In Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium, pages 230-240, June 1996.

Efficient Worst Case Timing Analysis of Data Caching

Sung-Kwan Kim Sang Lyul Min
Department of Computer Engineering
Seoul National University
Seoul 151-742, Korea
symin@dandelion.snu.ac.kr

Rhan Ha
Department of Computer Engineering
Hong-Ik University
Mapo-gu, Seoul 121-791, Korea
rhanha@cs.hongik.ac.kr

Abstract

Recent progress in worst case timing analysis of programs has made it possible to perform accurate timing analysis of pipelined execution and instruction caching, which is necessary when a RISC processor is used as the target processor of a real-time system. However, there has not been much progress in worst case timing analysis of data caching. This is mainly due to load/store instructions that reference multiple memory locations such as those used to implement array and pointer-based references. These load/store instructions are called dynamic load/store instructions and most current analysis techniques take a very conservative approach to their timing analysis. In many cases, it is assumed that each of the references from a dynamic load/store instruction will miss in the cache and replace a cache block that would otherwise lead to a cache hit. This conservative approach results in severe overestimation of the worst case execution time (WCET). This paper proposes two techniques to minimize the WCET overestimation due to such load/store instructions. The first technique uses a global data flow analysis technique to reduce the number of load/store instructions that are misclassified as dynamic load/store instructions. The second technique utilizes data dependence analysis to minimize the adverse impact of dynamic load/store instructions. This paper also compares the WCET bounds of simple benchmark programs that are predicted with and without applying the proposed techniques. The results show that they significantly (up to 20%) improve the accuracy of WCET estimation especially for programs with a large number of references from dynamic load/store instructions.

1. Introduction

To calculate tight worst case execution time (WCET) bounds of programs is an important research topic in the real-time computing area since most scheduling algorithms for real-time systems assume that such bounds are available *a priori* to base their scheduling decision. For a RISC processor, however, the calculation of a tight WCET bound of a program involves difficulties that come from the very characteristics of RISC processors: pipelined execution and instruction/data caching. Recently there has been much progress in worst case timing analysis for RISC processors [2, 5, 7, 10, 11, 13, 14].

However, most of the previous studies focused mainly on the timing analysis of pipelined execution and instruction caching, while largely ignoring the data caching effects [2, 5, 10, 13, 14]. Even the approaches that do consider the timing effects of data caching have severe restrictions. For example, the technique explained in [11] requires that the addresses of references from each program construct be fixed. For instruction block[1] references, such a requirement is satisfied for programming languages that do not have any dynamic control flow structures such as computed `gotos`. However, in the case of data block references, the requirement does not hold in general. For example, a load/store instruction that is used to implement an array access references many different memory locations. If a load/store instruction references more than one memory location, it is called a dynamic load/store instruction and the extended timing schema approach [11] on which this paper is based takes a very conservative approach to such load/store instructions. The approach assumes that each of the references from a dynamic load/store instruction misses in the cache and replaces from the cache a memory block that would otherwise lead to a cache hit. This conservative approach results in severe overestimation of the WCET for programs with a large number of dynamic load/store instructions for array and pointer-based references [7].

This paper proposes two techniques to minimize the WCET overestimation resulting from dynamic load/store instructions. The first technique uses a global data flow

[1] A *block* is the minimum unit of information that can be either present or not present in the cache-main memory hierarchy [6]. We assume without loss of generality that memory references are in the unit of blocks.

analysis technique [1] to reduce the number of load/store instructions that are misclassified as dynamic load/store instructions. The second technique utilizes a data dependence analysis technique [3] to minimize the WCET overestimation resulting from the two conservative assumptions explained earlier.

This paper is organized as follows. In the next section, we describe the timing schema approach [17] and its extensions [7, 11] on which our two proposed techniques are based. Sections 3 and 4 present, in detail, the two techniques. In Section 5, we give the results from our experiments to assess the effectiveness of the proposed techniques. Finally, we conclude this paper in Section 6.

2. Timing Schema Approach and Its Extensions

A timing schema is a set of formulas for reasoning about the timing behavior of various language constructs [17]. Table 1 gives a timing schema for computing WCET bounds of commonly used language constructs. Methods based on this timing schema can derive a WCET bound of a given program by processing the program's syntax tree in a bottom-up manner and applying the formulas according to language constructs [16, 17].

The timing schema approach is simple and allows for efficient bottom-up timing analysis of programs. One problem with this approach, however, is that in its purest form it lacks provisions for the case where program constructs have variable execution times depending on factors that are not known when the program constructs are processed in a bottom-up manner. Such a case, for example, arises when the target processor has cache memories. With cache memories, the execution time of a program construct is affected by the cache hits/misses of memory references (instruction fetches, loads, and stores) made by the program construct and these cache hits/misses, in turn, are affected by memory references made by other program constructs.

The extended timing schema approach [11] is proposed to rectify the problem above. In this approach, the WCET bound in the original timing schema approach is replaced with what is called the worst case timing abstraction (WCTA) [11]. In general, a program construct may have more than one execution path as in the case of an **if** statement and the WCETs of these execution paths may differ significantly depending on other program constructs. Thus, the worst case execution path of a program construct may not always be determined by simply analyzing the program construct independently of other program constructs. For this reason, the WCTA of a program construct contains timing information of every execution path in the program construct that *might* be the worst case execution path of the program construct.

The WCTA of a program construct contains, for each execution path in the program construct, what is called the path abstraction (PA) of the execution path. The PA of an execution path encodes the factors that affect the (worst case) execution time of the execution path but are not known when the execution path is processed in a bottom-up manner. For example, when the target processor has cache memory, such factors include the first reference to each cache block assuming a direct mapped cache [11]. The hits/misses of these references significantly affect the execution time of the execution path but they cannot be determined when the execution path is processed. For this reason, the PA includes two components called `first_reference` and `last_reference`. The `first_reference` component encodes the first reference to each cache block from the associated execution path. Determination of the hits/misses of references in `first_reference` requires the information about the cache contents at the entry of the execution path. Such cache contents are determined by the last reference to each cache block from the preceding execution path. For this reason, each PA has this last reference information encoded in `last_reference`. In addition to the `first_reference` and `last_reference` components, each PA has t_{max}, which is the WCET bound of the execution path. Initially, its value is computed by assuming that all the memory references in `first_reference` are cache misses but is later revised as the program syntax tree is processed.

