# An Accurate Worst Case Timing Analysis for RISC Processors

Sung-Soo Lim, *Student Member, IEEE*, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, *Member, IEEE*, Chang Yun Park, Heonshik Shin, *Member, IEEE*, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim, *Senior Member, IEEE*

*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, we propose extensions to the original timing schema where the timing information associated with each program construct is a simple time-bound. In our approach, associated with each program construct is worst case timing abstraction, (WCTA), 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 analysis for a RISC processor. Our results show that tight WCET bounds (within a maximum of about 30% overestimation) can be obtained by using the revised timing schema approach.**

*Index Terms*—**Cache memory, pipelined execution, real-time system, RISC processor, worst case execution time.**

## I. 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 worst case execution time (WCET) of each task in the system be known a priori. Therefore, it is not surprising that considerable research has focused on the estimation of the WCETs of tasks.

In a nonpipelined processor without cache memory, it is relatively easy to obtain a tight bound on the WCET of a sequence of instructions. One simply has to sum up their individual execution times that are usually given in 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 by assuming the worst case execution scenario (e.g., each instruction suffers from every kind of hazard and every memory access results in a cache miss). 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* [28]. 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 instruction 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 to the original timing schema to rectify the above problem. We associate with each program construct what we call a worst case timing abstraction (WCTA). The WCTA of a program construct contains timing information of every execution path that *might* be the worst case execution path of the program 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 which results from the history sensitive nature of pipelined execution and cache memory.

We assume that there is no parallelism within a task and that some form of cache partitioning [15], [30] is used to prevent

Manuscript received Apr. 14, 1995.

Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Heonshik Shin, Kunsoo Park, and Chong Sang Kim are with the Department of Computer Engineering, Seoul National University, Seoul 151-742, Korea; e-mail: symin@dandelion.snu.ac.kr.

Chang Yun Park is with the Department of Computer Science and Engineering, Chungang University, Seoul 156-756, Korea.

Soo-Mook Moon is with the Department of Electronics Engineering, Seoul National University, Seoul 151-742, Korea.

IEEECS Log Number S95017.

tasks from affecting each other's timing behavior. Without these assumptions, it would not be possible to eliminate the unpredictability due to task interaction. For example, consider a real-time system in which a preemptive scheduling policy is used and the cache is not partitioned. In such a system, a burst of cache misses usually occurs when a previously preempted task resumes execution. Increase of the task execution time resulting from such a burst of cache misses cannot be bounded by analyzing each task in isolation.

This paper is organized as follows. In Section II, we survey the related work. Section III explains the problems associated with accurately estimating the WCETs of tasks in pipelined processors and presents our method for solving these problems. In Section IV, 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 III. Section V 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 VI, we report on preliminary results of WCET analyses for a RISC processor. Finally, the conclusion is given in Section VII.

## II. 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. Due to these limitations, analytical approaches are becoming more popular [2], [4], [8], [9], [18], [19], [20], [21], [24], [25], [26], [29], [31]. Many of these analytical studies, however, consider a simple machine model, thus largely ignoring the timing effects of pipelined execution and cache memory [19], [24], [25], [29].

### A. 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 the resources in time. In order to mechanize the process of calculating the execution time, they use various techniques: pattern matching [9], synchronous calculus of communicating systems (SCCS) [8], retargetable pipeline simulation [20], and algebra of communicating shared resources (ACSR) [4]. Although these approaches have the advantage 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 [31] 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 significant progress over the previous schemes that did not consider 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 within the execution pipeline, a basic block's execution time will differ depending on what the surrounding basic blocks are. However, since their approach requires that the WCET of each basic block be independently calculated, 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 has 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. Second, although their mathematical model is very effective for the Intel 80C188 processor, the model is not general enough to be applicable to other pipelined processors. This is due to many machine specific assumptions made in their model that are difficult to generalize.

### B. 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 concern stems from the following two sources: intertask interference and intratask interference. Intertask 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 miss cannot be avoided in real-time systems with preemptive scheduling of tasks. The result is a wide variation in task execution time. This execution time variation can be eliminated by partitioning the cache and

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].

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

dedicating one or more partitions to each real-time task [15], [30]. This cache partitioning approach eliminates the intertask interference caused by task preemption.

Intratask 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 miss: *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 miss 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 four schemes take into account the timing variation resulting from intratask cache interference (three for instruction caches [2], [18], [21] and one for data caches [26]). The *static cache simulation* approach which statically predicts hits or misses of instruction references is due to Arnold, Mueller, Whalley, and Harmon [2]. 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 may not be in the cache.
- *first-hit*: The first reference to the instruction hits in the cache. However, all the subsequent references miss in the cache.
- *first-miss*: The first reference to the instruction misses in the cache. However, all the subsequent references hit in the cache.

