differences between the WCET analysis of instruction caches and that of data caches, and explains how we address the issues resulting from these differences. In Section 6, we report on preliminary results of WCET analyses for a pipelined processor. Finally, we conclude this paper in Section 7.

2 Related work

A timing prediction method for real-time systems should be able to give safe and accurate WCET bounds of tasks. Measurement-based and analytical techniques have been used to obtain such bounds. Measurement-based techniques are, in many cases, inadequate to produce a timing estimation for real-time systems since their predictions are usually not guaranteed, or enormous cost is needed. Because of these limitations of the measurement-based approaches, analytical approaches are becoming more popular. There have been several recent studies about this issue [4, 8, 9, 18, 19, 20, 22, 23, 24, 27, 28]. In many of these studies, the assumed machine model is a simple non-pipelined processor without cache memory [18, 22, 23, 27]. Thus the timing effects of pipelined execution and cache memory are not taken into account.

2.1 Timing analysis of pipelined execution

The timing effects of pipelined execution have been recently studied by Harmon, Baker, and Whalley [9], Harcourt, Mauney, and Cook [8], Narasimhan and Nilsen [20], and Choi, Lee, and Kang [4]. In these studies, the execution time of a sequence of instructions is estimated by modeling a pipelined processor as a set of resources and representing each instruction as a process that acquires and consumes a subset of resources in time. In order to mechanize the process of calculating the execution time, they use various techniques: pattern matching [9], SCCS (Synchronous Calculus of Communicating Systems) [8], retargetable pipeline simulation [20], and ACSR (Algebra of Communicating Shared Resources) [4]. Although these approaches have the advantages of being formal and machine independent, their applications are currently limited to calculating the execution time of a sequence of instructions or a *given* sequence of basic blocks[1]. Therefore, they rely on ad hoc methods to calculate the WCETs of programs.

The pipeline timing analysis technique by Zhang, Burns and Nicholson [28] can mechanically calculate the WCETs of programs for a pipelined processor. Their analysis technique is based on a mathematical model of the pipelined Intel 80C188 processor. This model takes into account the overlap between instruction execution and opcode prefetching in 80C188. In their approach, the WCET of each basic block in a program is individually calculated based on the mathematical model. The WCET of the program is then calculated using the WCETs of the constituent basic blocks and timing formulas for calculating the WCETs of various language constructs.

Although this approach represents a significant progress over the previous schemes that did not account for the timing effects of pipelined execution, it still suffers from two inefficiencies. First, the pipelining effects across basic blocks are not accurately accounted for. In general, due to data dependencies and resource conflicts with the execution pipeline, a basic block's execution time will differ depending on what the surrounding basic blocks are. However, since it is required in their approach that the WCET of each basic block be independently calculated, it appears that they make the worst case assumption on the preceding basic block (e.g., the last instruction of every basic block that can precede the basic block being analyzed needs data memory access, which prevents the opcode prefetching of the first instruction of the basic block being analyzed). This assumption is reasonable for their target processor since its pipeline has only two stages. However, completely ignoring pipelining effects across basic blocks may yield a very loose WCET estimation for more deeply pipelined processors as we will see in Section 6. Second, although

[1] A *basic block* is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [1].

their mathematical model is very effective for the Intel 80C188 processor, it appears that the model is not general enough to be applied to other pipelined processors. This is due to many machine specific assumptions made in the model that are difficult to be generalized to other pipelined processors.

2.2 Timing analysis of cache memory

Cache memories have been widely used to bridge the speed gap between processor and main memory. However, designers of hard real-time systems are wary of using caches in their systems since the performance of caches is considered to be unpredictable. This unpredictable performance of caches stems from the following two sources: inter-task interference and intra-task interference. Inter-task interference is caused by task preemption. When a task is preempted, most of its cache blocks[2] are displaced by the newly scheduled task and the tasks scheduled thereafter. When the preempted task resumes execution, it makes references to the previously displaced blocks and experiences a burst of cache misses. This type of cache misses cannot be avoided in real-time systems with preemptive scheduling of tasks and results in a wide variation in task execution times. This execution time variation can be eliminated by partitioning the cache and dedicating one or more partitions to each real-time task [14, 15]. This cache partitioning approach eliminates the cache unpredictability caused by task preemption. However, it still suffers from the cache unpredictability caused by intra-task interference that will be explained next.

Intra-task interference in caches occurs when more than one memory block of the same task compete with each other for the same cache block. This interference results in two types of cache misses: *capacity* misses and *conflict* misses [11]. Capacity misses are due to finite cache size. Conflict misses, on the other hand, are caused by a limited set associativity. These types of cache misses cannot be avoided if the cache has a limited size and/or set associativity.

Among the analytical WCET prediction schemes that we are aware of, only three schemes take into account the timing variation resulting from intra-task cache interference (two for instruction caches [19, 21] and one for data caches [24]). The *static cache simulation* approach which statically predicts hits or misses of instruction references is due to Mueller, Whalley and Harmon [19]. In this approach, instructions are classified into the following four categories based on a data flow analysis:

- *always-hit*: The instruction is always in the cache.
- *always-miss*: The instruction is never in the cache.
- *first-miss*: The first reference to the instruction misses in the cache. However, all the subsequent references hit in the cache.
- *conflict*: The instruction may or may not be in the cache.

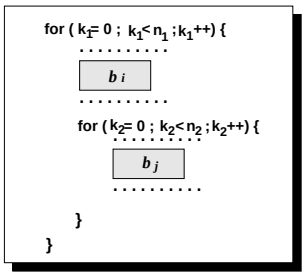


Figure 1: Sample **C** program fragment

This approach is simple but has a number of limitations. One limitation is that the analysis is too conservative. As an example, consider the program fragment given in Figure 1. Assume that both of the instruction memory blocks b_i and b_j are mapped to the same cache block and that no other instruction memory blocks are mapped to that cache block. During the actual execution, among the $n_1 \times n_2$ references to b_j, $n_1 \times n_2 - n_1$ references are cache hits and only n_1 references are cache misses. However, by being classified as *conflict*, all the $n_1 \times n_2$ references to b_j are treated as cache misses in this approach. Another limitation of this approach is that the approach has not addressed the issues of locating the worst case execution path and of calculating the WCET, which are critical in scheduling tasks in real-time systems.

In [21], Niehaus et al. discuss the potential benefits of identifying instruction references corresponding to *always-hit* and *first-miss* in the static cache simulation approach. However, as it is noted in [19], their analysis is rather abstract and no general method to analyze the worst case timing behavior of programs in the presence of instruction caches is given.