This extended timing information leads to timing formulas that are different from those of the original timing schema in that $\oplus$ (concatenation) and $prune$ operations on PAs are newly defined to replace the $+$ and max operations on the WCET bounds in the original timing schema. The $\oplus$ operation between two PAs models the execution of one path followed by that of another path and yields the PA of the combined path. During this operation, the hits/misses of the memory references in the succeeding execution path's `first_reference` can be determined from the preceding execution path's `last_reference`. The additional cache hits determined in this way are reflected in the t_{max} of the combined path, thus tightening the previously overestimated WCET bound.

The $prune$ operation, which is the counterpart of the max operation of the original timing schema, is performed on the set of PAs of a program construct and prunes the PAs whose associated execution paths cannot be the worst case execution path of the program construct. In other words, a PA of a program construct can be pruned if its WCET is always smaller than the WCET of another PA in the same program construct regardless of what the surrounding program constructs are.

Table 2 shows the timing formulas of the extended timing schema. The timing formula for $\mathbf{S}$: $\mathbf{S}_1$; $\mathbf{S}_2$ first enumerates all the possible execution paths within $\mathbf{S}_1$; $\mathbf{S}_2$. The $prune$

	Formulas for computing WCET bounds
S: S$_1$; S$_2$	$T(S) = T(S_1) + T(S_2)$ where $T(S)$, $T(S_1)$, and $T(S_2)$ are the WCET bounds of S, S_1, and S_2, respectively.
S: if (exp) then S$_1$ else S$_2$	$T(S) = \max(T(exp) + T(S_1), T(exp) + T(S_2))$
S: while (exp) S$_1$	$T(S) = N \times (T(exp) + T(S_1)) + T(exp)$ where N is a loop bound.
S: f(exp$_1$, ..., exp$_n$)	$T(S) = T(exp_1) + \ldots + T(exp_n) + T(f())$

Table 1. Formulas for computing WCET bounds of various language structures in the timing schema approach

	Formulas for computing WCTAs
S: S$_1$; S$_2$	$W(S) = W(S_1) \bigoplus W(S_2)$ where $W(S)$, $W(S_1)$, and $W(S_2)$ are the WCTAs of S, S_1, and S_2, respectively and $\bigoplus$ is defined as $W_1 \bigoplus W_2 = \{w_1 \oplus w_2 \| w_1 \in W_1, w_2 \in W_2\}$.
S: if (exp) then S$_1$ else S$_2$	$W(S) = (W(exp) \bigoplus W(S_1)) \bigcup (W(exp) \bigoplus W(S_2))$
S: while (exp) S$_1$	$W(S) = (\bigoplus_{i=1}^{N}(W(exp) \bigoplus W(S_1))) \bigoplus W(exp)$
S: f(exp$_1$, ..., exp$_n$)	$W(S) = W(exp_1) \bigoplus \ldots \bigoplus W(exp_n) \bigoplus W(f())$

Table 2. Formulas for computing WCTAs of various language constructs in the extended timing schema approach

operation after the enumeration (although it is not shown in the formula) prunes a subset of the resulting PAs whose associated execution paths cannot be the worst case execution path of **S**. Similarly, the timing formula for an **if** statement enumerates all the execution paths in both the **then** path and the **else** path. As previously, the execution paths that cannot be the worst case execution path of the **if** statement are pruned.

The timing formula for a loop statement with a loop bound N models the unrolling of the loop N times. This approach is exact but is computationally intractable for large N. In [11], Lim *et al.* give an efficient approximate loop timing analysis method using a maximum cycle mean algorithm due to Karp [9]. This approximate analysis method has an $O(|P|^3)$ time complexity where P is the set of the execution paths in the loop body that *might* be the worst case execution path of the loop body (*i.e.*, the set of the execution paths in $W(exp) \bigoplus W(S_1)$ that survive the pruning).

Function calls are processed like sequential statements. The WCTAs of functions are calculated in reverse topological order in the call graph[2] so that the WCTAs of callees are available when the caller is processed.

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

Data caching analysis within the extended timing schema approach: To compute `first_reference` and `last_reference` for data caching analysis purposes, we need to know the reference address of each load/store instruction in program constructs. For this purpose, in the extended timing schema approach load/store instructions are categorized into the following two classes depending on whether their reference addresses are fixed or variable: static load/store instructions and dynamic load/store instructions. A load/store instruction is categorized as a static load/store instruction if its reference address does not change. Otherwise it is categorized as a dynamic load/store instruction. For dynamic load/store instructions, the extended timing schema approach takes a very conservative approach; it assumes two cache miss penalties for each reference from dynamic load/store instructions and completely ignores them in the calculation of `first_reference` and `last_reference`. One cache miss penalty is because the reference may miss in the cache. The other cache miss penalty is because the reference may replace a cache block that would contribute a cache hit in the data caching analysis without such references. Although this approach is simple, it suffers from severe overestimation of the WCET especially when there are a large number of dynamic load/store instructions in the program.

Most RISC processors, for which the extended tim-

ing schema approach is targeted, are load/store architectures and they provide a very limited set of addressing modes for load/store instructions. Often, the `base_register+displacement` addressing mode is the only addressing mode provided as in the case of MIPS R3000 [8]. In this addressing mode, the address of an operand in memory is specified by the sum of the base register's value and the displacement.

In MIPS R3000, a static load/store instruction has either `gp` (global pointer) register or `sp` (stack pointer) register as its base register. The `gp` register is used to access global data and its value does not change throughout the program execution. The `sp` register is used to access local data and its value does not change within a function. A dynamic load/store instruction has a base register other than `gp` or `sp` register and is used to implement array and pointer-based references.