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 Fig. 1. Assume that both of the instruction memory blocks corresponding to $S_i$ (i.e., $b_i$) and $S_j$ (i.e., $b_j$) are mapped to the same cache block and that no other instruction memory block is mapped to that cache block. Further assume that the execution time of $S_i$ is much longer than that of $S_j$. Under these assumptions, the worst case execution scenario of this program fragment is to repeatedly execute $S_i$ 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, by being classified as *always-miss*, all the references to $b_i$ are treated as cache misses in this approach, which leads to a loose estimation of the loop's WCET. (In the approach, a reference to an instruction memory block is classified as *always-miss* if other instruction memory blocks in the loop map to the same cache block and the memory block is not in the cache initially.) Another limitation of this approach is that the approach does not address the issues regarding pipelined execution and the use of data caches, which are commonly found in most RISC processors.

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 stated in [2], their analysis is rather abstract and no general method for analyzing the worst case timing behavior of programs is given.

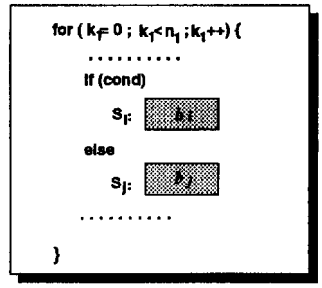


Fig. 1. Sample C program fragment.

In [18], Liu and Lee propose techniques to derive WCET bounds of a cached program based on a transition diagram of cached states. Their WCET analysis uses an exhaustive search technique through the state transition diagram which has an exponential time complexity. To reduce the time complexity of this approach, they propose a number of approximate analysis methods each of which makes a different trade-off between the analysis complexity and the tightness of the resultant WCET bounds. Although the paper mentions that the methods are equally applicable to the data cache, the main focus is on the instruction cache since the issues pertinent to the data cache such as handling of write references and references with unknown addresses (cf. Section V) are not considered. Also, it is not clear how one can incorporate the analysis of pipelined execution into the framework.

Rawat performs a static analysis for data caches [26]. His approach is similar to the graph coloring approach to register allocation [5]. The analysis proceeds as follows. 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 times the program control flows into the live ranges). Finally, the total number of 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 is a step forward from 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 result-

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 [26].

ing from the assumption that every possible execution path can be the worst case execution path. This limitation is similar to the first limitation of the static cache simulation approach. The third limitation is that it does not address the issues of locating the worst case execution path and of calculating the WCET, again limiting its applicability.

## III. Pipelining Effects

In pipelined processors, various execution steps of instructions are simultaneously overlapped. Due to this overlapped execution, an instruction's execution time will differ depending on what the surrounding instructions are. However, this timing variation could not 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 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. Fig. 2 shows a sample basic block in the MIPS assembly language [12] 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. In the reservation table, not only the conflicts in the use of pipeline stages but also data dependencies among instructions are considered.

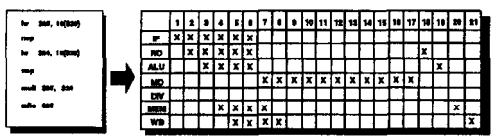

Fig. 2. Sample MIPS assembly code and the corresponding reservation table.

A program construct such as an **if** statement may have more than one execution path. Moreover, in pipelined processors, it is not always possible 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 Fig. 3. The worst case execution path here depends on the instructions in the preceding 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 is 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 preceding program constructs is known.

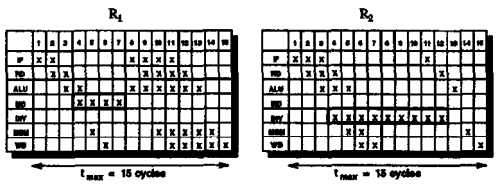


Fig. 3. Two reservation tables with equal $t_{max}$.

Fig. 4 shows the data structure for a reservation table used in our approach in both textual and graphical form. 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 are maintained. Instead, we maintain only a first few (i.e., $\delta_{head}$) columns and a last few (i.e., $\delta_{tail}$) columns. The larger $\delta_{head}$ and $\delta_{tail}$ are, the tighter the resulting WCET estimation is since more execution overlap between program constructs can be modeled as we will see later. $\delta_{head} = \delta_{tail} = \infty$ corresponds to the case where the full reservation table is maintained.