Rawat performs a static analysis for data caches [24]. His approach is similar to the graph coloring approach to register allocation [5]. In his analysis, first, live ranges of variables and those of memory blocks are computed[3]. Second, an interference graph is constructed for each cache block. An edge in the interference graph connects two memory blocks if they are mapped to the same cache block and their live ranges overlap with each other. Third, live ranges of memory blocks are split until they do not overlap with each other. If a live range of a memory block does not overlap with that of any other memory block, the memory block never gets replaced from the cache during execution within the live range. Therefore, the number of cache misses due to a memory block can be calculated from the frequency counts of its live ranges (i.e., how many

[2] A *block* is the minimum unit of information that can be either present or not present in the cache-main memory hierarchy [10].

[3] A live range of a variable (memory block) is a set of basic blocks during whose execution the variable (memory block) potentially resides in the cache [24].

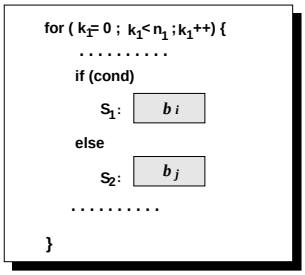


Figure 2: Another sample **C** program fragment

times the program control flows into the live ranges). Finally, the number of total data cache misses is estimated by summing up the frequencies of all the live ranges of all the memory blocks used in the program.

Although this analysis method represents a significant progress over the analysis methods in which every data reference is treated as a cache miss, it still suffers from the following three limitations. First, the analysis does not allow function calls and global variables, which severely limits its applicability. Second, the analysis leads to an overestimation of data cache misses by assuming that all possible execution paths are taken during program execution. To see this point, consider the program fragment in Figure 2. Assume that S_1 and S_2 reference data memory blocks b_i and b_j, respectively and that these two blocks are mapped to the same cache block. Further assume that the execution time of S_1 is much longer than that of S_2. Under these assumptions, the worst case execution scenario of this program fragment is to repeatedly execute S_1 within the loop. In this worst case scenario, only the first access to b_i will miss in the cache and all the subsequent accesses within the loop will hit in the cache. However, since the live range of b_i overlaps with that of b_j, all the accesses to b_i are assumed to miss in the cache. This results in an excessive overestimation of the data cache misses due to accesses to b_i. The third limitation of the approach is that it has not addressed the issues of locating the worst case execution path and of calculating the WCET, again limiting its applicability.

3 Pipelining effects

In pipelined processors, various execution steps of instructions are simultaneously overlapped. Because of this overlapped execution, an instruction's execution time will differ depending on what the surrounding instructions are. However, this timing variation cannot be accurately accounted for in the original timing schema, since the timing information associated with each program construct is a simple time-bound. In this section, we extend the timing schema to rectify this problem.

In our extended timing schema, the timing information of each program construct is a set of reservation tables

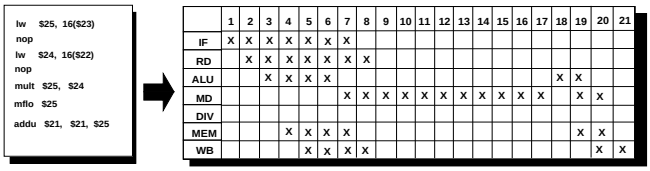


Figure 3: Sample MIPS assembly code and the corresponding reservation table

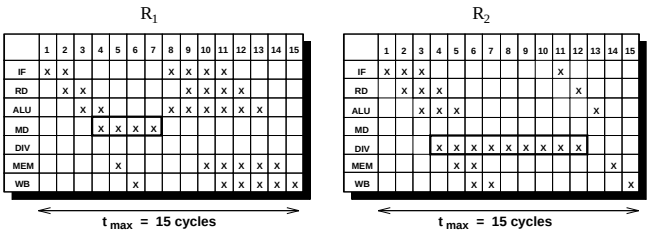


Figure 4: Two reservation tables with equal t_{max}

rather than a time-bound. The reservation table was originally proposed to describe and analyze the activities within a pipeline [16]. In a reservation table, the vertical dimension represents the stages in the pipeline and the horizontal dimension represents time. Figure 3 shows a simple basic block in MIPS assembly language and the corresponding reservation table. In the figure, each **x** in the reservation table specifies the use of the corresponding stage for the indicated time slot. In the proposed approach, we analyze the timing interactions among instructions within a basic block by building its reservation table.

A program construct such as an **if** statement may have more than one execution path. Moreover, it is not always possible in pipelined processors to determine which one of the execution paths is the worst case execution path by analyzing the program construct alone. As an example, suppose that an **if** statement has two execution paths corresponding to the two reservation tables shown in Figure 4. Here the worst case execution path depends on the instructions in the surrounding program constructs. For example, if one of the instructions near the end of the preceding program construct uses the **MD** stage, the execution path corresponding to $\mathbf{R}_1$ will become the worst case execution path. On the other hand, if there exists an instruction using the **DIV** stage instead, the execution path corresponding to $\mathbf{R}_2$ will become the worst case execution path. Therefore, we should keep both reservation tables until the timing information of the surrounding program constructs is known.

Figure 5 shows the data structure for a reservation table used in our approach both in textual and graphical forms. In the data structure, t_{max} is the worst case execution time of the reservation table, which is determined by the number of columns in the reservation table. In implementation, not all the columns in the reservation table

```
struct pipeline_timing_information {
    time t_max;
    reservation_table head[δ_head];
    reservation_table tail[δ_tail];
}
```

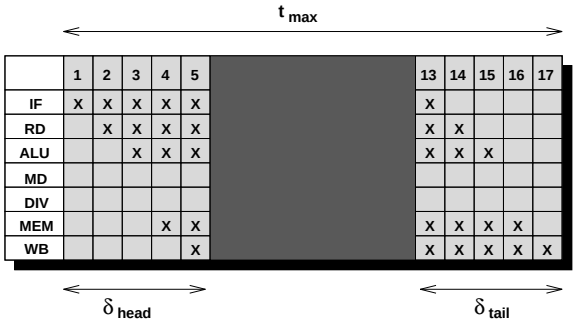


Figure 5: Reservation table data structure

are maintained. Instead, we maintain only a first few (i.e., δ_{head}) columns whose timing behavior may be affected by the instructions in the preceding program construct and a last few (i.e., δ_{tail}) columns that may affect the timing behavior of the succeeding program construct. Regardless of how small or how large δ_{head} and δ_{tail} are, a reservation table represented in this way is safe in that the timing behavior deduced from this approximate representation does not underestimate the actual WCET. However, the larger δ_{head} and δ_{tail} are, the tighter the resulting WCET estimation is, as we will see later. $\delta_{head} = \delta_{tail} = \infty$ corresponds to the case where the full reservation table is maintained.