The two techniques proposed in this paper aim at minimizing the WCET overestimation resulting from dynamic load/store instructions. The first technique tries to reduce the number of load/store instructions that are misclassified as dynamic load/store instructions. Such a misclassification, for example, occurs when a load/store instruction has a base register other than `gp` or `sp` but the value of the base register can be expressed by `gp+constant` or `sp+constant`. The technique uses a data flow analysis technique called the use-def(ine) analysis [1] to derive expressions for base registers that use only `gp` or `sp`. The second technique, on the other hand, tries to reduce the WCET overestimation resulting from the two conservative assumptions about dynamic load/store instructions explained earlier. The next two sections detail the two techniques.

3. Accurate Classification of Load/Store Instructions

This section describes a technique that tries to minimize the load/store instructions that are misclassified as dynamic load/store instructions. As an example of such misclassification, consider the following MIPS R3000 assembly code fragment.

```
...
addiu    $15,$sp,16
...
lw       $24,0($15)
...
```

In the extended timing schema approach, the load instruction `lw $24,0($15)` is classified as a dynamic load/store instruction since its base register, register 15 (`$15`) in this case, is other than `gp` or `sp`. Thus two cache miss penalties are assumed for each of the references generated by this load instruction. However, since the value of `$15` at the load instruction can be symbolically expressed by `sp+16`, which is determined by the preceding add instruction `addiu $15,$sp,16`, the load instruction is, in fact, a static load/store instruction. A symbolic expression that has only constants, `gp`, and `sp` in its expression such as the one above is called a *resolvable* symbolic expression in this paper. Note that the value of a resolvable symbolic expression does not change within a function.

The technique explained in this section utilizes a global data flow analysis called the use-def(ine) analysis [1] to derive resolvable symbolic expressions for as many dynamic load/store instructions as possible. In the data flow analysis terms, `$15` in the previous example is *defined* in the add instruction and this definition *reaches* the load instruction where `$15` is *used* as its base register. In this case, the add instruction is called a *reaching definition* for the *use* of $15 in the load instruction. Since the add instruction is the only reaching definition for $15 in this example, the load instruction `lw $24,0($15)` can be symbolically replaced with `lw $24,16($sp)` and, afterwards, it can be regarded as a static load/store instruction. In general, there may be more than one definition for a use and the procedure for finding the set of reaching definitions for a use can be explained as follows using data flow analysis terminology.

1. Define the following equation for each basic block[3] **B** and for each register R.

   $$\mathbf{out}^R[\mathbf{B}] = \mathbf{gen}^R[\mathbf{B}] \cup (\mathbf{in}^R[\mathbf{B}] - \mathbf{kill}^R[\mathbf{B}]),$$

   where $\mathbf{in}^R[\mathbf{B}]$ and $\mathbf{out}^R[\mathbf{B}]$ are the sets of the reaching definitions for R at the entry and exit points of basic block **B**, respectively. $\mathbf{gen}^R[\mathbf{B}]$ and $\mathbf{kill}^R[\mathbf{B}]$ are the sets of the reaching definitions for R generated and killed within basic block **B**, respectively. In our analysis, $\mathbf{gen}^R[\mathbf{B}]$ is the last instruction in basic block **B** that has R as one of its target operands. If $\mathbf{gen}^R[\mathbf{B}]$ is not empty, that is, if there is at least one instruction in **B** that defines R, then $\mathbf{kill}^R[\mathbf{B}]$ is the set of all other definitions of R in the program. On the other hand, if $\mathbf{gen}^R[\mathbf{B}]$ is empty, $\mathbf{kill}^R[\mathbf{B}]$ is an empty set.

2. Perform an iterative forward data flow analysis [1] to compute $\mathbf{in}^R[\mathbf{B}]$ and $\mathbf{out}^R[\mathbf{B}]$ for each basic block **B** and for each register R. This iterative analysis uses the following two equations:

   $$\begin{aligned} \mathbf{out}^R[\mathbf{B}] &= \mathbf{gen}^R[\mathbf{B}] \cup (\mathbf{in}^R[\mathbf{B}] - \mathbf{kill}^R[\mathbf{B}]), \\ \mathbf{in}^R[\mathbf{B}] &= \bigcup_{\mathbf{P} \in pred(\mathbf{B})} \mathbf{out}^R[\mathbf{P}], \end{aligned}$$

   where $pred(\mathbf{B})$ is the set of predecessors of basic block **B**. Initially, $\mathbf{in}^R[\mathbf{B}]$ and $\mathbf{out}^R[\mathbf{B}]$ are set to $\emptyset$

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

and **gen**^{R}**[B]**, respectively, and the iteration continues until all the **in**^{R}**[B]**s and **out**^{R}**[B]**s converge.

Deriving a resolvable symbolic expression for the base register of a load/store instruction can be complicated when a defining instruction for the base register, in turn, uses registers other than `gp` or `sp`. For such a case, the definitions of the intermediate registers should be resolved and this process is repeated in the depth first search tree order until one of the following three conditions holds.

1. A defining instruction is other than simple arithmetic/logical instructions.
2. The tree forms a cycle.
3. All the leaf definitions have resolvable symbolic expressions.

The main source of the first case is when the defining instruction is a load instruction as in the case of pointer-based references. On the other hand, the second case corresponds to the case of array references. For these two cases, the base register is marked as *unresolvable*.

For the last case, we compare all the reaching definitions of each intermediate register. Only when all the reaching definitions of every intermediate register have an identical resolvable symbolic expression, the base register is replaced by the derived resolvable symbolic expression. Any mismatch among them means that there exists inconsistency among the reaching definitions of a use. Thus the base register, in this case, is marked as *unresolvable*. Figure 1-(b) shows an example of the above process for the example given in Figure 1-(a). In the example, the node at the root, $15 in this case, contains the base register for which a resolvable symbolic expression is to be derived. In Figure 1-(b), we can note that both of the two definitions reaching the use of $15 in `lw $25,12($15)` have an identical resolvable symbolic expression, $sp+48 in this case. Thus the `lw $25,12($15)` instruction is symbolically replaced with `lw $25,60($sp)` and afterwards it is regarded as a static load/store instruction.