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

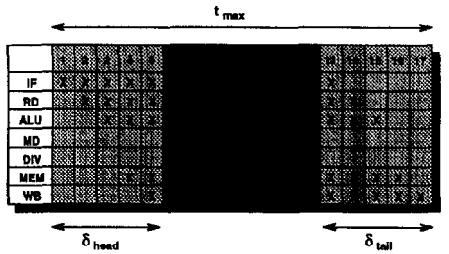


Fig. 4. Reservation table data structure.

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 worst case timing abstraction (WCTA) of the program construct. This WCTA corresponds to the time-bound in the original timing schema and each element in the WCTA is denoted by ($t_{max}$, *head*, *tail*).

With this framework, the timing schema can be extended so that the timing interactions 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

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

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

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

where $w_1$ and $w_2$ are reservation tables and the $\oplus$ operation concatenates two reservation tables resulting in another 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. Fig. 5 shows an application of the $\oplus$ operation. From the figure, one can note that as more columns are maintained in `head` and `tail`, more overlap between adjacent program constructs can be modeled and, therefore, a tighter WCET estimation can be obtained.

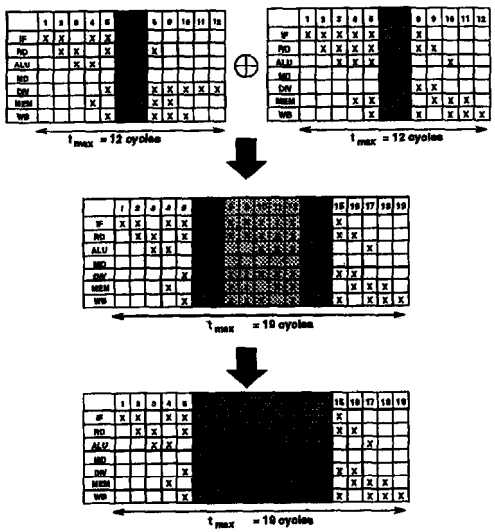


Fig. 5. Example application of $\oplus$ operation.

The above timing formula for **S: $S_1$; $S_2$** effectively enumerates all the possible candidates for the worst case execution path of $S_1$; $S_2$. However, 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 removed from the WCTA if we can guarantee that the element's WCET in the worst case scenario is shorter than the best case scenario WCET of some other element in the same WCTA. 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 $w$ (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 $w'$ (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

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

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 $\cup$ 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 the functions that call it are processed.

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

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

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 provide approximate methods for loop timing analysis.

*Approximate Loop Timing Analysis.* The problem of finding the worst case execution scenario for a loop statement with loop bound $N$ can be formulated as a problem to find the longest weighted path (not necessarily simple) containing exactly $N$ arcs in a weighted directed graph. Thus, the approximate loop timing analysis method is explained using a graph theoretic formulation.

Let $G = (P, A)$ be a weighted directed graph where $P = \{p_1, p_2, \ldots, p_{|P|}\}$ is the set of the execution paths in the loop body that *might* be the worst case execution path (i.e., those in $W(exp) \oplus W(S_1)$) and associated with each arc is weight $w_{ij}$ which is the execution time of path $p_j$ when its execution is immediately preceded by path $p_i$. Define $D_{\ell,i,j}$ as the weight of the longest path (not necessarily simple) from $p_i$ to $p_j$ in $G$ containing exactly $\ell$ arcs. With these definitions, the $t_{max}$ of the loop's worst case execution scenario that starts with path $p_i$ and ends with path $p_j$ is given by $p_i.t_{max} + D_{N-1,i,j}$ where $p_i.t_{max}$ is $t_{max}$ of path $p_i$. The WCTA of this worst case execution scenario inherits $p_i$'s `head` since it starts with $p_i$. Likewise, it inherits $p_j$'s `tail`. From these, the WCTA of the loop's worst case execution scenario that starts with path $p_i$ and ends with path $p_j$, which is denoted by $wcta(wp_{ij}^{N})$, is given by $(p_i.t_{max} + D_{N-1,i,j}, p_i.head, p_j.tail)$. Since the actual worst case execution scenario of the loop depends on the program constructs surrounding the loop statement, we do not know with which paths the actual worst case execution scenario starts and ends when we analyze the loop statement. Therefore, one has to consider all the possibilities. The corresponding WCTA of the loop statement is given 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 that of $g$ in their call graph.

$$\left(\bigcup_{p_i,p_j \in P} wcta\left(wp_{ij}^{N}\right)\right) \oplus W(exp).$$

The only remaining problem is to determine $D_{N-1,i,j}$. One can determine the value by solving the following equations.

$$D_{0,i,j} = \begin{cases} -\infty & \text{if } p_i \neq p_j \\ 0 & \text{otherwise} \end{cases}$$

$$D_{\ell,i,j} = \max_{p_k \in P} \left\{ D_{\ell-1,i,k} + w_{kj} \right\}$$

Computation of $D_{\ell,i,j}$ for all $p_i, p_j \in P$ and $0 \leq \ell < N$ using dynamic programming takes $O(N \times |P|^3)$ time. For a large $N$, this time complexity is still unacceptable. In the following, we describe a faster technique that gives a very tight upper bound for $D_{\ell,i,j}$. This technique is based on the calculation of the maximum cycle mean of $G$.

