An Iterative Integer Linear Programming Method for Bounding Program Performance on Embedded Systems

Tai-Yi Huang and Kuang-Li Huang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, ROC
TEL: +886-3-574-2965
FAX: +886-3-572-3694
tyhuang@cs.nthu.edu.tw

Yeh-Ching Chung
Department of Information Engineering
Feng Chia University
Taichung, Taiwan 407, ROC
TEL: +886-4-2451-7250x3765
FAX: +886-4-2451-6101
ychung@fcu.edu.tw

July 3, 2002

Abstract

The integer linear programming method has been used to bound the performance of a program in a hard-real-time embedded system. The maximum value of the cost function of a program under a set of linear constraints on the execution count of each instruction is an upper bound of the worst-case execution time of the program. In this paper we extend this method to bound the execution time of a program executed on a dynamic architecture where the execution time of each instruction depends on not only itself but also its adjacent instructions. Our method follows the control flow of the program iteratively to determine the set of all possible execution times of each instruction and construct a set of linear constraints on their execution counts. We demonstrate the capability of this method on an architecture where a processor has an instruction cache and an instruction pipeline, and cycle-stealing DMA I/O is concurrently executing. We conducted extensive simulations on a widely-used embedded microprocessor. The experimental results show that our method safely and tightly bounds the worst-case execution time of a program executed on such an architecture.

Keywords: integer linear programming, hard-real-time systems, embedded system, worst-case execution time, cycle-stealing DMA I/O

Submitted Workshop: Workshop on Computer Systems, ICS2002

1 Introduction

A hard-real-time embedded system is required to process tasks with timing constraints that must be met to prevent the failure of the whole system. The schedulability analysis that determines whether a particular system can meet its timing constraints relies on prior information on the worst-case execution time (WCET) of each task. A number of methods have been developed by different research groups to predict the WCET of a program [1, 3–6, 8–13]. Examples include, but not limited to, the static cache simulation developed by Muller *et al.* [11], the timing schema approach developed by Park *et al.* [12], and the distance-bound method developed by Lim *et al.* [10].

Li *et al.* [8] first converted the problem of bounding the WCET of a program into one of solving a set of integer linear programming problems. This method defines the execution time of a program as the sum of the products of the execution count of each instruction and its corresponding execution time. The maximum value of the cost function under a set of linear constraints on the execution counts is an upper bound of the WCET of the program. This method was extended by Li *et al.* [9] to consider the effect of instruction caching, and incorporated with abstract interpretation by Theiling *et at.* [13] to predict the cache behavior. However, all previous work assume that the execution time of each instruction can be analyzed independently, without considering its adjacent instructions.

This paper extends the integer linear programming method to bound the WCET of a program executed on a dynamic architecture where the execution time of an instruction depends on not only itself but also its adjacent instructions. Without loss of generality, we illustrate the idea with an architecture where a processor has an instruction cache and an instruction pipeline, and cycle-stealing DMA I/O is concurrently executing. A DMA controller (DMAC) operates either in the burst mode or in the cycle-stealing mode. A DMAC that operates in the cycle-stealing mode transfers data by stealing bus cycles from the executing program. The cycle-stealing operation retards the progress of the executing program and extends its execution time. The execution time of an instruction executing concurrently with cycle-stealing DMA I/O is the sum of the execution time of the instruction when it executes alone and the delay due to cycle-stealing DMA I/O. Because a DMA transfer may cross two instructions on such an architecture, the delay suffered by an instruction varies, depending upon the execution of the instruction itself and its adjacent instructions.

Our method first follows the control flow of the program iteratively to determine the set of all possible execution times of each instruction and their relationship with its adjacent instructions. We next model the execution behavior of each instruction and its adjacent instructions by a directed graph. Each execution time and each edge in the directed graph is assigned an execution count. These execution counts must satisfy a set of linear constraints. These possible execution times, execution counts, and linear constraints are used as inputs to an integer linear programming problem, the solution of which is an upper bound of the WCET of the program being analyzed.