4. Minimizing the WCET Overestimation Due to Dynamic Load/Store Instructions

In Section 2, we explained that the extended timing schema approach suffers from WCET overestimation due to dynamic load/store instructions. This section explains a technique for minimizing such overestimation. Before explaining the technique, let us consider, as an example of WCET overestimation, the C code fragment given in Figure 2-(a). In the example, we assume the following:

S_1

M(2)	M(19)	M(4)	M(21)
miss	miss	miss	miss

S_2	S_3	S_2	S_3	S_2	S_3	S_2	S_3
M(10)	M(12)	M(11)	M(13)	M(12)	M(14)	M(13)	M(15)
a[0][0]	a[1][0]	a[0][1]	a[1][1]	a[1][0]	a[2][0]	a[1][1]	a[2][1]
miss	miss	miss	miss	hit	miss	hit	miss

S_4

M(2)	M(3)	M(4)	M(5)
miss	miss	miss	miss

Figure 4. Actual cache hit/miss

1. S_1, which precedes the loop nest, references memory locations $M(2), M(19), M(4), M(21)$ using static load/store instructions where $M(k)$ represents the memory location that has k as its address.
2. Array `a[][]` that is referenced within the loop nest has the memory map shown in Figure 2-(b).
3. S_4, which succeeds the loop nest, references memory locations $M(2), M(3), M(4), M(5)$ using static load/store instructions.
4. The data cache is direct-mapped with eight cache blocks and its block size is equal to the size of an integer variable.

The extended timing schema approach assumes two cache miss penalties for each of the references generated by two array accesses `a[i][j]` and `a[i+1][j]` within the loop nest since these array accesses are implemented by dynamic load/store instructions and these references are completely ignored in the calculation of `first_reference` and `last_reference` of the statements (cf. Figure 3).

To compute the number of overestimated cache miss penalties, consider the actual execution shown in Figure 4, which gives the hit/miss of every data reference from the program fragment. The overestimation due to the array accesses can be computed from this hit/miss information. On one extreme, the reference to `a[0][0]` by S_2 does not suffer from any overestimation since the reference is a cache miss as predicted and, again as predicted, replaces from the cache a memory block that would otherwise lead to a cache hit (in this case $M(2)$). On the other extreme, the reference to `a[1][0]` by S_2 suffers from maximum overestimation corresponding to two cache miss penalties. One cache miss penalty overestimation is because the access is a cache hit, contrary to the prediction. The other cache miss penalty overestimation is because the replaced memory block is not a useful one, again contrary to the prediction. The overestimation caused by other accesses to array `a[][]` can be