The maximum cycle mean of a weighted directed graph $G$ is $m = \max_{c \in C} m(c)$ where $C$ ranges over all directed cycles in $G$ and $m(c)$ is the mean weight of $c$. The maximum cycle mean can be calculated in $O(|P| \times |A|)$ time, which is independent of $N$, using an algorithm due to Karp [13]. Let $m$ be the maximum cycle mean of $G$, then $D_{\ell,i,j}$ can safely be approximated as $D'_{\ell,i,j} = \ell \times m + (m - w_{ji})$. 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 $\ell$ 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 has been 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 $W(exp) \oplus W(S_1)$ that cannot be pruned by each other.)

*Interference.* Up to now, we have assumed 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. For a task, these preemptions are interference that breaks in the task's execution flow. The problem regarding interference is that of adjusting the prediction made under the assumption of no interference such that the prediction is applicable in an environment with interference. Fortunately, the additional per-preemption delay introduced by pipelined execution is bounded by the maximum number of cycles for which an instruction remains in the pipeline (in MIPS R3000 it is 36 cycles in the case of the **div** instruction). Once this information is available, adjusting the predictions to reflect interference can be done using the techniques explained in [22].

## IV. Instruction Caching Effects

For a processor with an instruction cache, the execution time of a program construct will differ depending on which execution path was taken prior to the program construct. This is a result of the 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. Fig. 6). 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.*

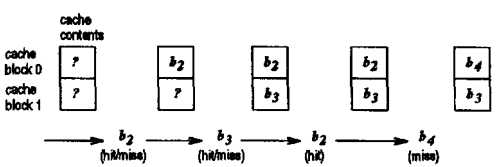


Fig. 6. Sample instruction block references from a program construct.

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 the mean time. 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. (Note 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 can be determined by local analysis, the hit or miss of the first reference to $b_2$ cannot be determined locally and is dependent on the cache contents immediately before executing this program construct. Similarly, the hit or miss of the reference to $b_3$ will depend on the previous cache contents. The hits or misses of these two references will affect the (worst case) execution time of this program construct. Moreover, the cache contents after executing this program construct will, in turn, affect the execution time of the succeeding program construct in a similar

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

way. These timing variations, again, cannot be accurately represented by a simple time-bound of the original timing schema.

This situation is similar to the case of pipelined execution discussed in the previous section and, therefore, we adopt the same strategy; we simply extend the timing information of elements in the WCTA leaving the timing formulas intact. Each element in the 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. Fig. 7 gives the data structure for an element in the WCTA in this new setting where $n_{block}$ denotes the number of blocks in the cache.

```
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];
};
```

Fig. 7. Structure of an element in a WCTA.

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 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. These are 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 calculating $t_{max}$, we accurately account for the hits and misses that can be locally determined 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 at a later stage of the analysis. Fig. 8 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

$$W(S) = W(S_1) \oplus W(S_2).$$

This timing formula is structurally identical to the one given in the previous section for the sequential statement. The differences are in the structure of the elements in the WCTAs and in the semantics of the $\oplus$ operation. The revised semantics of the $\oplus$ operation is procedurally defined in Fig. 9.

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, $w_3$ would inherit $w_2$'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. 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.

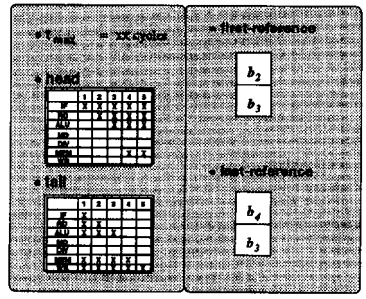


Fig. 8. Contents of the WCTA element corresponding to the example in Fig. 6.

```
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 nhits, i;
6
7      nhits = 0;
8      for (i = 0; i < nblock; i++) {
9        if (w1.first_reference[i] != NULL)
10         w3.first_reference[i] = w1.first_reference[i];
11       else
12         w3.first_reference[i] = w2.first_reference[i];
13       if (w2.last_reference[i] != NULL)
14         w3.last_reference[i] = w2.last_reference[i];
15       else
16         w3.last_reference[i] = w1.last_reference[i];
17       if ((w1.last_reference[i] = = w2.first_reference[i]) &&
18         (w1.last_reference[i] != NULL))    nhits++;
19     }
20     w3.head = w1.head;
21     w3.tail = w2.tail;
22     w3.tmax = ((w1.tmax, w1.head, w1.tail) ⊕pipeline
23               (w2.tmax, w2.head, w2.tail)).tmax
24               - nhits * tmiss penalty;
25     return w3;
26  }
```

