A Worst Case Timing Analysis Technique for Multiple-Issue Machines*

Sung-Soo Lim
Dept. of Computer Engineering
Seoul National University
Seoul, Korea, 151-742

Jung Hee Han
Dept. of Electrical Engineering and Computer Science
University of Michigan, 1301 Beal Ave.
Ann Arbor, MI 48109-2122

Jihong Kim†
Dept. of Computer Science
Seoul National University
Seoul, Korea, 151-742

Sang Lyul Min
Dept. of Computer Engineering
Seoul National University
Seoul, Korea, 151-742

Abstract

We propose a worst case timing analysis technique for in-order, multiple-issue machines. In the proposed technique, timing information for each program construct is represented by a directed acyclic graph (DAG) that shows dependences among instructions in the program construct. From this information, we derive for each pair of instructions the distance bounds between their issue times. Using these distance bounds, we identify the sets of instructions that can be issued at the same time. Deciding such instructions is an essential task in reasoning about the timing behavior of multiple-issue machines. In order to reduce the complexity of analysis, the distance bounds are progressively refined through a hierarchical analysis over the program syntax tree in a bottom-up fashion. Our experimental results show that the proposed technique can predict the worst case execution times for in-order, multiple-issue machines as accurately as ones for simpler RISC processors.

1. Introduction

In building a real-time system, the worst case execution times (WCETs) of tasks in the system should be predicted in advance since they are required in schedulability analysis for the system. Results of the WCET prediction should be both *safe* (i.e., the predicted WCET should not be smaller than the real WCET) and *accurate* (i.e., the difference between the predicted WCET and the real WCET should be small). Unsafe prediction causes unexpected deadline misses of tasks that may result in catastrophic consequences. On the other hand, inaccurate prediction leads to a pessimistic schedulability analysis that results in underutilization of system resources.

To obtain accurate prediction for modern high-performance processors, the timing effect of advanced architectural features should be taken into account. For example, several groups including Zhang *et al.*[15], Lim *et al.*[11], Li *et al.*[10], and Healy *et al.*[4] had investigated the prediction techniques for pipelined processors. However, most of existing techniques assume that processors can issue at most one instruction at each cycle, thus cannot produce accurate analysis results for modern multiple-issue machines such as superscalar processors.

In this paper, we propose a worst case timing prediction technique which is applicable to multiple-issue machines. In reasoning about the timing behavior of multiple-issue machines, it is essential to identify the instructions that can be issued at the same time. In the proposed technique, the timing information for each program construct is represented by a directed acyclic graph called an instruction dependence graph (IDG). The IDG shows dependences among instructions in a program. From the IDG, the distance bounds between the issue times of every pair of instructions are derived. Using these distance bounds, we can identify the sets of instructions that can be issued simultaneously. Deriving the distance bounds and identifying simultaneously-issued instructions are two key steps in predicting the WCET of a program on multiple-issue machines.

To reduce the complexity of analysis, our approach is based on an existing hierarchical timing analysis framework called the extended timing schema (ETS) [11]. In this framework, the distance bounds are progressively refined

*This work was supported in part by KOSEF under Grant 97-01-02-05-01-3.

†Jihong Kim was supported in part by Equipment Award for New Faculty from the College of Natural Sciences, Seoul National University.

over the program syntax tree in a bottom-up fashion. As the distance bounds of surrounding blocks are known during the hierarchical refinements, some adjacent instructions can be merged into a single node of the IDG if they have a constant distance. This merging step reduces the number of nodes maintained by the IDG, thus reducing the complexity of analysis.

The proposed technique is described and validated using a simple in-order, multiple-issue machine model. Since the primary purpose of the work described in this paper is to understand whether the multiple-issue feature can be accurately analyzed to predict the WCETs of programs, we significantly simplify a machine model except for the multiple-issue capability. (This model is described in detail in Section 3.) For the validation purpose, we build a timing tool based on the proposed technique using the multiple-issue machine model, and compare the WCET analysis results from the tool with measurements obtained using simulation. The results show that our technique can predict the WCETs for in-order, multiple-issue machines as accurately as ones for simple RISC processors based on the similar hierarchical technique [11].

The rest of this paper is organized as follows. In Section 2, we explain the ETS that forms the basis of the technique proposed in this paper. In Section 3, we describe our multiple-issue machine model used in this paper. An IDG is formally defined as well in this section. Section 4 presents the key algorithms to derive the distance bounds and to identify the instructions that can be issued simultaneously. In Section 5,we explain how to augment the original ETS framework for the multiple-issue machine model using the distance bounds computed. Experimental results follow in Section 6, and we conclude with future works in Section 7.

2. Extended Timing Schema

Before we discuss the proposed technique, we first describe the extended timing schema (ETS) on which our algorithms are based. The original timing schema is a set of formulas for reasoning about the timing behavior of various language constructs [14]. The ETS extends the timing schema to reflect timing properties of modern architectural features such as pipelining and caching in two aspects: (1) redefinition of the timing information for each program construct and (2) redefinition of the timing formulas using newly introduced operations, concatenation ($\oplus$) and pruning [11].

In the ETS, each program construct is associated with a Worst Case Timing Abstract (WCTA) which is a data structure containing information needed in hierarchical timing analysis. A program construct may have more than one execution path as an **if** statement and the WCETs of these execution paths differ significantly depending on preceding program constructs. Therefore, the worst case execution path of the program construct cannot be determined by analyzing the program construct without considering the preceding program constructs. For this reason, the WCTA of a program construct keeps timing information of every execution path in the program construct that *might* be the worst case execution path. Each execution path of a program construct forms a Path Abstraction (PA) that contains the timing-related information of the execution path. The PA of an execution path encodes the factors that affect the WCET of the execution path. These factors may include the information on the pipelined execution and cache states. The encoding is done in such a way that allows for refinement of the execution path's WCET when the detailed information about the preceding execution paths becomes available. For example, in the case of pipelined execution analysis, the PA is about the use of pipeline stages in the associated execution path. In [11], the PA structure is defined using *reservation tables* which represent the usage of pipeline stages. This information allows for the refinement of the path's execution time once the pipeline usage information of preceding execution paths becomes available.

	Extended Timing Schema
S: S_1; S_2	$W(S) = W(S_1) \oplus_p W(S_2)$
S: if (exp) then S_1 else S_2	$W(S) = (W(exp) \oplus_p W(S_1)) \cup (W(exp) \oplus_p W(S_2))$
S: while (exp) S_1	$W(S) = (\oplus_{i=1}^{N} (W(exp) \oplus_p W(S_1))) \oplus_p W(exp)$
S: f(exp$_1$, . . . , exp$_n$)	$W(S) = W(exp_1) \oplus_p \ldots \oplus_p W(exp_n) \oplus_p W(f())$

$W(S)$ is the WCTA of a statement **S**. The $\oplus$ operation between two WCTAs, W_1 and W_2, is defined as $W_1 \oplus W_2 = \{w_1 \oplus w_2 | w_1 \in W_1, w_2 \in W_2\}$. The $\oplus_p$ operation is the $\oplus$ operation followed by the pruning operation, and N is the upper bound on the number of the loop iterations.