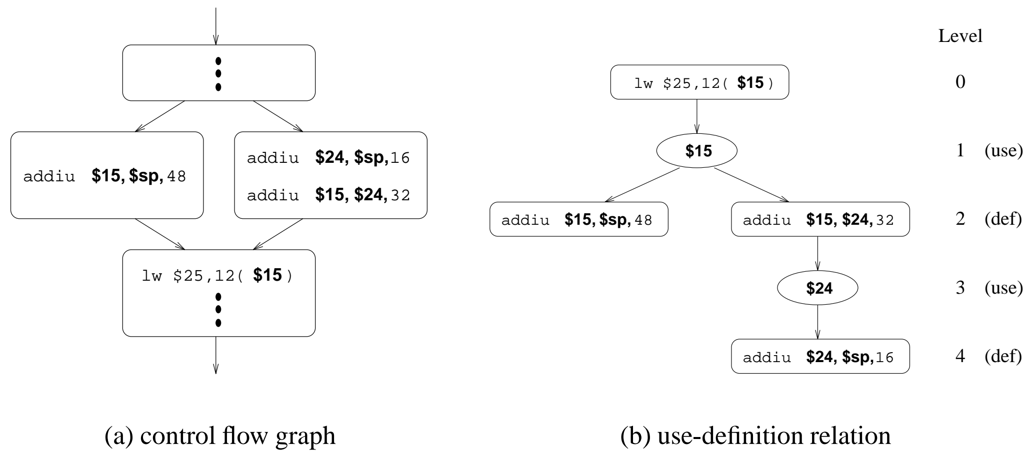

Figure 1. Tree showing use-define relationship for a base register

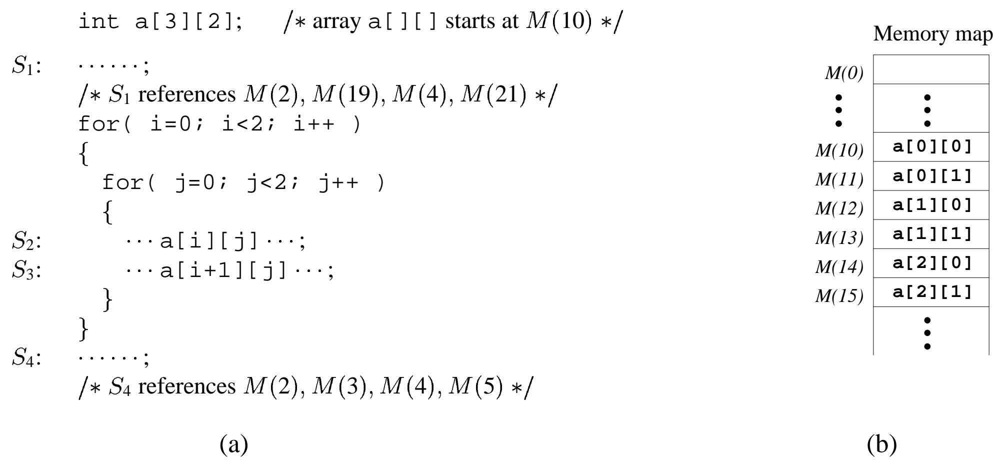

Figure 2. Example C code fragment

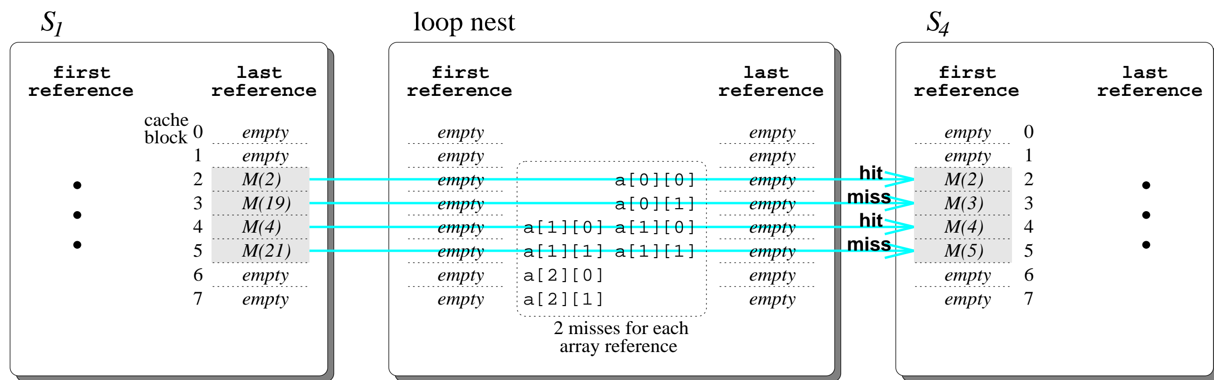

Figure 3. Cache hit/miss prediction by the extended timing schema approach

determined similarly and the results are given in Table 3. A total of eight cache miss penalties are unnecessarily assumed due to the references from the two dynamic load/store instructions implementing the two array accesses.

a[i][j]	# of overestimations
a[0][0]	0
a[0][1]	1
a[1][0]	2
a[1][1]	2

a[i+1][j]	# of overestimations
a[1][0]	0
a[1][1]	1
a[2][0]	1
a[2][1]	1

Table 3. Overestimation due to array accesses

Our technique for minimizing WCET overestimation due to dynamic load/store instructions is applied to each loop nest that is defined by an outermost loop in functions and proceeds as follows:

1. We identify the set of memory locations referenced by each dynamic load/store instruction in the loop nest.
2. We compute the union of the sets of memory locations referenced by the dynamic load/store instructions in the loop nest.
3. We invalidate in `last_reference` of the loop nest the set of cache blocks corresponding to the above union.
4. We derive a lower bound on the number of cache hits generated by dynamic load/store instructions in the loop nest and use this lower bound to tighten the WCET bound of the loop nest.

By invalidating in `last_reference` the cache blocks that are accessed by dynamic load/store instructions in Step 3, we no longer need to assume the one cache miss penalty that arises from the conservative assumption that a reference from a dynamic load/store instruction may replace a useful cache block; we just assume only one cache miss penalty for each reference from a dynamic load/store instruction.

The derivation of a lower bound on the number of cache hits in Step 4 is based on the *pigeonhole principle*, which is used by mathematicians to refer to the following simple observation [12]. If we put n objects into m boxes (pigeonholes), and if $n > m$, then some boxes inevitably have more than one object in them. In our analysis, the pigeonhole principle says that if two dynamic load/store instructions generate n_1 and n_2 references, respectively and they totally reference less than n_3 distinct locations, then at least $n_1 + n_2 - n_3$ references from the two dynamic load/store instructions are cache hits.

In our example in Figure 2, both n_1 and n_2 are 4, which is derived from the loop bounds of the two loops in the loop nest. n_3 is 6 since the distinct locations referenced by the two array accesses are `a[0][0]` through `a[2][1]`. This gives a lower bound of 2 ($= 4 + 4 - 6$) on the number of cache hits generated by the two array accesses. The WCET bound of the loop nest can be tightened by using this lower bound on cache hits.

To systematically derive a lower bound on the number of cache hits in this way, we should perform the following two tasks:

1. The region of references by each dynamic load/store instruction within a loop nest should be determined.
2. An upper bound on the number of distinct memory locations referenced by the set of dynamic load/store instructions in the loop nest should be derived.

The next two subsections detail the two tasks.

4.1. Specifying Reference Regions

This subsection explains how to derive the reference region of a dynamic load/store instruction. In general, a reference region is a multiset since duplicate reference addresses can be generated by a dynamic load/store instruction. Figure 5 shows the MIPS R3000 assembly code fragment corresponding to the C code fragment in Figure 2. The two shaded load instructions correspond to the two array accesses made in statements S_2 and S_3, respectively. The figure also shows the use-def tree for the base register \$15 used in `lw $15,0($15)`, which is for array access `a[i][j]`. The use-def tree for \$25 in `lw $25,0($25)` can be drawn similarly. From the use-def trees, reference regions R_1 and R_2 corresponding to `lw $15,0($15)` and `lw $25,0($25)`, respectively can be specified as follows:

$$\begin{array}{lcl} R_1 & : & 8 * \$13 + 4 * \$12 + \$\texttt{sp} \\ & & 0 \leq \$13 \leq 1,\ 0 \leq \$12 \leq 1 \\ R_2 & : & 8 * \$13 + 4 * \$12 + \$\texttt{sp} + 8 \\ & & 0 \leq \$13 \leq 1,\ 0 \leq \$12 \leq 1 \end{array} \quad (1)$$

In the above, the ranges of registers \$13 and \$12 are obtained from the loop bounds of the two loops in the loop nest.

In this paper, we restrict ourselves to reference regions that can be specified in the following form:

$$a_1 I_1 + a_2 I_2 + \ldots + a_m I_m + c,$$

where $I_1, \ldots, I_m$ are the registers for loop index variables and $a_i,\ 1 \leq i \leq m$, and c are constants. If a reference region cannot be specified in the above form, we conservatively assume two cache miss penalties for each reference from the associated load/store instruction.