To demonstrate the effectness of our method on bounding the WCET, we conducted an experiment on a widely-used embedded microprocessor. We compare our WCET predictions with the traditional pessimistic WCET predictions for several sample programs. The experimental results show that our predictions safely bound the WCETs of these programs. In addition, our predictions are as much as 47% tighter than the pessimistic predictions.

The rest of the paper is structured as follows. Section 2 describes related work. Section 3 describes the machine model and Section 4 describes the interference of cycle-stealing DMA I/O on program execution time. Section 5 describes the iterative process that determines the set of all possible execution times of each instruction. We present our experimental results in Section 6. Finally, Section 7 gives some concluding remarks.

2 Related Work

The integer linear programming method was first used by Li *et al.* [8] to compute the WCET of a program executed alone without interrupts on a simple architecture where the execution time of each instruction is fixed. This method first decomposes a program into a number of basic blocks, each of which is a straight-line sequence of instructions. The execution time c_i of a basic block B_i is equal to the sum of the execution times of all instructions in the block. The execution time of the program can be computed by summing the products of the execution counts of the basic blocks in the program and their corresponding execution times. Let x_i be the execution count of the basic block B_i, and N be the number of basic blocks in a program. The WCET of the program is bounded by the maximum value of the cost function

$$\sum_{i=1}^{N} c_i x_i \tag{1}$$

under a set of linear constraints on the x_i's. The execution behaviors of the basic blocks are related together through the set of linear constraints on their execution counts. In addition, the user can provide path information in the form of linear constraints to eliminate infeasible paths, which can never be executed, to tighten the WCET prediction.

The assumption that the execution time of each instruction is fixed does not hold on an instruction-cache architecture. Li *et al.* [9] extended the integer linear programming method described above to consider the effects of both direct-mapped and set-associative instruction-cache architectures. This method first partitions each basic block into one or more l-blocks. An l-block is a sequence of contiguous instructions within the same basic block that are mapped to the same cache line. Consequently, all instructions except the first one in each l-block will always reside in the instruction cache when accessed. An l-block has two possible execution times on an instruction-cache architecture, one when the first instruction of the l-block causes a cache hit and one when it causes a cache miss. Let $c_{i,j}^h$ and $c_{i,j}^m$ denote the cache-hit and cache-miss execution times of the l-block $B_{i,j}$, and let $x_{i,j}^h$ and $x_{i,j}^m$ denote its cache-hit count and cache-miss count, respectively. The execution count of $B_{i,j}$ is equal to the execution count x_i of B_i, i.e.,

$$x_i = x_{i,j}^h + x_{i,j}^m,\ j = 1, \ldots, n_i.$$

Furthermore, the cost function of the execution time of the program is replaced by

$$\sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i,j}^h x_{i,j}^h + c_{i,j}^m x_{i,j}^m). \tag{2}$$

The cache behavior, direct-mapped or set-associative, can be specified in the form of linear constraints on the $x_{i,j}^h$'s and $x_{i,j}^m$'s. The maximum value of the cost function (2) under the set of linear constraints on the execution counts is an upper bound of the WCET of the program being analyzed.