Table 1. Timing formulas of the extended timing schema.

Two new operations, concatenation ($\oplus$) and pruning, are introduced to form the timing formulas in the ETS. 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 execution times of both paths are revised using each other's timing information encoded in their PAs. The pruning operation is performed on a set of PAs for a program construct and prunes the PAs whose associated execution paths cannot be the worst case execution path of the program construct. Table 1 shows the timing formulas of the extended timing schema. The timing formula for **S: S_1; S_2** first enumerates all the possible execution paths within **S**. The pruning operation after the enumeration prunes a subset of the resulting execution paths that cannot be the worst case execution path of the sequential statement. Similarly, the timing formula for an **if** statement first enumerates all the execution paths in the **then** path and

those in the **else** path, and 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 a loop unrolled N times. This approach is exact but is computationally intractable for a 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].

In summary, in order to perform timing analysis within the ETS framework, the following three components should be decided: (1) determination of the PA structure, (2) definition of the $\oplus$ operation on PAs, and (3) definition of the pruning condition. Once these three components are determined, the worst case timing analysis of a program can be performed mechanically by traversing the program's syntax tree in a bottom-up fashion and applying the timing formulas of the ETS.

3. Multiple-Issue Machine Model and Program Representation

3.1. Multiple-Issue Machine Model

Throughout this paper, we use a simple multiple-issue machine model with the following assumptions:

- Instructions are issued in the order shown in a program (in-order issue). However, each instruction can complete its execution regardless of other instructions' completion (out-of-order completion).
- The delayed branch scheme is used. For a branch instruction, one delay slot is assumed. This assumption is to ignore the effect of the branch prediction results on the execution time. (A worst case timing analysis under branch prediction is one of our main future research topics.)
- All memory accesses are cache hits. This assumption is to ignore the effect of cache misses on the execution time. (Since our approach does not make any assumption on the cache system, previously proposed techniques on the cache analysis [1, 11, 10] can easily be integrated with our approach.)
- Processor can issue up to k instructions simultaneously, where k is provided as a parameter to our analysis. k is equal to or smaller than the number of functional units in the processor (i.e., $k \leq 9$ in our processor model).

We further assume that our machine model has nine functional units as shown in Figure 1 and its instruction set is similar to that used in MIPS R3000/R3010 [8]. In Figure 1, each functional unit is shown with the latency of operations performed by the corresponding functional unit. Both the Load/Store and Branch units are assumed to have one delay slot.

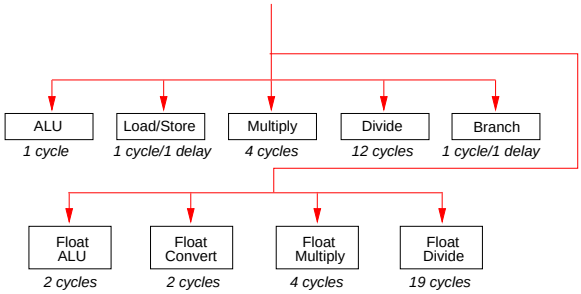


Figure 1. Functional units of target processor model.

3.2. Program Representation

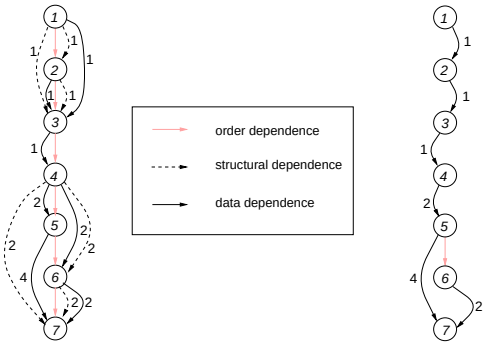
In multiple-issue processors, the amount of exploitable parallelism is restricted by dependences among instructions. Such dependences include *structural dependences* (i.e., resource conflicts) and *data dependences* [7][1]. Structural dependences are caused by multiple instructions competing for the same functional unit. On the other hand, data dependences are caused by multiple instructions that use the same register or memory location. In this paper, we introduce another type of dependences, *order dependences*, that are used to model that instructions are issued in-order in our machine model.

To represent dependences among instructions, we use a directed acyclic graph (DAG), which we call an *instruction dependence graph* (IDG). The IDG representation of a program is similar to popular DAG representation of a program used in compiler research [3, 5]. In an IDG, a node can represent either a single instruction or a sequence of instructions. In the proposed technique, each node of an IDG initially corresponds to a single instruction in a basic block. Through a hierarchical refinement process of our timing analysis technique, several nodes are merged into a single node which represents more than one instruction. Each edge $e = (i, j)$ of an IDG indicates that there is at least one *non-redundant* dependence between the instruction(s) represented by the node i and the instruction(s) represented by the node j. (The exact definition of *redundant* dependences is described below.) Each edge $e = (i, j)$ is associated with a weight ℓ_e (or $\ell_{i,j}$) that represents the minimum cycles

[1] There is another type of dependences called *control dependences*. Control dependences are caused by branch instructions. Since we assume the delayed branch scheme in which a compiler resolves control dependences, we do not consider control dependences in our machine model.

```
1  sll    $25, $12, 0x2       ; $25 <- $12 << 2
2  lui    $1, 0x1000          ; $1  <- 0x1000
3  addu   $1, $1, $25         ; $1  <- $1 + $25
4  lwc1   $f18, 96($1)        ; $f18 <- mem[$1 + 96]
5  mul.s  $f16, $f18, $f18    ; $f16 <- $f18 * $f18
6  add.s  $f18, $f18, $f18    ; $f18 <- $f18 + $f18
7  add.s  $f18, $f18, $f16    ; $f18 <- $f18 + $f16
```

(a)



(b) (c)

Figure 2. An example program representation.

required between the issues of the first instructions of the nodes i and j. (For the description convenience, we use the *issue of a node* to mean the issue of the first instruction of a node.)

As an example, consider the basic block shown in Figure 2(a). The basic block contains a sequence of instructions based on the MIPS R3000/R3010 instruction set. Among the instructions, `sll`, `lui`, and `addu` use the ALU functional unit, `lwc1` and `add.s` use the Float ALU functional unit, and `mul.s` uses the Float Multiply functional unit. The graph shown in Figure 2(b) represents all the dependences among the instructions including redundant dependences. In the dependence graph, since the weights for order dependences are all 0, they are not shown explicitly in the figure. Formally, we define that a dependence $e = (v, w)$ between nodes v and w is redundant if one of the following two conditions is satisfied:

(1) there exists another dependence $e' = (v, w)$ between the same nodes v and w and $\ell_{e'} \geq \ell_e$ (when there are more than one dependence with the same weight between the two nodes, we choose one arbitrarily among them),

(2) there exists a set of dependences, $S = \{e_0, e_1, \ldots, e_p\}$, such that

(i) $e_0 = (v, i_0), e_p = (i_{p-1}, w)$, and $e_j = (i_{j-1}, i_j)$ for $1 \leq j < p$ and

(ii) $\sum_{i=0}^{p} \ell_{e_i} \geq \ell_e$.