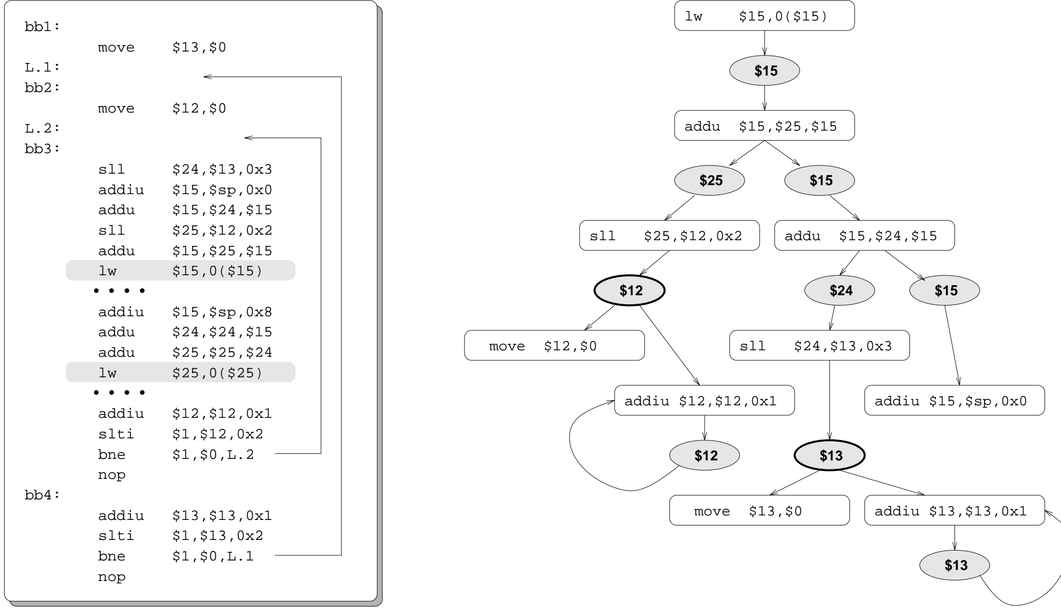

Figure 5. Assembly code and the use-def tree for `lw $15,0($15)`

4.2. Deriving a Lower Bound on Cache Hits

Once all the reference regions within a loop nest are determined, we can derive a lower bound on the number of cache hits generated by the reference regions. In our example, a lower bound on the number of cache hits generated by two reference regions R_1 and R_2 can be derived as follows:

$$\begin{aligned} MinHit &= n^\star(R_1) + n^\star(R_2) - n(R_1 \cup R_2) \\ &= (n(R_1) + s(R_1)) + (n(R_2) + s(R_2)) - n(R_1 \cup R_2) \\ &= s(R_1) + s(R_2) + (n(R_1) + n(R_2) - n(R_1 \cup R_2)) \\ &= s(R_1) + s(R_2) + n(R_1 \cap R_2), \end{aligned} \quad (2)$$

where $n^\star(R)$ is the number of elements in R (with duplication allowed), $n(R)$ the number of distinct elements in R, and $s(R)$ the number of repeated elements in R. $s(R)$ can be computed by the following equation:

$$s(R) \;=\; \prod_{I \in \mathbf{C}^1_R} loop_bound_I \;-\; \prod_{I \in \mathbf{C}^2_R} loop_bound_I,$$

where $\mathbf{C}^1_R$ is the set of the index variables of the loops nesting the corresponding load/store instruction of the reference region R, $\mathbf{C}^2_R$ the set of the index variables that appear in the reference region expression, and $loop_bound_I$ the loop bound of the loop that has I as its index variable. In our example, $s(R_1) = s(R_2) = 0$ since in both of the two reference regions R_1 and R_2 all the index variables appear in the two reference region expressions. $n(R_1 \cap R_2)$ is given by the cardinality of the solution space ($\$13_1, \$13_2, \$12_1, \$12_2$) satisfying the following equation.

$$8 * \$13_1 + 4 * \$12_1 = 8 * \$13_2 + 4 * \$12_2 + 8, \quad (3)$$

$$\begin{aligned} 0 \le \$13_1 \le 1, \quad 0 \le \$12_1 \le 1, \\ 0 \le \$13_2 \le 1, \quad 0 \le \$12_2 \le 1. \end{aligned} \quad (4)$$

This diophantine equation[4] is derived by equating the two reference expressions in (1). The equation (3) can be expressed in a matrix form as follows:

$$\begin{pmatrix} \$13_1 & \$12_1 & \$13_2 & \$12_2 \end{pmatrix} \begin{pmatrix} 8 \\ 4 \\ -8 \\ -4 \end{pmatrix} = \begin{pmatrix} 8 \end{pmatrix} \quad (5)$$

This equation can be solved using the *generalized GCD test* [3]. The test gives the following solution space represented by integer parameters, $\mathbf{t}$=(t_1,t_2,t_3).

$$\begin{aligned} \$13_1 = t_1, \quad \$12_1 = -2t_1 + t_2 + 2, \\ \$13_2 = t_3, \quad \$12_2 = t_2 - 2t_3. \end{aligned} \quad (6)$$

We use the *Fourier elimination method* [3] to compute the cardinality of the solution space **t** satisfying both (4) and (6). This gives $n(R_1 \cap R_2)$, which is 2 in our example. This, in turn, gives $MinHit = 2 \; (= 0 + 0 + 2)$ and this value is used to tighten the WCET bound of the loop nest.

[4] An equation in integer variables is called a diophantine equation.