As explained earlier, we associate with each program construct a set of reservation tables where each reservation table contains the timing information of an execution path that *might* be the worst case execution path of the program construct. We call this set the WCTA (Worst Case Timing Abstraction) of the program construct. This WCTA corresponds to the time-bound in the original timing schema.

With this framework, the timing schema can be extended so that the timing interactions among instructions not only within but also across program constructs can be accurately accounted for. In the extended timing schema, the timing formula of a sequential statement **S: S$_1$; S$_2$** is given by

$$W(S) = W(S_1) \bigoplus W(S_2)$$

where $W(S)$, $W(S_1)$ and $W(S_2)$ are the WCTAs of S, S_1 and S_2, respectively. The operation $\bigoplus$ between two WCTAs is defined as

$$W_1 \bigoplus W_2 = \{w_1 \oplus w_2 | w_1 \in W_1, w_2 \in W_2\}$$

where w_1 and w_2 are reservation tables and the $\oplus$ operation concatenates two reservation tables giving another

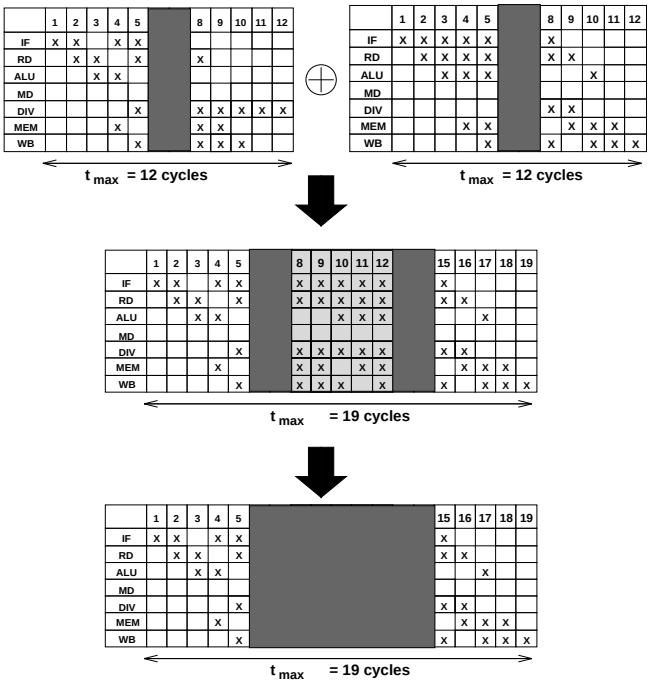


Figure 6: Example application of $\oplus$ operation

reservation table. This concatenation operation models the pipelined execution of a sequence of instructions followed by another sequence of instructions. The semantics of this operation for a target processor can be deduced from its data book. Figure 6 shows an application of the $\oplus$ operation. From the figure, we can note that as more columns are maintained in `head` and `tail`, more overlap between `head` and `tail` of adjacent program structures can be modeled and, therefore, a tighter WCET estimation can be obtained.

The above timing formula effectively enumerates all the possible candidates for the worst case execution path in $S_1; S_2$. During each instantiation of this timing formula, a check is made to see whether the resulting WCTA can be pruned. An element in a WCTA can be eliminated completely from the WCTA if we can guarantee that the element's WCET assuming the worst case scenario for the element on the surrounding program constructs is shorter than the WCET of some other element in the same WCTA assuming the best case scenario for this element on the surrounding program constructs. This pruning condition can be more formally specified as follows:

> A reservation table w in a WCTA W can be pruned without affecting the prediction for the worst case timing behavior of W if
>
> $$\exists w' \in W, \;\; w.t_{max} < w'.t_{max} - \delta_{head} - \delta_{tail}$$

In this condition, $w.t_{max}$ is w's execution time when we assume the worst case scenario for w on the surrounding program constructs (i.e., when no part of w's `head` and `tail`

is overlapped with the surrounding program constructs). On the other hand, $w'.t_{max} - \delta_{head} - \delta_{tail}$ is the execution time of w' when we assume the best case scenario for w' on the surrounding program constructs (i.e., when its `head` is completely overlapped with the `tail` of the preceding program construct and its `tail` is completely overlapped with the `head` of the succeeding program construct).

The timing formula of an **if** statement **S: if (exp) then S_1 else S_2** is given by

$$\begin{aligned} W(S) &= (W(exp) \bigoplus W(S_1)) \bigcup (W(exp) \bigoplus W(S_2)) \\ &= W(exp) \bigoplus (W(S_1) \bigcup W(S_2)) \end{aligned}$$

where $W(S)$, $W(exp)$, $W(S_1)$ and $W(S_2)$ are the WCTAs of S, exp, S_1 and S_2, respectively and $\bigcup$ is the set union operation. As in the previous timing formula, pruning is performed during each instantiation of this timing formula.

Function calls are processed like sequential statements. In our approach, functions are processed in a reverse topological order in the call graph[4] since the WCTA of a function should be calculated before any of the functions that call it is processed.

Finally, the timing formula of a loop statement **S: while (exp) S_1** is given by

$$W(S) = (\bigoplus_{i=1}^{N} (W(exp) \bigoplus W(S_1))) \bigoplus W(exp)$$

where N is a loop bound that is provided by some external means (e.g., from user input). This timing formula effectively enumerates all the possible candidates for the worst case execution scenario of the loop statement. This approach is exact but is computationally intractable for a large N. In the following, we will give an alternative formulation of the loop timing analysis.

Loop timing Analysis Assume that the number of elements in $W(exp) \bigoplus W(S_1)$ is p corresponding to the set of execution paths $P = \{p_1, p_2, \ldots, p_p\}$. Consider an execution scenario of ℓ loop iterations where p_i is executed in the first iteration and p_j is executed in the last iteration. There are $|P|^{\ell-2}$ possible such scenarios. Among them let wp_{ij}^{ℓ} be the execution scenario with the longest execution time and $wcta(wp_{ij}^{\ell})$ be its WCTA. Note that wp_{ij}^{ℓ} is unique since all of the $|P|^{\ell-2}$ possible scenarios share the same `head` inherited from p_i and the same `tail` inherited from p_j and, therefore, the surrounding program constructs cannot make any difference in the way they affect the timing of the scenarios. With this framework, the WCTA of the loop statement is obviously contained in $(\bigcup_{p_i,p_j \in P} wcta(wp_{ij}^{N})) \bigoplus W(exp)$.