For example, because of rule (2), both of the two dependences between nodes 4 and 6 in Figure 2(b) are redundant. Figure 2(c) shows the corresponding IDG with the redundant dependences eliminated. In an IDG, since all redundant dependences were eliminated, there can be at most one edge between any pair of nodes.

An IDG succinctly represents the timing-related information for a sequence of instructions. However, an IDG alone is generally not sufficient to predict the WCETs. While an IDG contains the complete information on the dependence relationship between the instructions represented by the IDG, more information from the surrounding blocks is necessary to predict the worst case issue patterns that result in the WCETs. For example, if the first instruction of the sequence requires the Float Divide unit, its issue time can vary significantly depending on whether the Float Divide unit is used by the preceding instruction. This type of variation on the issue times makes it difficult to identify the instructions that can be issued simultaneously. To efficiently manage these difficulties, we divide IDG nodes into two types. For the first type of a node (called a *resolved node*), each instruction represented by the node knows which instructions *in the same IDG* use its functional unit and operands. For the second type of a node (called an *unresolved node*), at least one instruction in the node cannot tell when its functional unit or operands may be used, simply by examining the preceding instructions in the same IDG. In our proposed technique, for unresolved nodes, we assume that their executions may be delayed by the maximum possible delay for safe prediction. For accurate prediction, the maximum delay value is reduced as more information from surrounding blocks are known through the hierarchical refinement process. Formally, we define the resolved and unresolved nodes as follows:

Definition 1 A node is said to use a register if the register is one of the source register(s) of an instruction in the node. Let *use(i)* for a node i be the set of registers used by the instruction(s) in the node i.

Definition 2 A node is said to define a register if the register is the destination register of an instruction in the node. Let *def(i)* for a node i be the set of registers defined by the instruction(s) in the node i.

Definition 3 A node is said to occupy a functional unit if an instruction in the node utilizes the functional unit during its execution. Let *occupy(i)* for a node i be the set of functional units occupied by the instruction(s) in the node i.

Definition 4 A node i is said to be resolved if all the registers in *use(i)* and *def(i)* are defined by preceding nodes in

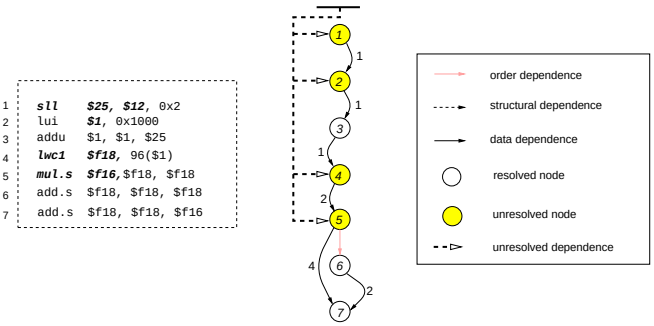


Figure 3. An IDG with resolved/unresolved nodes.

the same IDG and all the functional units in *occupy(i)* are occupied by preceding nodes in the same IDG. Otherwise, the node is said to be unresolved. If a node is resolved, we say that all the dependences for the node are resolved. If a node is unresolved, we say that the node has unresolved dependences.

Figure 3 shows the same IDG used in Figure 2 with the resolved/unresolved information augmented. As an example of the unresolved nodes, consider node 4 in the IDG. We can note that $use(4)$ = { `$1` }, $def(4)$ = { `$f18` }, and $occupy(4)$ = { Float ALU } according to Definitions 1, 2, and 3. Among the elements in $use(4)$ and $def(4)$, register `$f18` is not defined by nodes 1, 2, and 3. The functional unit Float ALU in $occupy(4)$ is also not occupied by any of the preceding three nodes 1, 2, and 3. Therefore, node 4 is an unresolved node which has unresolved dependences for register `$f18` and the Float ALU functional unit. The dotted thick edge from outside of the IDG to node 4 in Figure 3 represents the unresolved dependence. Similarly, nodes 1, 2, and 5 are unresolved nodes as well due to unresolved dependences for registers or functional units.

4. Distance Bounds between Instructions

In this section, we first describe an algorithm for deriving the distance bounds on the issue times of each pair of instructions in a given IDG. (We call this *a bounding step*.) These distance bounds are used to identify the instructions that can be simultaneously issued. (We call this *a multiple issuing step*.) Once the distance bounds are computed for the IDG and the instructions are identified for multiple issues, the IDG can be simplified. (we call this *a merging step*.) For example, if two adjacent instructions have a constant issue distance, without loss of information, we can combine these two instructions into a single IDG node. The merging step reduces the complexity of analysis by reducing the number of nodes that need to be kept in an IDG.

4.1. Definitions and Terminology

We use the following terms and notations in explaining the algorithms:

- We use the term *resources* to refer to both registers and functional units. Let $\mathcal{R} = \{r_1, r_2, \ldots, r_{N_R}\}$ be the set of resources in a processor model where N_R is the total number of resources. N_R is the sum of the number of registers and the number of functional units. (In our processor model, N_R is 73, consisting of 64 registers and 9 functional units.)

- Each resource r_i is associated with the maximum latency z_i, which is defined to be the maximum duration for which an instruction may occupy the resource r_i. If r_i is a functional unit, z_i is the maximum instruction latency among all the instructions that occupy r_i during their executions. Let z_{max} be the maximum z_i for all the functional units. If an r_i is a register, z_i is defined as z_{max}. For a given processor, z_i's can be obtained from the processor manual. In our processor model, z_{max} is 19 cycles which is the latency of instructions for the Float Divide functional unit.

- For an unresolved node i in an IDG, let $unresolved(i)$ be the set of resources that are not resolved in node i. The $unresolved(i)$ set consists of the registers and the functional units that are not resolved in node i. The unresolved registers are the registers in $use(i)$ or $def(i)$ but not in $def(j)$ for $j < i$ while the unresolved functional units are the functional units in $occupy(i)$ but not in $occupy(j)$ for $j < i$.

- For each node i, $max\_latency\_node(i)$ is the time duration from the issue of the first instruction of node i to the completion of the executions of all the instructions in the node i while $min\_latency\_node(i)$ is the time duration from the issue of the first instruction of node i to the issue of the last instruction of node i. $Max\_latency\_node(i)$ is used to assign the maximum possible weight of an edge from node i to a succeeding node and $min\_latency\_node(i)$ is to assign the minimum possible weight of an edge from node i to a succeeding node. For each node i, $max\_latency\_unresolved(i)$ is the maximum of the maximum latencies of the resources in $unresolved(i)$. $Max\_latency\_unresolved(i)$ is used to assign the maximum possible weight of an edge from a preceding node to node i.