So far we considered only the case where there are two reference regions. The equation (2) can be generalized for the case where there are r reference regions in a loop nest.

$$
\begin{aligned}
MinHit = & \sum_{i=1}^{r} s(R_i) \\
& + \sum_{<R_{i_1},R_{i_2}>\in \mathbf{C}_2} n(R_{i_1} \cap R_{i_2}) \\
& - \sum_{<R_{i_1},R_{i_2},R_{i_3}>\in \mathbf{C}_3} n(R_{i_1} \cap R_{i_2} \cap R_{i_3}) \\
& \vdots \\
& + (-1)^r \sum_{<R_{i_1},\ldots,R_{i_r}>\in \mathbf{C}_r} n(R_{i_1} \cap \cdots \cap R_{i_r}),
\end{aligned}
$$

where $\mathbf{C}_k,\ 2 \le k \le r$, is the set of all combinations consisting of k reference regions. In this equation, $n(R_{i_1} \cap \cdots \cap R_{i_{k+1}})$ can be computed from $n((R_{i_1} \cap \cdots \cap R_{i_k})\ \cap (R_{i_2} \cap \cdots \cap R_{i_{k+1}}))$ where $R_{i_1} \cap \cdots \cap R_{i_k}$ and $R_{i_2} \cap \cdots \cap R_{i_{k+1}}$ are by-products from the computation of $n(R_{i_1} \cap \cdots \cap R_{i_k})$ and $n(R_{i_2} \cap \cdots \cap R_{i_{k+1}})$. For example, $n(R_1 \cap R_2 \cap R_3)$ can be computed from the intersection of $R_1 \cap R_2$ and $R_2 \cap R_3$. In calculating the intersection, $R_1 \cap R_2$ and $R_2 \cap R_3$ are regarded as if they were simple reference regions.

In our previous discussion, we assume that disjoint reference regions do not conflict with each other in the cache. In the case where they do conflict in the cache, we partition the reference regions and disregard from the calculation of $MinHit$ the subregions that conflict in the cache. In the case where the references from a static load/store instruction conflict with reference regions within the same loop nest, we assume two cache miss penalties for each reference of the static load/store instruction and process the reference regions as previously.

4.3. Overall Framework

To summarize, the processing of a loop nest within the proposed framework proceeds as follows:

Step 1. Mark invalid in `last_reference` the cache blocks corresponding to the union of the reference regions in the loop nest.

Step 2. Compute the $MinHit$ for the loop nest and use it to tighten the WCET bound of the loop nest.

To illustrate the above procedure, we consider again the example in Figure 2. Figure 6 shows Step 1 of the procedure. We explicitly invalidate cache blocks in the loop nest's `last_reference` that correspond to the union of two reference regions. For each of the eight array references from the two reference regions in the loop nest, we assume only one cache miss penalty in this revised procedure as compared to two cache miss penalties in the previous approach as shown in Figure 3. Using this method, we can eliminate 6 overestimated cache misses. However, we still have in the resultant WCET bound of the loop nest two overestimated cache misses which come from the fact that we have ignored cache hits due to repeated references from the two dynamic load/store instructions to the same memory locations, *i.e.*, `a[1][0]` and `a[1][1]`. To eliminate such overestimation we use $MinHit$, which is 2 as we showed earlier. This $MinHit$ information is used to tighten the WCET bound of the loop nest obtained in Step 1. Overall, the revised procedure eliminates all the eight overestimated cache misses that the previous approach suffered from.

For the simple example above, we have no overestimated data cache misses due to references from dynamic load/store instructions. In general, however, we may not completely eliminate WCET overestimation due to dynamic load/store instructions because of the following.

- We assume two cache miss penalties for each pointer-based dynamic reference.
- We assume two cache miss penalties for each reference from a static load/store instruction that conflicts in the cache with one or more reference regions in the loop nest.

5. Experimental Results

To assess the effectiveness of the proposed techniques, we compare the WCET bounds of several benchmark programs predicted with and without applying the techniques using the timing tool explained in [11]. The target machine of the timing tool is an IDT7RS383 board. The target machine's CPU is a 20 MHz R3000 processor [8] which is typical of a RISC processor. The machine has instruction and data caches of 16 Kbytes each. Both caches are direct-mapped and have block sizes of 4 bytes. The cache miss service times of both the instruction and data caches are 4 cycles.

Five simple benchmark programs were used: *Arrsum*, *Fib*, *Isort*, *MM*, and *Sqrt*. *Arrsum* calculates the total sum of 10 integer array elements. *Fib* computes the 30th element of the Fibonacci sequence. *Isort* sorts 10 integer array elements using the insertion sort algorithm. *MM* multiplies two 5×5 integer matrices. *Sqrt* performs the square root computation on an integer number.

Table 4 compares the measured execution times and the WCET bounds predicted by the timing tool for the five benchmark programs. For each benchmark program, three WCET predictions were made. $Pred_{P+I}$ considers only the effects of pipelined execution and instruction caching treating each data reference as a cache miss. $Pred_{P+I+D}^{no\ opt}$

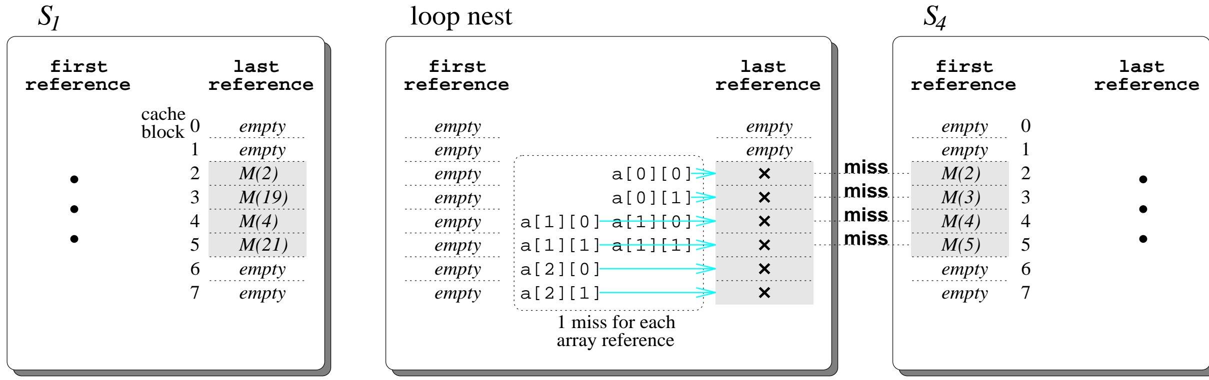

Figure 6. Cache hit/miss prediction by the extended timing schema approach when we apply the proposed technique