The calculation of $wcta(wp_{ij}^{\ell})$ can proceed as follows: First, since wp_{ij}^{ℓ} begins with p_i's execution, $wcta(wp_{ij}^{\ell})$ inherits p_i's `head`. Likewise, since wp_{ij}^{ℓ} ends with p_j's execution, $wcta(wp_{ij}^{\ell})$ inherits p_j's `tail`. Second, assuming that $wp_{ij}^{\ell} = p_i, p_{i_1}, p_{i_2}, \ldots, p_{i_{\ell-2}}, p_j$, $wcta(wp_{ij}^{\ell})$'s t_{max} is given by the following equation:

$$\begin{aligned} wcta(wp_{ij}^{\ell}).t_{max} = \; & p_i.t_{max} + (p_{i_1}.t|p_i) + (p_{i_2}.t|p_{i_1}) + \cdots \\ & + (p_{i_{\ell-3}}.t|p_{i_{\ell-4}})(p_{i_{\ell-2}}.t|p_{i_{\ell-3}}) + (p_j.t|p_{i_{\ell-2}}) \end{aligned}$$

where $(p_{k_2}.t|p_{k_1}) = (p_{k_1} \oplus p_{k_2}).t_{max} - p_{k_1}.t_{max}$ denoting p_{k_2}'s execution time when its execution is immediately preceded by p_{k_1}'s execution (cf. Figure 7).

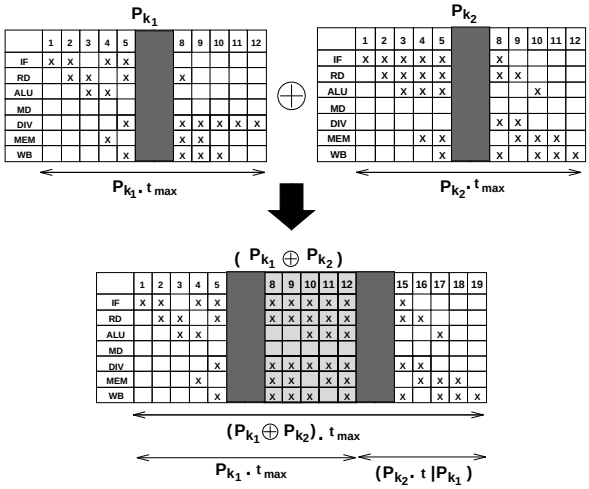


Figure 7: Calculation of $(p_{k_1}.t|p_{k_2})$

Let $D_{\ell-1,i,j}$ be $(p_{i_1}.t|p_i) + (p_{i_2}.t|p_{i_1}) + \ldots + (p_{i_{\ell-3}}.t|p_{i_{\ell-4}}) + (p_{i_{\ell-2}}.t|p_{i_{\ell-3}}) + (p_j.t|p_{i_{\ell-2}})$. Consider a complete weighted directed graph $G = (P, A)$ where $P = \{p_1, p_2, \ldots, p_p\}$ and each weight w_{mn} is $(p_n.t|p_m)$. With this setting, $D_{\ell,i,j}$ is the maximum weight of a path (not necessarily simple) from p_i to p_j in G containing exactly ℓ arcs. This problem can be solved using the following equations.

$$\begin{aligned} D_{0,i,j} &= \begin{cases} -\infty & \text{if } p_i \neq p_j \\ 0 & \text{otherwise} \end{cases} \\ D_{\ell,i,j} &= \max_{p_k \in P} \{D_{\ell-1,i,k} + w_{kj}\} \end{aligned}$$

Computation of $D_{\ell,i,j}$ for all $p_i, p_j \in P$, $\ell = 0$ to $N-1$ using this dynamic programming technique takes $O(N \times |P|^3)$ time. For a large N, this time complexity is still unacceptable. In the following, we will describe a technique that gives very tight upper bounds for $D_{\ell,i,j}$'s. The technique is based on the calculation of the maximum cycle mean of G.

The maximum cycle mean of a weighted directed graph G is $m = max_C\ m(C)$ where $m(C)$ is the mean weight of C and C ranges over all directed cycles in G. The maximum cycle mean can be calculated in $O(|P| \times |A|)$ using the algorithm due to Karp [13]. Let m be the maximum cycle mean of G, then $D_{\ell,i,j}$ can be safely approximated by

[4] A *call graph* contains the information on how functions call each other [6]. For example, if f calls g, then an arc connects f's vertex to g's in their call graph.

$D'_{\ell,i,j} = \ell \times m + (m - w_{ji})$. We prove this in the following proposition.

Proposition 1 *If $D_{\ell,i,j}$ is the maximum weight of a path (not necessarily simple) from p_i to p_j containing exactly ℓ arcs in a complete weighted directed graph $G = (P, A)$ and m is the maximum cycle mean of G, then $D_{\ell,i,j} \leq D'_{\ell,i,j} = \ell \times m + (m - w_{ji})$ for all $p_i, p_j \in P$ and $\ell > 0$.*

Proof. Assume for the sake of contradiction that $D_{\ell,i,j}$ is greater than $\ell \times m + (m - w_{ji})$. Then we can construct a cycle containing $\ell + 1$ arcs by adding the arc from p_j to p_i to the path from which $D_{\ell,i,j}$ is calculated. The arc should exist since G is a complete graph. The resulting cycle has a mean weight greater than m since $\frac{D_{\ell,i,j}+w_{ji}}{\ell+1} > \frac{\ell\times m+(m-w_{ji})+w_{ji}}{\ell+1} = m$. This implies an existence of a cycle in G whose mean weight is greater than m. This contradicts our hypothesis that m is the maximum cycle mean of G and thus $D_{\ell,i,j} \leq \ell \times m + (m - w_{ji})$. □

Moreover, it can be shown that $D'_{\ell,i,j} - D_{\ell,i,j}$, which indicates the looseness of the approximation, is bounded above by $3 \times (m - w_{min})$ where w_{min} is the minimum weight of an arc in A [17]. We can expect this bound to be very tight since $m \simeq w_{min}$. (Remember that P consists of the paths in $W(exp) \bigoplus W(S_1)$ that cannot be pruned by each other.)