4.2. Bounding Step

In order to represent the variation of instruction issue times, we derive the distance bounds between the issue times

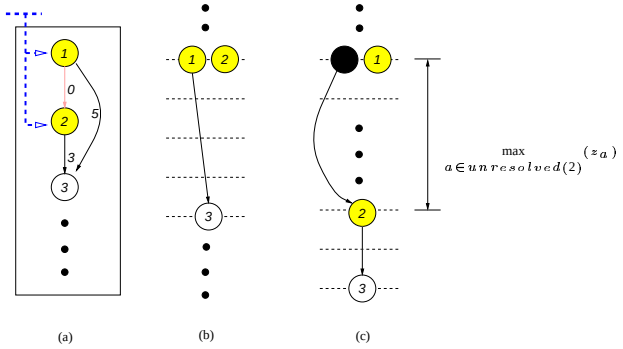


Figure 4. A bounding step example.

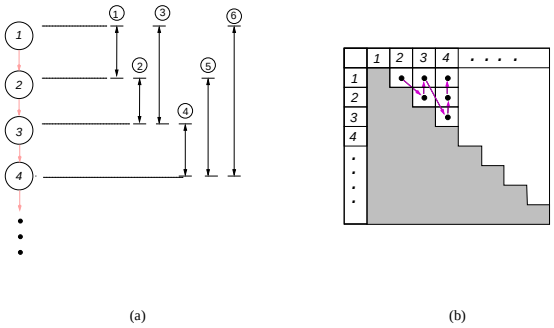


Figure 5. The bounding order.

for each pair of nodes. As an example, consider the IDG shown in Figure 4(a) where nodes 1 and 2 have unresolved dependences. In order to compute the distance bounds for each pair of nodes, there are three combinations to consider: the distance bounds for nodes 1 and 2, nodes 2 and 3 and nodes 1 and 3. The lower bound for nodes 1 and 2 occurs when node 2 is issued at the same time as node 1 as shown in Figure 4(b). Such a scenario is possible because node 2 has only an order dependence with node 1. Thus, the lower bound for nodes 1 and 2 is 0. On the other hand, the upper bound occurs when the issue of node 2 is delayed as much as possible from the issue of node 1. Such a scenario is shown in Figure 4(c). In the figure, the black node represents the node that immediately precedes the current program construct. In this scenario, the black node is assumed to have only an order dependence with node 1, so it can be issued at the same time as node 1. Furthermore, we assume that the black node occupies the unresolved resources of node 2 making the unresolved dependences of node 2 be forced to be resolved. To give the upper bound, this unresolved dependence should have the largest possible latency, $\max_{a \in unresolved(2)}(z_a)$, where a is an element of the set of unresolved resources for node 2.

Next, consider the bounds on the distance between nodes 2 and 3. This distance is affected by three weights, $\ell_{2,3}$, $\ell_{1,3}$, and $\ell_{1,2}$. The lower bound can be obviously $\ell_{2,3}$ (= 3), but it can be tightened more. For instance, if the upper bound of the distance between nodes 1 and 2 is less than $(\ell_{1,3} - \ell_{2,3})$(= 2), the distance between nodes 2 and 3 is guaranteed to be larger than $\ell_{2,3}$ (= 3). Therefore, the lower bound is the larger of (1) $\ell_{2,3}$ and (2) the difference between $\ell_{1,3}$ and the upper bound of the distance between nodes 1 and 2. Similarly, the upper bound is the larger of (1) $\ell_{2,3}$ and (2) the difference between $\ell_{1,3}$ and the lower bound of the distance between nodes 1 and 2. These comparisons require the distance bounds between nodes 1 and 2 to be calculated in advance.

Finally, consider the distance bounds for nodes 1 and 3. The lower bound can be obviously $\ell_{1,3}$ (= 5). However, if the distance between nodes 1 and 2 is guaranteed to be larger than $(\ell_{1,3} - \ell_{2,3})$(= 2), the distance between nodes 1 and 3 should be computed by summing the distance between nodes 1 and 2 and $\ell_{2,3}$(= 3). Therefore, the lower bound for nodes 1 and 3 is the larger of (1) $\ell_{1,3}$ and (2) the sum of the lower bound of the distance between nodes 1 and 2 and $\ell_{2,3}$. Similarly, the upper bound on the distance between nodes 1 and 3 is the larger of (1) $\ell_{1,3}$ and (2) the sum of the upper bound of the distance between nodes 1 and 2 and $\ell_{2,3}$.

In calculating the distance bounds, some distance bounds must be computed earlier than others. For example, deriving the distance bounds for nodes 2 and 3 requires the distance bounds for nodes 1 and 2 to be available in advance. Similarly, deriving the distance bounds for nodes 1 and 3 also requires the distance bounds for nodes 1 and 2 to be available in advance. This implies that deriving distance bounds should be performed in the order shown in Figure 5. Figure 5(a) shows the order of calculating distance bounds with a sample IDG. The distance bounds for nodes 1 and 2 are calculated first followed by the calculation of distance bounds for nodes 2 and 3, nodes 1 and 3, and so on. Figure 5(b) shows the order of calculating distance bounds using a table whose cell represents the distance bounds between the row-numbered node and the column-numbered node.

Generalizing the above example, we can compute the distance bounds between two nodes as follows. Let $D_{i,j}^{min}$ and $D_{i,j}^{max}$ be the minimum (required) and maximum (possible) distances between the issue times of nodes i and j (i $<$ j), respectively. As shown in Figure 5, the calculation order is: $[D_{1,2}^{min}, D_{1,2}^{max}]$, $[D_{2,3}^{min}, D_{2,3}^{max}]$, $[D_{1,3}^{min}, D_{1,3}^{max}]$, $[D_{3,4}^{min}, D_{3,4}^{max}]$, $[D_{2,4}^{min}, D_{2,4}^{max}]$, $[D_{1,4}^{min}, D_{1,4}^{max}]$, ... In calculating $[D_{i,j}^{min}, D_{i,j}^{max}]$, we consider the nodes that have dependences with node j in the IDG. We classify such nodes into the following three classes according to their positions in the IDG as shown in Figure 6: (1) the nodes preceding node i (i.e., nodes $p_1, p_2, \ldots, p_m$ in Figure 6(b)), (2) node i (Figure 6(c)), and (3) the nodes succeeding node i (i.e.,

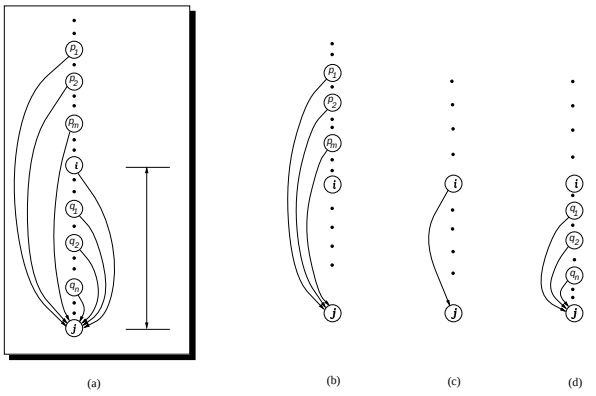


Figure 6. Three classes of nodes having dependences with node j.