Fig. 9. Semantics of the $\oplus$ operation.

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 Fig. 10. The function `prune` given in the figure checks whether either one of the two execution paths corresponding to the 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.

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

Fig. 10. Semantics of pruning operation.

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`. The difference bounds the cache memory related execution time variation between $w_1$ and $w_2$. 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 WCET assuming the worst case scenario for $w_2$ is shorter than $w_1$'s WCET assuming $w_1$'s 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

$$W(S) = (W(exp) \oplus W(S_1)) \cup (W(exp) \oplus W(S_2)) = W(exp) \oplus (W(S_1) \cup W(S_2))$$

As in the previous section, the problem of calculating $W(S)$ for a loop statement **S: while (exp) $S_1$** can be formulated as a graph theoretic problem. Here, $wcta(wp_{ij}^N)$ is given by

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

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

$$\left( \bigcup_{p_i, p_j \in P} wcta\left(wp_{ij}^N\right) \right) \oplus W(exp).$$

The loop timing analysis discussed in the previous section assumes that each loop iteration benefits only from the immediately preceding loop iteration. This is because in the calculation of $w_{ij}$, we only consider the execution time reduction of $p_j$ due to the execution overlap with $p_i$. 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. In the case of cache memory, however, the assumption 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 the result does not underestimate the WCET of the loop statement. The degradation of accuracy resulting from this conservative assumption can be reduced by analyzing a sequence of $k$ ($k > 1$) iterations at the same time 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 to now we have considered only the simplest cache organization called the direct-mapped cache in which each instruction block can be placed exactly in one cache block. In a more general cache organization called the $n$-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 the set to make room for a block fetched on a cache miss. The Least Recently Used (LRU) 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 method.

## V. 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, assumes 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

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.*

available. This bit, called the *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 preventing 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, i.e., 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 by taking two cache miss penalties for each data reference whose address cannot be determined at compile-time, and then ignoring the reference in the analysis [3]. The one cache miss penalty is due to the fact that the reference may miss in the cache. The other is due to the fact that the reference may replace a cache block that contributes a cache hit in our analysis.

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 popped on a runtime stack as the associated function is called and returns. In most 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 `sp` value of a function differs depending on from where the function is called. However, the number of distinct `sp` values a function may have is bounded. Therefore, the WCTA of a function can be computed for each `sp` value 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, either *write-through* or *write-back* policy is used to handle write accesses [10]. In the write-through policy, the effect of each write is reflected on both the block in the cache and the block in main memory. On the other hand, in the write-back policy, the effect is reflected only on the block in the cache and a dirty bit is set to indicate that the block has been modified. When a block whose dirty bit is set is replaced from the cache, the block's contents are written back to main memory.

The timing analysis of data caches with the write-through policy is relatively simple. One simply has to add a 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 without a replacement in-between, which we call a *write run*, requires only one write-back to main memory. We attribute this write-back overhead (i.e., delay) to the last write in the write run, which we call the *tail* of the write run. With this setting, one has to determine whether a given write access can be a tail to accurately estimate the delay due to write-backs. In some cases, local analysis can determine whether a write access is a tail or not as in the case of hit/miss analysis for a memory reference. However, local analysis is not sufficient to determine whether a write access is a tail in every case. Hence, when this is not possible, we conservatively assume that the write access is a tail and add a write-back delay to $t_{max}$. However, if later analysis over the program syntax tree reveals that the write access is not a tail, we subtract the incorrectly attributed write-back delay from $t_{max}$. This global analysis can be performed by providing a few bits to each block in `first_reference` and `last_reference` and augmenting the $\oplus$ and pruning operations [3].

## VI. Experimental Results

We tested whether our extended timing schema approach could produce useful WCET bounds by building a timing tool based on the approach and comparing the WCET bounds predicted by the timing tool to the measured times. Our timing tool consists of a compiler and a timing analyzer (Fig. 11). The compiler is a modified version of an ANSI C compiler called lcc [7]. The modified compiler accepts a C source program and generates the assembly code along with the program syntax information and the call graph. The timing analyzer uses the assembly code and the program syntax information along with user-provided information (e.g., loop bound) to compute the WCET of the program.