Once $D'_{\ell-1,i,j}$ (or $D_{\ell-1,i,j}$ if possible) is calculated, $wcta(wp^{\ell}_{ij})$ is given by

$$wcta(wp^{\ell}_{ij}) = (p_i.tmax + D'_{\ell-1,i,j} \ (or\ D_{\ell-1,i,j}),\ p_i.head,\ p_j.tail)$$

Finally, the WCTA of the loop statement is given by

$$W(S) = (\bigcup_{p_i,p_j \in P} wcta(wp^{N}_{ij})) \bigoplus W(exp)$$

Assuming that we use $D'_{\ell,i,j}$, the calculation of the WCTA of the loop statement in this way takes $O(|P| \times |A|)$ time.

Interference Up until now, we assume that tasks execute without preemption. However, in real systems, tasks may be preempted for various reasons: preemptive scheduling, external interrupts, resource contention, and so on. Each task preemption accompanies a task switch. The task switch delay introduced by pipelined execution in addition to that incurred in non-pipelined processors is bounded by the maximum number of cycles that the effects of an instruction remain in the pipeline (in MIPS R3000 it is 36 cycles in the case of the **div** instruction).

4 Instruction caching effects

For a processor with an instruction cache, the execution time of a program construct will differ depending on which execution path has been taken prior to the program construct. This results from history sensitive nature of the instruction cache. As an example, consider a program construct that accesses instruction blocks[5] (b_2, b_3, b_2, b_4) in the sequence given (cf. Figure 8). Assume that the instruction cache has only two blocks and is direct-mapped. In a direct mapped cache, each instruction block can be placed exactly in one cache block whose index is given by *instruction block number modulo number of blocks in the cache.*

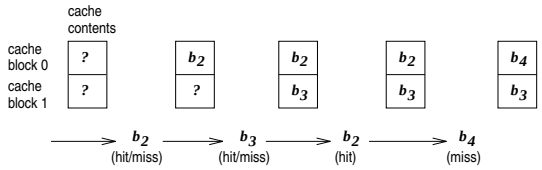


Figure 8: Sample instruction block references from a program construct

In this example, the second reference to b_2 will always hit in the cache because the first reference to b_2 will bring b_2 into the cache and this cache block will not be replaced in-between. On the other hand, the reference to b_4 will always miss in the cache even when b_4 was previously in the cache prior to this program construct because the first reference to b_2 will replace b_4's copy in the cache in-between. (Remember that b_2 and b_4 are mapped to the same cache block in the assumed cache configuration.) Unlike the above two references whose hits or misses are guaranteed, the hit or miss of the first reference to b_2 depends on the cache contents immediately before executing this program construct and so does the hit or miss of the reference to b_3. The hits or misses of these references will affect the (worst case) execution time of this program construct. Moreover, the cache contents after this program construct will, in turn, affect the execution time of the succeeding program construct. These timing variations again cannot accurately be represented by a simple time-bound of the original timing schema.

This situation is similar to the situation for pipelined execution in the previous section and, therefore, we adopt the same strategy; we simply extend the timing information of elements in a WCTA leaving the timing formulas intact. Each element in a WCTA now has two sets of instruction block addresses in addition to t_{max}, `head`, and `tail` used for the timing analysis of pipelined execution. Figure 9 gives the data structure for an element in a WCTA in the new setting where n_{block} denotes the number of blocks in the cache.

In the given data structure, the first set of instruction block addresses (i.e., `first_reference`) maintains the instruction block addresses of the references whose hits or misses depend on the cache contents prior to the program

[5] We regard a sequence of instruction references to an instruction block as a single reference to the instruction block without any loss of accuracy in analysis.

```
struct pipeline_cache_timing_information {
    time t_max;
    reservation_table head[δ_head];
    reservation_table tail[δ_tail];
    block_address first_reference[n_block];
    block_address last_reference[n_block];
};
```

Figure 9: Structure of an element in a WCTA

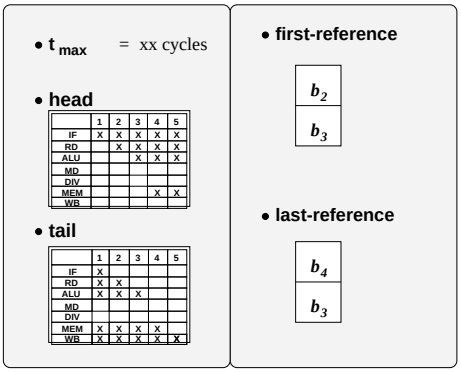


Figure 10: Contents of the WCTA element corresponding to the example in Figure 8

construct. In other words, this set maintains for each cache block the instruction block address of the first reference to the cache block. The second set (i.e., `last_reference`) maintains the addresses of the instruction blocks that will remain in the cache after the execution of the program construct. In other words, this set maintains for each cache block the instruction block address of the last reference to the cache block. This is the cache contents that will determine the hits or misses of the instruction block references in the `first_reference` of the succeeding program construct. In t_{max}, we accurately account for the "guaranteed" hits and misses such as the second reference to b_2 and the reference to b_4 in the previous example. However, the instruction block references whose hits or misses are not known (i.e., those in `first_reference`) are conservatively assumed to miss in the cache in the initial estimate of t_{max}. This initial estimate is later refined as the information on the hits or misses of those references becomes available in a later stage of analysis. Figure 10 shows the timing information maintained for the program construct given in the previous example.

With this extension, the timing formula of **S: S$_1$; S$_2$** is given by

$$W(S) = W(S_1) \bigoplus W(S_2)$$

This timing formula is structurally identical to the one given in the previous section for a sequential statement.

```
1   struct pipeline_cache_timing_information
2   concatenate(struct pipeline_cache_timing_information w1,
3               struct pipeline_cache_timing_information w2)
4   {   struct pipeline_cache_timing_information w3;
5       int n_hits, i;
6
7       n_hits = 0;
8       for (i = 0, i < n_block; i++) {
9           if (w1.first_reference[i] == NULL)
10              w3.first_reference[i] = w2.first_reference[i];
11          else
12              w3.first_reference[i] = w1.first_reference[i];
13          if (w2.last_reference[i] == NULL)
14              w3.last_reference[i] = w1.last_reference[i];
15          else
16              w3.last_reference[i] = w2.last_reference[i];
17          if (w1.last_reference[i] == w2.first_reference[i])
18              n_hits++
19      }
20      w3.head = w1.head
21      w3.tail = w2.tail
22      w3.t_max = ((w1.t_max, w1.head, w1.tail) ⊕_pipeline
23                  (w2.t_max, w2.head, w2.tail)).t_max
24                  - n_hits * t_miss penalty;
25      return w3;
26  }
```

Figure 11: Semantics of the $\oplus$ operation

The differences are in the structure of elements in the WCTAs and in the semantics of the $\oplus$ operation. The revised semantics of the $\oplus$ operation is procedurally defined in Figure 11.