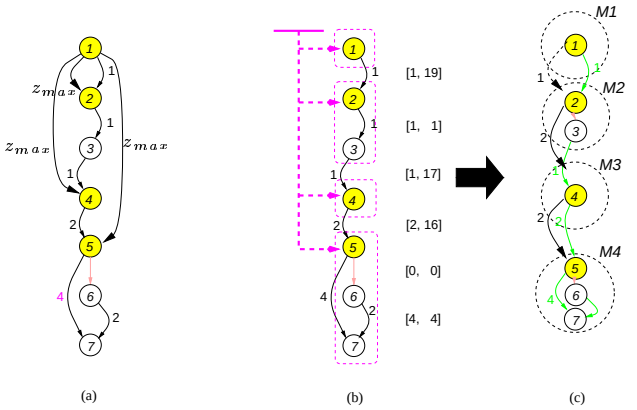


Figure 7. IDG modifications for $D^{max}_{i,j}$ calculation and merging operation.

nodes $q_1, q_2, \ldots, q_n$ in Figure 6(d)). In order to calculate $D^{min}_{i,j}$, three candidate lower bound values from the three class are computed separately, and then the largest of three candidate values is chosen to be $D^{min}_{i,j}$. For the first class, we calculate the differences between $\ell_{p_x,j}$ and $D^{max}_{p_x,i}$ where $1 \leq x \leq m$. $\ell_{p_x,j}$ is given in the original IDG and $D^{max}_{p_x,i}$ is calculated at an earlier stage. The largest of these differences is the candidate lower bound value from the first class. For the second class, $\ell_{i,j}$ is the candidate value. For the last class, we calculate the sums of D^{min}_{i,q_y} and $\ell_{q_y,j}$ where $1 \leq y \leq n$. The largest of these sums is the candidate value from the third class.

The maximum distance $D^{max}_{i,j}$ between the issue times of nodes i and j can be obtained when the issue of node j is maximally delayed relative to node i. For safe bounding on the maximum distance, we introduce additional edges (called *max edges*) between node 1 and the unresolved nodes of the IDG. The max edges between node 1 and unresolved node i has a weight of $max\_latency\_unresolved(i)$, thus effectively models the maximum possible delay between a preceding (but yet unknown) node and unresolved node i. Figure 7(a) shows the IDG of Figure 3 with the max edges added. Note that the weights are all z_{max} since the unresolved nodes have registers as their unresolved resources. Using the modified IDG with the max edges, the $D^{max}_{i,j}$ calculation proceeds similarly to the $D^{min}_{i,j}$ calculation. (In order to distinguish the modified IDG and the original IDG, we use $\ell'_{i,j}$ to indicate a weight in the modified IDG. $\ell'_{i,j}$ is defined as follows: $\ell'_{i,j} = \ell_{i,j}$ if $i \neq 1$, $\ell'_{i,j} = max\_latency\_unresolved(j)$ if node j is an unresolved node and $i = 1$, and $\ell_{i,j} = 0$ if there is no edge between nodes i and j in the IDG.) Three candidate upper bound values from the three classes are computed and the largest value is selected as $D^{max}_{i,j}$. For the first class, we calculate the differences between $\ell'_{p_x,j}$ and $D^{min}_{p_x,i}$ where $1 \leq x \leq m$. The largest of these differences is the candidate value from the first class. From the second class, we take $\ell'_{i,j}$ as in the $D^{min}_{i,j}$ calculation. For the third class, we calculate the sums of D^{max}_{i,q_y} and $\ell'_{q_y,j}$ where $1 \leq y \leq n$. The largest of the sums is the candidate from the last class. The $D^{min}_{i,j}$ and $D^{max}_{i,j}$ calculations can be summarized by the following equations:

$$D^{min}_{i,j} = \max(\max_{1\leq x\leq m}(\ell_{p_x,j} - D^{max}_{p_x,i}), \ell_{i,j}, \max_{1\leq y\leq n}(D^{min}_{i,q_y} + \ell_{q_y,j})) \quad (1)$$

$$D^{max}_{i,j} = \max(\max_{1\leq x\leq m}(\ell'_{p_x,j} - D^{min}_{p_x,i}), \ell'_{i,j}, \max_{1\leq y\leq n}(D^{max}_{i,q_y} + \ell'_{q_y,j})). \quad (2)$$

Each equation in the above has three elements in the outermost max operation and each element corresponds to each class of nodes.

4.3. Multiple Issuing and Merging Steps

In the multiple issuing step, we identify the instructions that are guaranteed to be issued simultaneously, based on the distance bounds computed in the previous section, and then refine the distance bounds of the IDG to reflect the effect of identified instructions to the IDG representation. In principle, two nodes whose distance bounds are [0, 0] can be issued simultaneously. If a series of nodes have the distance bounds of [0, 0], these nodes can be potentially issued at the same time. However, the number of simultaneously issued instructions cannot be greater than the multiple-issue limit of a target processor. For example, in our machine model where the maximum of k instructions can be issued simultaneously, consider a series of nodes $p, p+1, \ldots, p+q$ such that $[D^{min}_{p+i,p+i+1}, D^{max}_{p+i,p+i+1}]$ = [0, 0] for $0 \leq i < q$. For the nodes, $p, p+1, \ldots, p+q$, we first check if $q+1 \leq k$. If $q + 1 \leq k$, all the nodes in the series are guaranteed to be issued at the same time and no special refinement step is necessary to reflect the effect of the multiple-issue identification. On the other hand, if $q + 1 > k$, not all the nodes in the series can be issued simultaneously. Since our

machine model assumes an in-order issue, only the first k nodes, $p, p+1, \ldots, p+k-1$ can be issued at the same time. The remaining nodes, $p+k, \ldots, p+q$, should be issued at least one cycle later than the first k nodes. Therefore, in case that $q+1 > k$, after identifying the first k nodes, the IDG should be modified. To represent that only the first k nodes can be issued simultaneously, we replace an edge between nodes $p+k-1, p+k$ with an edge of a weight 1. Since the new edge will change the distance bounds of the IDG, we recalculate the distance bounds for the affected nodes. A more detailed description on the multiple issuing algorithm can be found in [12].