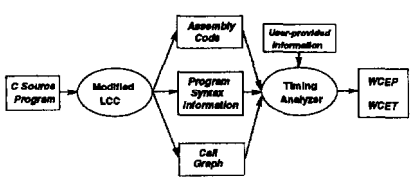


Fig. 11. Overview of the timing tool.

We chose an IDT7RS383 board as the timing tool's target machine. The target machine's CPU is a 20 MHz R3000 processor which is a typical RISC processor. The R3000 processor has a five-stage integer pipeline and an interface for off-chip instruction and data caches. It also has an interface for an off-chip floating-point unit (FPU).

The IDT7RS383 board contains instruction and data caches of 16 kbytes each. Both caches are direct-mapped and have block sizes of 4 bytes. The data cache uses the write-through policy and has a one-entry deep write buffer. The cache miss service times of both the instruction and data caches are four cycles. The FPU used in the board is a MIPS R3010. Although the board has a timer chip that provides user-programmable timers, their resolutions are too low for our measurement purposes. To facilitate the measurement of program execution times in machine cycles, we built a daughter board that consists of simple decoding circuits and counter chips, and provides one user-programmable timer. The timer starts and stops by writing to specific memory locations and has a resolution of one machine cycle (50 nsec).

Three simple benchmark programs were chosen: *Clock*, *Sort* and *MM*. The *Clock* benchmark is a program used to implement a periodic timer. The program periodically checks 20 linked-listed timers and, if any of them expires, calls the corresponding handler function. The *Sort* benchmark sorts an array of 20 integer numbers and the *MM* program multiplies two $5 \times 5$ floating-point matrices.

Table I compares the WCET bounds predicted by the timing tool and the measured execution times for the three benchmark programs. In all three cases, the tool gives fairly tight WCET bounds (within a maximum of about 30% overestimation). A closer inspection of the results revealed that more than 90% of the overestimation is due to data references whose addresses are not known at compile-time. (Remember that we have to account for two cache miss penalties for each such data reference.)

TABLE I
PREDICTED AND MEASURED EXECUTION TIMES OF THE BENCHMARK PROGRAMS

| | *Clock* | *Sort* | *MM* |
|---|---|---|---|
| **Predicted** | 3,202 | 14,259 | 8,376 |
| **Measured** | 2,768 | 11,471 | 6,346 |

(unit: machine cycles)

Program execution time is heavily dependent on the program execution path, and the logic of most programs severely limits the set of possible execution paths. However, we intentionally chose benchmark programs that do not suffer from overestimation due to infeasible paths. The rationale behind this selection is that predicting tighter WCET bounds by eliminating infeasible paths using dynamic path analysis is an issue orthogonal to our approach and that this analysis can be introduced into the existing timing tool without modifying the extended timing schema framework. In fact, a method for analyzing dynamic program behavior to eliminate infeasible paths of a program within the original timing schema framework is given in [23] and we feel that our timing tool will equally benefit from the proposed method.

We view our experimental work reported here as an initial step toward validating our extended timing schema approach. Clearly, much experimental work, especially with programs used in real systems, needs to follow to demonstrate that our approach is practical for realistic systems.

## VII. 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 timing information of the preceding program construct becomes available at a later stage of WCET analysis. We extended the existing timing schema using these two kinds of timing information so that we can accurately account for the timing variations resulting from the 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 part of the worst case execution path.

We also built a timing analyzer based on the proposed technique and compared the WCET bounds of sample programs predicted by the timing analyzer to their measured execution times. The timing analyzer gave fairly tight predictions (within a maximum of about 30% overestimation) for the benchmark programs we used and the sources of the overestimation were identified.

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, previously, was not possible. Furthermore, the timing analysis using the proposed technique is more accurate than that of any other technique we are aware of. Second, the proposed technique is applicable to most RISC processors with in-order issue and single-level cache memory. Finally, the proposed technique is extensible in that its general rule may be used to model other machine features that have history sensitive timing behavior. For example, we used the underlying general rule to model the timing variation due to write buffers [3].

One direction for future research is to investigate whether or not the proposed technique applies to more advanced processors with out-of-order issue [14] and/or multilevel cache hierarchies [10]. Another research direction is in the development of theory and methods for the design of a retargetable timing analyzer. Our initial investigation on this issue was made in [27]. The results indicated that the machine-dependent components of our timing analyzer such as the routines that implement the concatenation and pruning operations of the extended timing schema can be automatically generated from an architecture description of the target processor. The details of the approach are not repeated here and interested readers are referred to [27].