The function `concatenate` given in the figure concatenates two input elements w_1 and w_2 and puts the result into w_3, thus implementing the $\oplus$ operation. In lines 9-12 of function `concatenate`, w_3 inherits w_1's `first_reference` if the corresponding cache block is accessed in w_1. If the cache block is not accessed in w_1, the first reference to the cache block in $w_1 \oplus w_2$ is from w_2. Therefore, in this case, w_3 inherits w_2's `first_reference`. Likewise, in lines 13-16, w_3 inherits w_2's `last_reference` if the corresponding cache block is accessed in w_2 or w_1's `last_reference` otherwise. By comparing w_2's `first_reference` with w_1's `last_reference`, lines 17-18 determine how many of the memory references in w_2's `first_reference` will hit in the cache when w_2's execution is preceded by w_1's execution. These cache hits are used to refine w_3's t_{max}. (Remember that all the memory references in w_2's `first_reference` were previously assumed to miss in the cache in the initial estimate of w_2's t_{max}.) In lines 20-21, w_3 inherits w_1's `head` and w_2's `tail`. Lines 22-24 calculate w_3's t_{max} taking into account the pipelined execution across w_1 and w_2 and the cache hits determined in lines 17-18. In this calculation, the $\oplus_{pipeline}$ operation is the $\oplus$ operation defined in the previous section for the timing analysis of pipelined execution and $t_{miss\ penalty}$ is the time needed to service a cache miss.

As before, an element in a WCTA can safely be eliminated (i.e., pruned) from the WCTA if we can guarantee that the element's WCET is always shorter than that of some other element in the same WCTA regardless of what the surrounding program constructs are. This condition for pruning is procedurally specified in Figure 12. The function `prune` given in the figure checks whether either

```
1   struct pipeline_cache_timing_information
2   prune(struct pipeline_cache_timing_information w1,
3         struct pipeline_cache_timing_information w2)
4   {   int n_diff, i;
5
6       n_diff = 0;
7       for (i = 0, i < n_block; i++) {
8          if (w1.first_reference[i] != w2.first_reference[i])
9             n_diff++;
10         if (w1.last_reference[i] != w2.last_reference[i])
11            n_diff++;
12      }
13      if (w2.t_max < w1.t_max - n_diff * t_miss penalty - δ_head - δ_tail)
14         return w2;
15      else
16         if (w1.t_max < w2.t_max - n_diff * t_miss penalty - δ_head - δ_tail)
17            return w1;
18         else
19            return NULL;
20  }
```

Figure 12: Semantics of pruning operation

one of the two execution paths corresponding to two input elements w_1 and w_2 can be pruned and returns the pruned element if the pruning is successful and null if neither of them can be pruned.

In the function `prune`, lines 6-12 determine how many entries in w_1's `first_reference` and `last_reference` are different from the corresponding entries in w_2's `first_reference` and `last_reference`. This bounds the difference between the cache memory related execution time variation of w_1 and that of w_2. Line 13 checks whether w_2 can be pruned by w_1. Pruning of w_2 by w_1 can be made if w_2's execution time assuming the worst case scenario for w_2 is shorter than w_1's execution time assuming its best case scenario. Likewise, line 16 checks whether w_1 can be pruned by w_2.

Again as before, the timing formula of **S: if (exp) then** S_1 **else** S_2 is given by

$$\begin{aligned} W(S) &= (W(exp) \bigoplus W(S_1)) \bigcup (W(exp) \bigoplus W(S_2)) \\ &= W(exp) \bigoplus (W(S_1) \bigcup W(S_2)) \end{aligned}$$

As in the previous section, the problem of calculating $W(S)$ for a loop statement **S: while (exp)** $\mathbf{S}_1$ can be translated into a graph theoretic problem. Here $wcta(wp_{ij}^{\ell})$ is given by

$$(p_i.t_{max} + D'_{\ell-1,i,j}, p_i.head, p_j.tail, p_i.first\_reference, p_j.last\_reference)$$

After calculating $wcta(wp_{ij}^{N})$ for all p_i, $p_j \in P$, $W(S)$ can be computed as follows:

$$\Big(\bigcup_{p_i,p_j \in P} wcta(wp_{ij}^{N}) \Big) \bigoplus W(exp)$$

Our loop timing analysis assumes that each loop iteration benefits only from the immediately preceding loop iteration. This is because in the calculation of $(p_i.t|p_j)$, we consider only the execution time reduction of p_i resulting from being preceded by p_j. This assumption holds in the case of pipelined execution since the execution time of an iteration's `head` is affected only by the `tail` of the immediately preceding iteration. This results from our assumption that all the stages of the columns between `head` and `tail` are used in our representation of reservation tables. These columns prevent the pipelining effects from being propagated more than one iteration. In the case of cache memory, however, the assumption that each loop iteration benefits only from the immediately preceding loop iteration does not hold in general. For example, an instruction memory reference may hit to a cache block that was loaded into the cache in an iteration other than the immediately preceding one. Nevertheless, since the assumption is conservative, the resulting worst case timing analysis is safe in the sense that it does not underestimate the WCET of the loop statement. The degradation of accuracy resulting from this conservative assumption can be reduced by considering a sequence of k ($k > 1$) iterations at the same time in the analysis rather than just one iteration [17]. In this case, each vertex represents an execution of a sequence of k iterations and w_{ij} is the execution time of $sequence_j$ when its execution is immediately preceded by an execution of $sequence_i$. This analysis corresponds to the analysis of the loop unrolled k times and trades increased analysis complexity for more accurate WCTA calculation.

Set associative caches Up until now we have considered only the simplest cache organization called direct mapped cache in which each instruction block can be placed exactly in one cache block. In a more general cache organization called n-way set associative cache, each instruction block can be placed in any one of the n blocks in the mapped set[6]. Set associative caches need a policy that decides which block to replace among the blocks in a set to make a room for a block fetched on a cache miss. The LRU (Least Recently Used) policy is typically used for that purpose. Once this *replacement* policy is given (assuming that it is not random), it is straightforward to implement $\oplus$ and `prune` operations needed in our analysis.