As it will be described in detail in the next section, an IDG is encoded into the PA structure that represents a program construct, and two successive IDGs are concatenated into a new IDG during the hierarchical refinement process. Repeated concatenations, however, may require a large amount of space to maintain all the nodes of the concatenated IDGs. Therefore, we propose a merging operation on an IDG that reduces the number of nodes in an IDG. (In our proposed technique, a merging operation is performed after the multiple issuing step.) Two adjacent nodes in an IDG are merged into a single node if the lower bound of the distance between two nodes are same as the upper bound of the distance between two nodes. That is, if $D^{min}_{j,j+1}$ is equal to $D^{max}_{j,j+1}$, nodes j and $j+1$ can be merged into a single node. In general, N nodes, $p, p+1, \ldots, p+N-1$, can be merged into a single node (after the multiple issuing step) if $[D^{min}_{p,p+1}, D^{max}_{p,p+1}] = [c_{i_1}, c_{i'_1}]$, $[D^{min}_{p+1,p+2}, D^{max}_{p+1,p+2}] = [c_{i_2}, c_{i'_2}]$, $\ldots$, $[D^{min}_{p+N-2,p+N-1}, D^{max}_{p+N-2,p+N-1}] = [c_{i_{p+N-1}}, c_{i'_{p+N-1}}]$ and $c_{i_k} = c_{i'_k}$ for $1 \leq k \leq p+N-1$. Figure 7(b)-(c) illustrates the merging operation. The IDG shown in Figure 7(b) includes the distance bounds between nodes as well as the weights between nodes. In Figure 7(b), the nodes that can be merged are shown inside a dotted box. For example, nodes 2 and 3 can be merged because the distance bounds between them are [1, 1]. Nodes 5, 6 and 7 can be merged as well because they have the distance bounds [0, 0] and [4, 4]. Since merged nodes (except for the first node) have a constant distance on their issue times relative to the first node, no information is lost in predicting the WCETs by the merging operation. On the other hand, as the hierarchical analysis proceeds, the number of nodes that can be merged grows rapidly since most of unresolved nodes will be resolved during the hierarchical refinement process. Therefore, the reduction in the number of nodes by the merging operations is substantial.

There are two more factors to consider during the merging operation. First, since the merged node represents several nodes before the merging operations are performed, we have to modify the weights between the merged node and other nodes in the merged IDG. (Note that our definition of a weight between two nodes means the minimum distance between the issues of first instructions of the two nodes.) For example, in Figure 7(c), the weight between merged nodes M_2 and M_3 is computed to be the sum of a constant distance between nodes 2 and 3 and the weight between nodes 3 and 4 in the original IDG. Second, even when a series of nodes have their distance bounds as [0, 0], these nodes are not always merged for accurate prediction of the WCETs. This happens for the following two cases: (1) a series of nodes have distance bounds of [0, 0], but the consecutive nodes start from the first node in the IDG and (2) a series of nodes have distance bounds of [0, 0] but the last node in the IDG is also a part of these nodes. In the first case, if we merged the nodes, we will lose some accuracy when we concatenate with a preceding IDG whose last n nodes have distance bounds [0, 0]. Similarly, in the second case, if we merged the nodes, we will lose some accuracy when we concatenate with a succeeding IDG whose first n nodes have distance bounds [0, 0].

	1	2	3	4	5	6	7
1		*[1,19]*	*[2,20]*	*[3,21]*	*[5,23]*	*[7,21]*	*[9,25]*
2			*[1,1]*	*[2,18]*	*[4,20]*	*[6,20]*	*[8,24]*
3				*[1,17]*	*[3,19]*	*[5,19]*	*[7,23]*
4					*[2,16]*	*[2,16]*	*[6,20]*
5						*[0,0]*	*[4,4]*
6							*[4,4]*

(a)

	M1	M2	M3	M4
M1		*[1,19]*	*[3,21]*	*[5,23]*
M2			*[2,18]*	*[4,20]*
M3				*[2,16]*

(b)

Figure 8. A tabular form representation of distance bounds.

The distance bounds for all combinations of nodes in an IDG can be represented as a tabular form. Figure 8 shows the tabular form representations of the distance bounds for our example. Figure 8(a) shows the distance bounds for original IDG and Figure 8(b) shows the distance bounds after merging is applied to the IDG. This tabular representation is included into the PA structure for a program construct.

5. Extensions to ETS

In this section, we describe the extensions made to the original ETS framework to support the WCET prediction for our multiple-issue machine model. We explain how to encode the IDG for a program construct in the PA structure and define the concatenation ($\oplus$) operation and the pruning operation using the three distance bounds steps described in the previous section.

The PA Structure To take into account the multiple-issue capability in the ETS framework, we define the PA structure of the program construct to include the IDG. To understand why an IDG is used instead of a reservation table which was used for simple pipelined processors, consider Figure 9 that

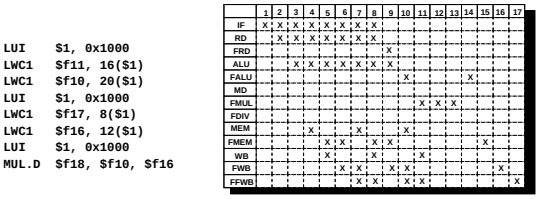


Figure 9. A reservation table example.

shows a reservation table for a sequence of instructions. In the reservation table, the rows represent the pipeline stages and the columns represent time, and an x in the reservation table specifies the use of the corresponding pipeline stage by an instruction at the indicated cycle. When a reservation table is constructed the issue time of an instruction relative to others in the same basic block is fixed. This static assignment of an issue time to an instruction leads to a severe overestimation of the WCET for multiple-issue processors because when information on the surrounding blocks are known later, the issue times cannot be easily adjusted reflecting the simultaneously issued instructions from the different blocks.

The PA structure for multiple-issue machines is shown in Figure 10. The PA structure for a program construct consists of five tables. The distance bounds derived from the IDG for the program construct are included in a table called `bounds_table` where each cell corresponds to the distance bounds between the row-numbered node and the column-numbered node. In order to compute the WCET of an IDG and to prune redundant PAs, for each node i in the IDG, the $min\_latency\_node(i)$ and $max\_latency\_node(i)$ are maintained in the PA. Two additional tables, `in` and `out`, are used to detect the transition of unresolved nodes to resolved nodes when two PAs are concatenated. The `in` table contains the information on the unresolved dependences of the IDG which may be refined as the resolved dependences by a preceding IDG at a later stage. For a register, the first node that uses or defines the register is recorded in the `in` table, while, for a functional unit, the first node that occupies the functional unit in the IDG is recorded in the `in` table. On the other hand, the `out` table contains the information on the factors that may cause the unresolved dependences of succeeding IDGs to be refined as the resolved dependences. In the `out` table, for a register, the last node that defines the register is recorded, and, for a functional unit, the last node that occupies the functional unit is recorded with its latency. When two PAs are concatenated, these five tables from each PA are used to construct new tables for the concatenated PA.