## ACKNOWLEDGMENTS

The authors are grateful to R. Gerber, S. Hong, and S. Noh, who provided many helpful suggestions and comments on earlier versions of this paper. The authors also want to thank the anonymous referees for their constructive comments. This work was supported in part by KOSEF under Grant KOSEF-93-01-00-06.

## REFERENCES

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

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding worst-case instruction cache performance," *Proc. 15th IEEE Real-Time Systems Symp.*, pp. 172–181, 1994.

[3] Y.H. Bae, "Data cache analysis techniques for real-time systems," Masters thesis, Seoul National University, Korea, 1995.

[4] J.-Y. Choi, I. Lee, and I. Kang, "Timing analysis of superscalar processor programs using ACSR," *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 63–67, May 1994.
[5] F. Chow and J.L. Hennessy, "Register allocation by priority-based coloring," *Proc. ACM SIGPLAN '84 Symp. Compiler Construction*, pp. 222–232, 1984.
[6] C.N. Fischer and R.J. LeBlanc, *Crafting a Compiler with C*. Redwood City, Calif.: The Benjamin/Cummings Publishing Co., Inc., 1991.
[7] C.W. Fraser and D.R. Hanson, "A code generation interface for ANSI C," Tech. Report CSL-TR-270-90, Dept. of Computer Science, Princeton Univ., 1990.
[8] E. Harcourt, J. Mauney, and T. Cook, "High-level timing specification of instruction-level parallel processors," Tech. Report TR-93-18, Dept. of Computer Science, North Carolina State Univ., 1993.
[9] M. Harmon, T.P. Baker, and D.B. Whalley, "A retargetable technique for predicting execution time," *Proc. 13th IEEE Real-Time Systems Symp.*, pp. 68–77, 1992.
[10] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, Calif.: Morgan Kaufmann, 1990.
[11] M.D. Hill, "Aspects of cache memory and instruction buffer performance," PhD thesis, Univ. of California at Berkeley, 1987.
[12] G. Kane and J. Heimrich, *MIPS RISC Architecture*. Englewood Cliffs, N.J.: Prentice Hall, 1991.
[13] R.M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, pp. 309–311, 1978.
[14] R.M. Keller, "Look-ahead processors," *Computing Surveys*, vol. 7, no. 4, pp. 177–195, Dec. 1975.
[15] D.B. Kirk, "SMART (Strategic Memory Allocation for Real-Time) cache design," *Proc. 10th IEEE Real-Time Systems Symp.*, pp. 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 technique for real-time systems," Masters thesis, Seoul National University, Korea, 1995.
[18] J.-C. Liu and H.-J. Lee, "Deterministic upperbounds of the worst-case execution times of cached programs," *Proc. 15th IEEE Real-Time Systems Symp.*, pp. 182–191, 1994.
[19] A. Mok, "Evaluating tight execution time bounds of programs by annotations," *Proc. Sixth IEEE Workshop on Real-Time Operating Systems and Software*, pp. 74–80, 1989.
[20] K. Narasimhan and K.D. Nilsen, "Portable execution time analysis for RISC processors," *Proc. Workshop Architectures for Real-Time Applications*, Apr. 1994.
[21] D. Niehaus, E. Nahum, and J.A. Stankovic, "Predictable real-time caching in the Spring System," *Proc. Eighth IEEE Workshop on Real-Time Operating System and Software*, pp. 76–80, 1991.
[22] C.Y. Park, "Predicting deterministic execution times of real-time programs," PhD thesis, Univ. of Washington, 1992.
[23] C.Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems J.*, vol. 5, no. 1, pp. 31–62, Mar. 1993.
[24] C.Y. Park and A.C. Shaw, "Experiments with a program timing tool based on source-level timing schema," *Proc. 11th IEEE Real-Time Systems Symp.*, pp. 72–81, 1990.
[25] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems J.*, vol. 1, no. 2, pp. 159–176, Sept. 1989.
[26] J. Rawat, "Static analysis of cache performance for real-time programming," Masters thesis, Iowa State Univ., 1993.
[27] B.-D. Rhee, S. L. Min, and H. Shin, "Retargetable timing analyzer for RISC processors," *Proc. First Int'l Workshop Real-Time Comput. Systems and Applications.*, pp. 76–79, Dec. 1994.
[28] A.C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. on Software Engineering*, vol. 15, no. 7, pp. 875–889, July 1989.
[29] A. Stoyenko, "A real-time language with a schedulability analyzer," PhD thesis, Univ. of Toronto, 1987.
[30] A. Wolfe, "Software-based cache partitioning for real-time applications," *Proc. Third Workshop Responsive Computer Systems*, Sept. 1993.
[31] N. Zhang, A. Burns, and M. Nicholson, "Pipelined processors and worst-case execution times," *Real-Time Systems J.*, vol. 5, no. 4, pp. 319–343, Oct. 1993.