Interference Unless the cache is partitioned and each task has dedicated cache partitions, the execution times and thus the WCETs of tasks are affected by task preemption. In systems that allow task preemption, when a preempted task resumes execution, it has to reload the cache blocks that have been displaced by other tasks between the time it was preempted and the time it resumed execution. The additional delay due to this cache reload in the worst case is bounded by the time needed to completely fill the cache or by the time to reload all of its instruction and data memory blocks into the cache, whichever shorter [3]. It is possible to obtain a tighter bound on this additional delay by taking into account the *usefulness* of cache blocks. A cache block is *useful* at an execution point if the cache block is accessed at least once after the point before

[6] In a set associative cache, the index of the mapped set is given by *instruction block number modulo number of sets in the cache.*

being replaced. Obviously the worst case preemption scenario for a task is to be preempted at the execution point with the maximum number of useful cache blocks and the corresponding cache-related delay is bounded by the time needed to reload all the useful cache blocks at that point.

5 Data caching effects

The timing analysis of data caches is analogous to that of instruction caches. However, the former differs from the latter in several important ways. First, unlike instruction references, the actual addresses of some data references are not known at compile-time. This complicates the timing analysis of data caches since the calculation of `first_reference` and `last_reference`, which is the most important aspect of our cache timing analysis, depends on the assumption that the actual address of every memory reference is known at compile-time. This complication, however, can be avoided completely, if a simple hardware support in the form of one bit in each load/store instruction is available. This bit, called *allocate* bit, decides whether the memory block fetched on a miss will be loaded into the cache. For a data reference whose address cannot be determined at compile-time, the allocate bit is set to zero, which prevents the memory block fetched on a miss from being loaded into the cache. For other references, this bit is set to one allowing the fetched block to be loaded into the cache. With this hardware support, the worst case timing analysis of data caches can be performed very much like that of instruction caches by treating the references whose addresses are not known at compile-time as misses and completely ignoring them in the calculation of `first_reference` and `last_reference`. Even when such hardware support is not available, the worst case timing analysis of data caches is still possible although it is with a consequential loss of accuracy [2].

The second difference stems from accesses to local variables. In general, data area for local variables of a function, called the *activation record* of the function, is pushed and poped on a runtime stack as the associated function is called and returns. In most **C** language implementations, a specially designated register, called `sp` (Stack Pointer), marks the top of the stack and each local variable is addressed by an offset relative to `sp`. The offsets of local variables are determined at compile-time. However, the value of `sp` of a function differs depending on from where the function is called and so do the actual addresses of the function's local variables. However, unlike the accesses discussed in the previous paragraph, the number of addresses that accesses to a local variable may have is bounded. Therefore, the WCTA of a function can be computed by taking into account the `sp` values the function may have. Such `sp` values can be calculated from the activation record sizes of functions and the call graph.

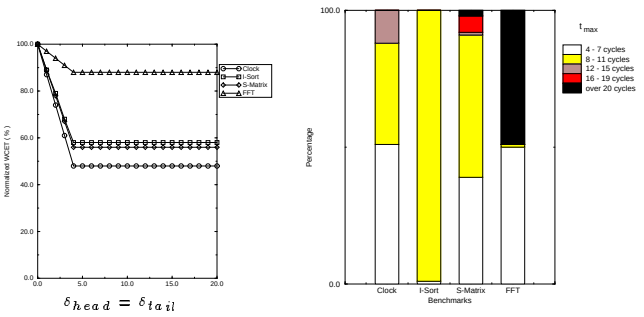
The final difference is due to write accesses. Unlike instruction references, which are read-only, data references may both read from and write to memory. In data caches, one of two policies is used to handle write accesses: *write-through* and *write-back* policies [10]. In the write-through policy, the effects of each write are reflected to both the block in the cache and to the block in main memory. On the other hand, in the write-back policy, the effects are reflected only to the block in the cache and a bit called *dirty* bit is set to indicate the block has been modified. When a block whose dirty bit is set is replaced from the cache, the block's contents is written back to main memory.

The timing analysis of data caches with the write-through policy is relatively simple. One simply has to add delay to each write access to account for the accompanying write access to main memory. However, the timing analysis of data caches with the write-back policy is slightly more complicated. In a write-back cache, a sequence of write accesses to a cached memory block, which we call a *write run*, require only one write-back to main memory. We attribute this write-back overhead (i.e., delay) to the first write in the write run, which we call the *leader* of the write run. With this setting, one has to determine for each write access whether it is a leader to accurately estimate the delay due to write-backs. In some cases, analyses within basic blocks can determine whether a write access is a leader. However, local analyses are not sufficient to determine whether a write access is a leader in every case. If we cannot determine whether a write access is a leader through local analyses, we conservatively assume that the write access is a leader and add write-back delay to t_{max}. However, if a later analysis reveals that the write access is not a leader, we subtract the incorrectly attributed write-back delay from t_{max}. This global analysis can be performed by providing one bit to each block in `first_reference` and `last_reference` and augmenting the $\oplus$ operation [2].

6 Experimental results

In order to assess the effects of the extended timing schema on the accuracy of resultant WCET estimation, we choose a set of four simple programs as our benchmarks and compare their WCET predictions using a timing tool described in [25]. Our timing tool consists of a compiler and a timing analyzer. The compiler is a modified version of an ANSI compiler called **lcc** [7]. This compiler accepts a **C** source program and generates the assembly code along with program structure information. The timing analyzer uses the assembly code and the program structure information along with user-provided information to compute the WCET of the program. In the current implementation, only the pipeline timing analysis is supported. We are currently integrating the timing analysis of instruction and data caches into our timing tool. The machine model of the timing tool is the MIPS R3000 CPU [12].

The four benchmark programs used in the experiment are *Clock*, *FFT*, *I-Sort* and *S-Matrix*. The *Clock* bench-



(a) (b)

Figure 13: Effects of considering pipelined execution across basic blocks

mark is a program used to implement a periodic timer. This program periodically checks 20 linked-listed timers and, if any of them expires, calls the corresponding handler function. The *FFT* benchmark performs the FFT (Fast Fourier Transform) on an array of 100 double precision floating point numbers. The *I-Sort* benchmark sorts an array of 100 integer numbers using an insertion sort algorithm and the *S-Matrix* program multiplies two 10×10 sparse matrices and puts the result into another sparse matrix.