From the PA information, we can derive the WCET of the IDG. Let the WCET of an IDG be the worst case execution time from the issue of the first node to the end of the executions of all the nodes. Then we can calculate the WCET of the IDG as follows. Assume $\mathcal{N}$ to be the set of nodes of the

```
struct PA of a path p {
    struct BOUNDS {
            d_min, d_max ;
    } bounds_table[N_nodes][N_nodes] ;
    min_latency_node[N_nodes] ;
    max_latency_node[N_nodes] ;
    struct IN
            in[N_resources] ;
    struct OUT {
            node ;
            latency ;
    } out[N_resources] ;
};
```

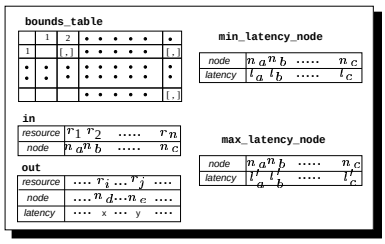


Figure 10. The structure of PA.

IDG and $max\_latency\_node(i)$ to be the maximum of the latencies of the functional units in $occupy(i)$ as defined in Section 4. Then the WCET of an IDG can be computed as follows:

$$T_{worst} = \max_{i \in \mathcal{N}}(D^{max}_{1,i} + max\_latency\_node(i)).$$

This formula produces the time required from the issue of the first instruction of an IDG to the completion of all the instructions in the IDG in the worst case. Once the PA for a whole program is built, we can compute the WCET of the program from the PA in a similar way. However, in the calculation of the WCET for the whole program, we can assume that the instructions in unresolved nodes in the final IDG are issued with the minimal delay because there will be no instructions that can resolve these unresolved dependences. Therefore, the WCET for a program is computed as follows:

$$WCET_{program} = \max_{i \in \mathcal{N}}(D^{min}_{1,i} + max\_latency\_node(i)).$$

The Concatenation and Pruning Operations The second step of augmenting the extended timing schema for multiple-issue machines is to define the semantic of the concatenation ($\oplus$) operation according to the structure of the PA. The concatenation operation on two PAs produces a new PA containing the information on a new IDG constructed by linking two IDGs from the original PAs. In each concatenation, the three bounds steps (the bounding step, the multiple issuing step, and the merging step) are applied to the newly constructed PA. Since the IDG is represented by

the five tables, `bounds_table`, `min_latency_node`, `max_latency_node`, `in`, and `out`, the five tables are newly created from the tables in the original PAs.

The concatenation operation on two PAs w_1 and w_2 producing a new PA w_3 consists of the following four steps. Assume that w_2 follows w_1 in a program order. Let N_{w_1} and N_{w_2} denote the number of nodes of the IDGs in w_1 and w_2, respectively. Then, the dimensions of bounds table $\texttt{bounds\_table}_{w_1}$ for w_1 is $N_{w_1} \times N_{w_1}$, while the dimensions of the bounds table $\texttt{bounds\_table}_{w_2}$ for w_2 is $N_{w_2} \times N_{w_2}$.

The first step of the concatenation operation is to build a new bounds table $\texttt{bounds\_table}_{w_3}$ from the original two bounds tables $\texttt{bounds\_table}_{w_1}$ and $\texttt{bounds\_table}_{w_2}$. The size of the new `bounds_table` will be initially $(N_{w_1} + N_{w_2}) \times (N_{w_1} + N_{w_2})$. This initial table models the new IDG where two original IDGs from w_1 and w_2 are linked together. Once $\texttt{bounds\_table}_{w_3}$ is initially filled, we apply the bounding algorithm (to compute the distance bounds), the multiple issuing algorithm (to identify the instructions that can be issued simultaneously) and the merging algorithm (to shrink the table size) to $\texttt{bounds\_table}_{w_3}$. The new bounds table $\texttt{bounds\_table}_{w_3}$ is initially filled as follows. Since the distance bounds of instructions within w_1 and the distance bounds of instructions within w_2 are not modified by the concatenation operation, $\texttt{bounds\_table}_{w_3}[i][j] = \texttt{bounds\_table}_{w_1}[i][j]$ for $1 \leq i, j \leq N_{w_1}$ and $\texttt{bounds\_table}_{w_3}[N_{w_1} + i][N_{w_1} + j] = \texttt{bounds\_table}_{w_2}[i][j]$ for $1 \leq i, j \leq N_{w_2}$.

The second step of the concatenation is to check if new dependences exist between a node n_1 of w_1 and a node n_2 of w_2 after two PAs were concatenated. Since the new dependences exist when n_1 and n_2 share the common resources, we compare the `out` table of w_1 and the `in` table of w_2. If a resource r_i is found to be shared by `out` of w_1 and `in` of w_2, a new edge with the weight of `out[i].latency` from w_1 is attached between the corresponding nodes (i.e., node p from w_1 and node q from w_2 if `out[i].node` of w_1 is p and `in[i].node` of w_2 is q). This new edge with the corresponding weight is used in filling $\texttt{bounds\_table}_{w_3}[i][j + N_{w_1}]$ for $1 \leq i \leq N_{w_1}$ and $1 \leq j \leq N_{w_2}$. Once $\texttt{bounds\_table}_{w_3}$ is built, we apply the three bounding steps to find the simplified version of $\texttt{bounds\_table}_{w_3}$.

The next step is to fill the `min_latency_node` and the `max_latency_node` for each node. These latencies are calculated in the merging step. If nodes j and k are found to be merged into node m, the `min_latency_node`$[m]$ and the `max_latency_node`$[m]$ of the merged node m are obtained by adding the constant distance between the nodes j and k to the `min_latency_node`$[k]$ and the `max_latency_node`$[k]$ of node k. In this way, the latencies are modified to represent the times from the issue of the first instruction as defined in Section 4.1.

Finally, the contents of `in` and `out` for w_3 are filled using `in`s and `out`s of w_1 and w_2. The `in` of w_3 is filled by copying the contents of `in`s of w_1 and w_2. The contents of `in` of w_1 will override the contents of `in` of w_2. The `out` of w_3 is filled in a similar manner.

The last missing component for applying the ETS is to define the pruning operation. The purpose of a pruning operation is to eliminate the paths that cannot be the worst case execution path. In other words, in the same WCTA, if the WCET of a PA w_1 in the worst case scenario is shorter than the WCET of any other PA w_2 in the best case scenario, a PA w_1 can be safely pruned from the WCTA. The WCET in the best case scenario of a PA w, which we call $T_{best\_worst}(w)$, is defined as follows:

$$T_{best\_worst}(w) = D^{min}_{1,m} + min\_latency\_node(i),$$

where m is the number of nodes in the IDG of a PA w. In this equation, $T_{best\_worst}(w)$ is the minimum delay from the issue of the first instruction to the issue of the last instruction in the IDG. On the other hand, the WCET in the worst case scenario of a PA w, which we call $T_{worst\_worst}(w)$, is defined as follows:

$$T_{worst\_worst}(w) = D^{min}_{1,m} + 2 \times z_{max},$$

where z_{max} is the maximum latency of an instruction in the target processor (as defined in Section 4.1). The worst scenario occurs when the issue of the first instruction in w is maximally delayed by the dependences with the preceding program constructs and the instructions in w maximally delay the issue of the first instruction in the succeeding program constructs. In order to account for two maximal delays (before and after w), we add $2 \times z_{max}$ to $D^{min}_{i,m}$. The pruning condition can be more formally specified as follows:

> A PA w in a WCTA W can be pruned without affecting the prediction for the worst case timing behavior of W if $\exists w' \in W$ such that $T_{worst\_worst}(w) < T_{best\_worst}(w')$.