**Sung-Soo Lim** received the BS and MS degrees in computer engineering from Seoul National University, Korea, in 1993 and 1995, respectively. Currently, he is a PhD student in the Department of Computer Engineering, Seoul National University.

His research interests include real-time systems with an emphasis on worst-case timing analysis, computer architecture, and computer networks.

He is a student member of the Association for Computing Machinery and the IEEE Computer Society.

**Young Hyun Bae** received the BS and MS degrees in computer engineering from Seoul National University, Korea, in 1993 and 1995, respectively. He is presently a PhD student in the Department of Computer Engineering at Seoul National University.

His current research interests include computer architecture, compiler optimization, real-time systems, computer networks, and computer music.

**Gyu Tae Jang** received the BS degree in computer engineering from Pusan National University, Korea, in 1993 and the MS degree in computer engineering from Seoul National University, Korea, in 1995.

Since 1995 he has worked at Shinsegi Telecomm. Inc., Seoul, Korea, on CDMA system engineering and internetworking with mobile satellite communication systems.

**Byung-Do Rhee** received the BS and MS degrees in computer engineering from Seoul National University, Korea, in 1982 and 1984, respectively. He is completing his PhD degree in the Department of Computer Engineering at Seoul National University.

From 1984 to 1992 he was a research engineer at Korea Telecommunications. His current research interests include real-time computing, computer architecture, and data communications.

**Sang Lyul Min** received the BS and MS degrees in computer engineering (both from Seoul National University, Korea) in 1983 and 1985, respectively. In 1985, he was awarded a Fulbright scholarship to pursue further graduate studies at the University of Washington. He received the MS and PhD degrees in computer science from the University of Washington, Seattle in 1988 and 1989.

He is currently an assistant professor in the Department of Computer Engineering, Seoul National University, Korea. Previously, he was an assistant professor in the Department of Computer Engineering, Pusan National University, Pusan, Korea from 1989 to 1992 and a visiting scientist at the IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., from 1989 to 1990.

His research interests include computer architecture, real-time computing, parallel processing, and computer performance evaluation. Dr. Min is a member of the Association for Computing Machinery and the IEEE Computer Society.

**Chang Yun Park** received the BS and MS degrees in computer engineering from Seoul National University, Korea, in 1984 and 1986, respectively; and the PhD degree in computer science from the University of Washington, Seattle, in 1992.

He is currently an assistant professor in the Department of Computer Science and Engineering, Chungang University, Seoul, Korea. His research interests include real-time systems, computer networks, and software specification methods.

**Heonshik Shin** received the BS degree in applied physics from Seoul National University, Korea, in 1973 and the PhD degree in computer engineering from the University of Texas at Austin in 1985.

He is currently an associate professor of Computer Engineering at Seoul National University. His research interests include real-time computing, distributed systems, and I/O processing. H. Shin is a member of the IEEE Computer Society.

**Kunsoo Park** received a BS and an MS degree in computer engineering from Seoul National University, Korea, in 1983 and 1985, respectively, and a PhD in computer science from Columbia University in 1991.

From 1991 to 1993 he was a lecturer in computer science at King's College, University of London. Since August 1993, he has been a lecturer in the Department of Computer Engineering at Seoul National University. His research interests include algorithm design and analysis, parallel computation, and cryptography.

**Soo-Mook Moon** received the BS degree in computer engineering from Seoul National University, Korea in 1987, and the MS and PhD degrees from the University of Maryland at College Park, in 1990 and 1993, respectively.

During the summer of 1991, 1992, and 1993, he was a co-op student at the IBM Thomas J. Watson Research Center. During 1993-1994, he was a software design engineer at the Hewlett-Packard Company. In 1994, he joined the Department of Electronics at Seoul National University.

His current research interests are instruction-level parallelism, optimizing and parallelizing compilation, microprocessor architecture, and performance evaluation.

**Chong Sang Kim** received the PhD degree in electronic engineering from Seoul National University.

Since 1977 he has been a professor at the Department of Computer Engineering, Seoul National University. His research interests are in computer architecture and computer networks. C.S. Kim is a senior member of the IEEE.