Figure 13-(a) shows the WCET prediction of each of the four benchmarks. In the WCET prediction, only the timing effects of pipelined execution are taken into account and all the instruction and data accesses are assumed to hit in the cache. The results are shown for $0 \leq \delta_{head} = \delta_{tail} \leq 20$ and they are normalized to the case where $\delta_{head} = \delta_{tail} = 0$. The latter corresponds to the case where the pipelining effects across basic blocks are ignored. However, even in this case, the pipelining effects within basic blocks are accurately taken into account.

The results show that considering the pipelining effects across basic blocks can make up to 50 % difference in the WCET prediction. This large difference can be explained by the following two factors. First, the execution times of basic blocks in our benchmark programs are mostly short (cf. Figure 13-(b)). For example, in the *Clock* benchmark, more than 80 % of the basic blocks in the worst case execution path have execution times less than 11 cycles. Second, the assumed machine model (i.e., MIPS R3000) has a relatively large number of stages (5 stages) in its integer execution pipeline. They together compound the prediction inaccuracies in the case of ignoring pipelining effects across basic blocks.

For the *FFT* benchmark, the WCET overestimation resulting from ignoring pipelining effects across basic blocks is not as severe as for the other three benchmarks. This can be explained by a large number of basic blocks found in *FFT* whose execution times are significantly longer than those of the other benchmarks (cf. Figure 13-(b)). These long running basic blocks slightly dilute the importance of considering pipelining effects across basic blocks in the case of the *FFT* benchmark.

The results also show that maintaining about five columns in `head` and `tail` data structures is sufficient to accurately model the pipelined execution across basic blocks. This result is closely related to the number of stages in MIPS R3000's integer execution pipeline.

7 Conclusion

In this paper, we described a technique that aims at accurately estimating the WCETs of tasks for RISC processors. In the proposed technique, two kinds of timing information are associated with each program construct. The first type of information is about the factors that may affect the timing of the succeeding program construct. The second type of information is about the factors that are needed to refine the execution time of the program construct when the first type of the timing information of the preceding program construct becomes available at a later stage of WCET analysis. We rewrote the existing timing schema using these two kinds of timing information so that we can accurately account for the timing variations resulting from history sensitive nature of pipelined execution and cache memory. We also described an optimization that minimizes the overhead of the proposed technique by pruning the timing information associated with an execution path that cannot be a part of the worst case execution path.

The proposed technique has the following advantages. First, the proposed technique makes possible an accurate analysis of combined timing effects of pipelined execution and cache memory, which was not possible in previous approaches. Second, the timing analysis using the proposed technique is more accurate than that of any other technique we are aware of. Third, the proposed technique is applicable to most RISC processors since it does not make any machine specific assumption. Finally, the proposed technique is extensible in that its general rule can be used to model the timing behavior of other machine features. For example, its underlying general rule can be used to model the timing variation due to TLB and write buffers.

This paper also reported on preliminary results of WCET analyses that were obtained from a timing tool being developed by our research group. The results showed that up to 50 % tighter WCET bounds can be obtained even when only the effects of pipelined execution across basic blocks are taken into account.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[2] Y. H. Bae. Data Cache Analysis Techniques for Real-Time Systems. Master's thesis in preparation, Seoul National University, 1994.

[3] S. Basumalick and K. D. Nilsen. Incorporating Caches in Real-Time Systems. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, April 1994.

[4] J.-Y. Choi, I. Lee, and I. Kang. Timing Analysis of Superscalar Processor Programs Using ACSR. In *Proceedings of the 11th Workshop on Real-Time Operating Systems and Software*, pages 63–67, May 1994.

[5] F. Chow and J. L. Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 222–232, 1984.

[6] C. N. Fischer and Jr. R. J. Leblanc. *Crafting a Compiler with C.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[7] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. Technical Report CSL-TR-270-90, Dept. of Computer Science, Princeton University, July 1990.

[8] E. Harcourt, J. Mauney, and T. Cook. High-Level Timing Specification of Instruction-Level Parallel Processors. Technical Report TR-93-18, Dept. of Computer Science, North Carlina State University, August 1993.

[9] M. Harmon, T. P. Baker, and D. B. Whalley. A Retargetable Technique for Predicting Execution Time. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 68–77, 1992.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[11] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, Nov. 1987.

[12] G. Kane and J. Heimrich. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[13] R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Mathematics*, 23:309–311, 1978.

[14] D. B. Kirk. Process Dependent Static Cache Partitioning for Real-Time Systems. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 181–190, 1988.

[15] D. B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proceedings of the 10th Real-Time Systems Symposium*, pages 229–237, 1989.

[16] P. M. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Publishing Corp., 1981.

[17] S.-S. Lim. Instruction Cache and Pipelining Analysis Techniques for Real-Time Systems. Master's thesis in preparation, Seoul National University, 1994.

[18] A. Mok. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proceedings of the 6th Workshop on Real-Time Operating Systems and Software*, pages 74–80, 1989.

[19] F. Mueller, D. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. Unpublished Technical Report, 1993.

[20] K. Narasimhan and K. D. Nilsen. Portable Execution Time Analysis for RISC Processors. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, April 1994.

[21] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable Real-Time Caching in the Spring System. In *Proceedings of the 8th Workshop on Real-Time Operating Systems and Software*, pages 76–80, 1991.

[22] C. Y. Park and A. C. Shaw. Experiments With A Program Timing Tool Based On Source-Level Timing Schema. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 72–81, 1990.

[23] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159–176, Sept. 1989.

[24] J. Rawat. Static Analysis of Cache Performance for Real-Time Programming. Master's thesis, Iowa State University, 1993.

[25] B.-D. Rhee, S.-S. Lim, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. Issues of Advanced Architectural Features in the Design of a Timing Tool. In *Proceedings of the 11th Workshop on Real-Time Operating Systems and Software*, pages 59–62, May 1994.

[26] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions On Software Engineering*, 15(7):875–889, July 1989.

[27] A. Stoyenko. *A Real-Time Language with a Schedulability Analyzer.* PhD thesis, University of Toronto, Dec. 1987.

[28] N. Zhang, A. Burns, and M. Nicholson. Pipelined Processors and Worst-Case Execution Times. *Journal of Real-Time Systems*, 5(4):319–343, Oct. 1993.