	Arrsum	*Fib*	*Isort*	*MM*	*Sqrt*
$Measured$	242	683	2849	9141	302
$Pred_{P+I}$	256	710	6709	11653	327
$Pred_{P+I+D}^{no\ opt}$	296	710	8077	13153	327
$Pred_{P+I+D}^{opt}$	**256**	710	**6517**	**10453**	327

(unit: machine cycles)

Table 4. Measured and predicted execution times of the benchmark programs

considers data caching effects in addition to the effects of pipelined execution and instruction caching but without applying the techniques explained in this paper. As we mentioned earlier, in this prediction, the timing tool assumes two cache miss penalties for each data reference generated from a dynamic load/store instruction. Finally, $Pred_{P+I+D}^{opt}$ uses the proposed techniques while performing data caching analysis. For all the three predictions, the timing effects of pipelined execution and instruction caching are analyzed by the same analysis technique explained in [11].

One interesting point from the results is that $Pred_{P+I+D}^{no\ opt}$ (which considers all the aspects of the target machine including the data caching effects) yields looser WCET predictions than $Pred_{P+I}$ (which treats all data references as cache misses) for *Arrsum*, *Isort*, and *MM*. This rather anomalous result indicates the adverse impacts of dynamic load/store instructions. The three benchmark programs have a large number of data references from dynamic load/store instructions due to a large number of array references and the timing tool assumes two cache miss penalties for such data references in the case of $Pred_{P+I+D}^{no\ opt}$. On the other hand, in the case of $Pred_{P+I}$, the timing tool assumes only one cache miss penalty for such data references by treating all the data references as cache misses including those from static load/store instructions. Therefore, when data references from dynamic load/store instructions are more than half of the total data references, $Pred_{P+I+D}^{no\ opt}$ yields a looser prediction than $Pred_{P+I}$. Such a condition holds for the three benchmark programs and, thus, $Pred_{P+I+D}^{no\ opt}$ yields looser predictions than $Pred_{P+I}$.

This rather anomalous behavior is cured by applying the proposed techniques. The boldfaced results in Table 4 for the three benchmark programs that previously exhibited the anomalous behavior now show significant improvements. Most of the improvements come from by applying the second technique since the three benchmark programs suffer from WCET overestimation in $Pred_{P+I+D}^{no\ opt}$ resulting from a large number of data references from dynamic load/store instructions.

The *Fib* and *Sqrt* benchmark programs do not contain any dynamic load/store instructions and all the data references from static load/store instructions are predicted to miss in the cache both in $Pred_{P+I+D}^{no\ opt}$ and $Pred_{P+I+D}^{opt}$. Thus there is no difference among $Pred_{P+I}$, $Pred_{P+I+D}^{no\ opt}$ and $Pred_{P+I+D}^{opt}$.

For all the benchmark programs except for *Isort*, $Pred_{P+I+D}^{opt}$ gives a very tight WCET bound as compared with the measured execution time. The WCET overestimation in the *Isort* benchmark is caused by execution paths that are infeasible in a real execution but considered in the WCET prediction [15], which, we think, is an issue orthogonal to the proposed techniques.

6. Conclusions

This paper has proposed two techniques for worst case timing analysis of data caching. Our particular focus was

on dynamic load/store instructions for which most current timing analysis techniques take very conservative approaches. The first technique aims at reducing the number of load/store instructions that are misclassified as dynamic load/store instructions. For this purpose, we make use of a global data flow analysis technique. The second technique tries to minimize WCET overestimation resulting from dynamic load/store instructions. The purposes of the second technique are twofold. First, it reduces WCET overestimation arising from the conservative assumption about dynamic load/store instructions that each reference from them may replace a useful cache block (i.e., a cache block that would otherwise lead to a cache hit). The reduction of WCET overestimation was made possible by invalidating in `last_reference` of the containing loop nest the cache blocks referenced by dynamic load/store instructions. Second, the technique derives a lower bound on the number of cache hits generated by dynamic load/store instructions and uses this lower bound to tighten the WCET bound.

Results from a preliminary evaluation study have shown that the two techniques significantly improve the tightness of WCET bounds. The improvement was most noticeable for programs that make heavy use of arrays, which are a main source of dynamic load/store instructions.

The current derivation of a lower bound on the number of cache hits due to dynamic load/store instructions is restricted to cache hits that are made within individual loop nests. One direction for future research is to derive a similar bound on the number of cache hits that are made across loop nests. This requires data dependence analysis between reference regions that belong to different loop nests. Furthermore, to determine the number of cache hits across two reference regions belonging to two different loop nests, we have to consider the set of memory references that come between the two reference regions. Another future research direction is to extend the second technique to handle the case where the cache block size is larger than one word. Many computer systems today use large cache block sizes to exploit spatial locality in programs and we expect that the above extension will enhance the applicability of the technique.

Acknowledgements

The authors wish to thank anonymous referees for their constructive comments. The authors also thank C. Y. Park, M. Lee, and S. Hong for many helpful discussions.

This work was supported in part by KOSEF (Grant KOSEF-93-01-00-06).

References

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

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proceedings of the 15th Real-Time Systems Symposium*, pages 172–181, 1994.

[3] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, 1993.

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

[5] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 288–297, 1995.

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

[7] Y. Hur, Y. H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 308–319, 1995.

[8] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

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

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 298–307, 1995.

[11] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. In *Proceedings of the 15th Real-Time Systems Symposium*, pages 97–108, 1994.

[12] P. Linz. *An Introduction to Formal Languages and Automata*. D. C. Heath and Company, Lexington, MA, 1990.

[13] J.-C. Liu and H.-J. Lee. Deterministic Upperbounds of the Worst-Case Execution Times of Cached Programs. In *Proceedings of the 15th Real-Time Systems Symposium*, pages 182–191, 1994.

[14] 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.

[15] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Journal of Real-Time Systems*, 5(1):31–62, March 1993.

[16] 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.

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