6. Experimental Results

We have performed experiments to validate our approach by building a timing tool based on the proposed technique and comparing the WCET bounds produced by the timing tool to the simulation results measured from a simulator. Figure 11 shows an overview of our timing analysis environment. The timing analyzer takes as input the assembly code, program syntax information, and the call graph along with the user-provided information (e.g., loop bound) to predict the WCET of the program. A modified lcc compiler

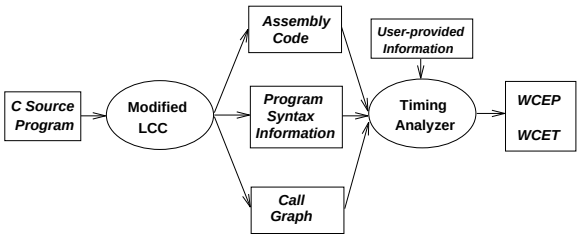


Figure 11. Experiment setup.

benchmarks	description
Arrsum	calculates the sum of 10 array elements.
Fib	computes the 30th element of the Fibonacci sequence.
MM	multiplies two 5×5 matrices.
BS	performs binary search over 15 integer array elements.
ISort	sorts 10 integer array elements using the insertion sort algorithm.
InLP	shows more ILP (relative to other benchmarks) by the manual rearrangement of the assembly instructions.

Table 2. The benchmarks used in our experiments.

[11] accepts a C source program and generates the inputs to the timing analyzer.

The predicted WCETs were compared with the measurements obtained from a simulator that models our target multiple-issue machine model. The simulator was built using a software tool, called Visualization-based Microarchitecture Workbench (VMW) [2], that provides a framework for systematically constructing a processor simulator at the microarchitecture level. In VMW, a new processor is specified using several machine specification files and the corresponding processor simulator is automatically generated. (The VMW tool has been successfully used to simulate multiple-issue processors such as superscalar processors (e.g., PowerPC620).) The machine specification for the target multiple-issue machine included a microarchitecture machine organization specification file (71 lines), an instruction syntax specification file (115 lines), an instruction semantic specification file (127 lines), and an instruction timing specification file (543 lines). The simulator assumes the MIPS R3000 instruction-set architecture and displays the machine status during simulations along with the execution times.

Table 2 summarizes the benchmarks used in our experiments. Since the modified lcc compiler was originally developed for simple pipelined machines without employing the special compiler optimizations techniques that can increase instruction-level parallelism (ILP) of the compiled code, the generated assembly code contains relatively small ILP. In order to validate the applicability of our technique on

	single issue		double issue		quadruple issue	
	S	P	S	P	S	P
Arrsum	108	108	92	92	92	92
Fib	227	227	190	190	189	189
MM	4142	4142	3553	3553	3552	3552
BS	101	106	81	84	80	83
ISort	1262	2126	1088	1844	1087	1844
InLP	3331	3331	2498	2498	2290	2290

S: simulation, P: prediction

Table 3. The experimental results.

programs with more ILP, we made a benchmark called *InLP* whose instructions were manually reordered for more ILP after the assembly code was generated from the modified lcc. The analysis results and the simulation results for the benchmarks are compared in Table 3. The results are shown for three different issue numbers: single issue, double issue, and quadruple issue. Because of the small ILP available in the assembly code, as shown in the table, the execution times of the benchmarks (except for *InLP*) are very close each other on a double-issue machine and a quadruple-issue machine.

For the *Arrsum*, *Fib*, *MM*, and *InLP* benchmarks, the analysis results are exactly same as the simulation results because the execution path of each benchmark program is unique. However, for *BS* and *ISort*, the analysis results are larger than the simulation results. The differences between the analysis results and the simulation results are mainly from infeasible paths. As we discussed in [11], we believe that the infeasible path problem exists in any static WCET prediction technique, and the elimination of these paths using dynamic path analysis is an issue orthogonal to the WCET prediction approach. The existing path analysis method (e.g., the work done by Park [13]) can easily be integrated with the proposed method, thus producing tighter WCETs for the *BS* and *ISort* benchmarks.

7. Conclusion and Future Work

In this paper, we described a timing analysis technique that can accurately predict the WCETs of tasks for multiple-issue machines. Our technique is based on the existing extended timing schema (ETS). We enhanced the ETS framework to account for the timing variation resulting from multiple issues of instructions per cycle. The main extension was on the PA structure. Instead of reservation tables used for the original ETS, we maintain an IDG (Instruction Dependence Graph) to represent dependences among instructions and include it in the PA structure. From the IDG, the minimum and maximum distance bounds between the issue times of the instructions are computed. These bounds are used to identify the instructions that can be issued simultaneously. The identification of the constant distance

instructions as well as the simultaneously issued instructions allows the IDG to be simplified, reducing the number of necessary nodes to be kept in the IDG. The concatenation operation is redefined to support the combining of two IDGs followed by the IDG simplifications. We also redefined the pruning condition that can eliminate an execution path that cannot be part of the worst case execution path, considering the effect of multiple issues on the execution time.

We also built a timing tool based on the proposed technique and compared the WCET bounds of several benchmark programs predicted by the timing tool to their measurements from a simulator. The results show that the proposed technique can predict the WCETs for in-order, multiple-issue machines in a similar accuracy to the results from simple pipelined processors.

Our work described in this paper strongly suggests that the multiple-issue capability of modern microprocessors can be accurately analyzed to predict the WCETs of programs. However, to estimate the WCETs of tasks for more realistic multiple-issue processors such as commercial superscalar processors, our technique needs to be extended further to handle other advanced architectural features that can cause the timing variation in multiple-issue processors. For example, many superscalar processors execute in an out-of-order, multiple-issue fashion and support the dynamic scheduling, dynamic branch prediction and speculative execution [6]. Since these features all affect the execution time, the proposed technique should be extended to account for these features. Therefore, our current research direction is focused on developing the techniques for modeling these dynamic architectural features and predicting the WCETs taking into account the timing effect of these features.

References

[1] 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.

[2] T. A. Diep and J. P. Shen. VMW: A Visualization-Based Microarchitecture Workbench. *IEEE Computer*, 28(12):57–64, 1995.

[3] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] 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, December 1995.

[5] J. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach 2nd Ed.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[7] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

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

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

[10] Y. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th Real-Time Systems Symposium*, pages 254–263, 1996.

[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 for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[12] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Processors. Technical Report SNU-CE-AN-98-001, Architecture and Network Laboratory, Seoul National University, 1998.

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

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

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