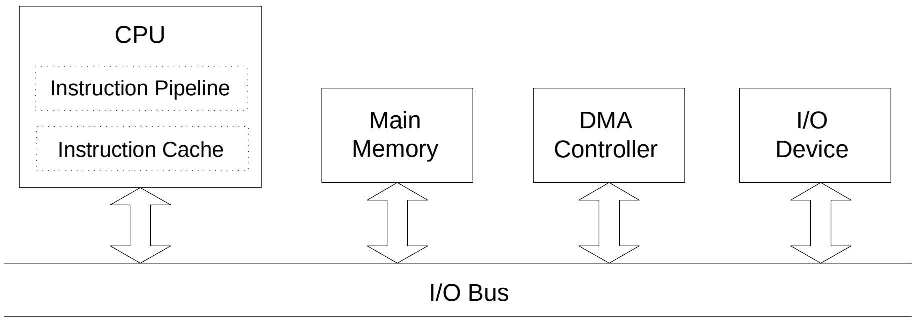


Figure 1: The machine model

3 The Machine Model

Our work builds on top of the method described in Section 2 and the cost function (2) to predict the WCET of a program executed on an architecture where the execution time of an l-block depends on not only itself but also its adjacent l-block. Without loss of generality, we base our method on a commonly-used machine model shown in Figure 1. The CPU has an on-chip instruction cache and an instruction pipeline. The DMAC and the CPU share a single I/O bus. We focus on the case in which the DMAC operates in the cycle-stealing mode. The bus controller allows only one bus master at any time. Consequently, either the CPU or the DMAC, but not both, can hold the bus and transfer data at any time instant.

On a pipelined processor, multiple instructions are overlapped in execution. We represent the execution of an l-block when it executes alone on this machine model by two reservation tables, one when it causes a cache hit and one when it causes a cache miss. We call them the cache-hit reservation table and the cache-miss reservation table, respectively. A *reservation table* describes the activities within a pipeline [7]. In a reservation table, the rows represent the stages in the pipeline and the columns represent the pipeline status in each processor cycle.

An example is given in Figure 2 to show an l-block and its cache-hit and cache-miss reservation tables. The instruction pipeline is composed for 4 stages. An instruction is fetched during the Instruction Fetch (**IF**) stage, and decoded during the Instruction Decode (**ID**) stage. The instruction executes during the Execution (**EX**) stage, and data produced by the instruction is written to the memory during the Write Back (**WB**) stage. Because we are concerned primarily with whether there is any bus-access activity during a pipeline stage, we classify all pipeline stages into two cat-

```
MOVE.L    D1,D0
MOVE.L    D2,-(A7)
NOP
ADD.L     D0,D4
LSL.L     #2,D4
ADD.L     D4,D6
```

(a) an 1-block

	1	2	3	4	5	6	7	8	9	10
IF	E	E	E	E	E		E			
ID		E	E	E	E		E	E		
EX			E	E			E	E	E	E
WB					B	B				

(b) its cache-hit reservation table

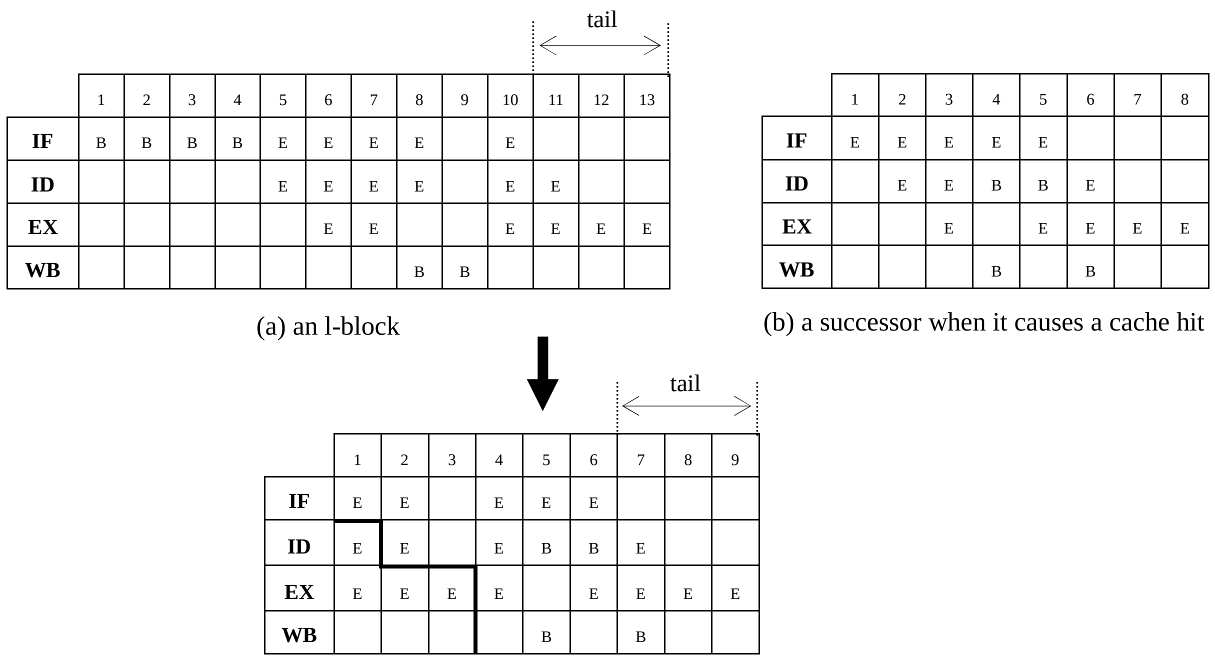
	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	B	B	B	B	E	E	E	E		E			
ID					E	E	E	E		E	E		
EX						E	E			E	E	E	E
WB								B	B				

(c) its cache-miss reservation table

Figure 2: An l-block and its cache-hit and cache-miss reservation tables

egories: **B** (bus-access) stages and **E** (execution) stages. B-stages are those pipeline stages during which there is bus-access activity. In contrast, during E-stages, there is no bus-access activity. We mark a processor cycle in the reservation table by a B if the corresponding stage at the indicated processor cycle is a B-stage, and by an E if it is an E-stage. The cache-miss reservation table shown in Figure 2c begins with several B-stages to fetch the instruction and the subsequent instructions in the l-block from the main memory. Each of the subsequent instructions begins with an E-stage to fetch the instruction from the on-chip instruction cache. Because none of the instructions in the l-block fetches any operand, all ID stages are E-stages. Finally, all EX stages are E-stages and all WB stages are B-stages.

The pipelined execution of an l-block affects the pipelined execution of a *successor* (i.e., an l-block executed immediately after it). We define the *tail* of a reservation table as its last few columns starting from the column at which the CPU is ready to fetch the first instruction of a successor to the last column of the table. We concatenate the tail of a reservation table of an l-block and a successor's cache-hit (cache-miss) reservation table to obtain another reservation table which describes the pipelined execution of the successor when it causes a cache hit (cache miss).



	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	B	B	B	B	E	E	E	E		E			
ID					E	E	E	E		E	E		
EX						E	E			E	E	E	E
WB								B	B				

(a) an l-block

	1	2	3	4	5	6	7	8
IF	E	E	E	E	E			
ID		E	E	B	B	E		
EX			E		E	E	E	E
WB				B		B		

(b) a successor when it causes a cache hit

	1	2	3	4	5	6	7	8	9
IF	E	E		E	E	E			
ID	E	E		E	B	B	E		
EX	E	E	E	E		E	E	E	E
WB					B		B		

(c) the pipelined execution of the successor

Figure 3: The execution of an l-block and its successor

Figure 3a shows the cache-miss reservation table of the l-block in Figure 2, and Figure 3b shows the cache-hit reservation table of a successor. The tail of the l-block's reservation table consists of columns 11 to 13. We concatenate the tail of the l-block's reservation table and the successor's cache-hit reservation table to obtain the reservation table shown in Figure 3c. In turn, the tail of the new reservation table consists of columns 7, 8, and 9. The execution time of an l-block is the interval from the time when the CPU is ready to fetch the first instruction of the l-block to the time when the CPU is ready to fetch the first instruction of a successor, if the l-block has any successor, or the time when the CPU finishes the execution of the l-block, if the l-block has no successor (i.e., the last l-block). Let $\mathrm{T_c}$ be the period of a processor clock cycle. Accordingly, the execution time of the l-block in Figure 3 is $10 * \mathrm{T_c}$ and the execution time of the successor is $6 * \mathrm{T_c}$, if it has any successor, or $9 * \mathrm{T_c}$ otherwise.

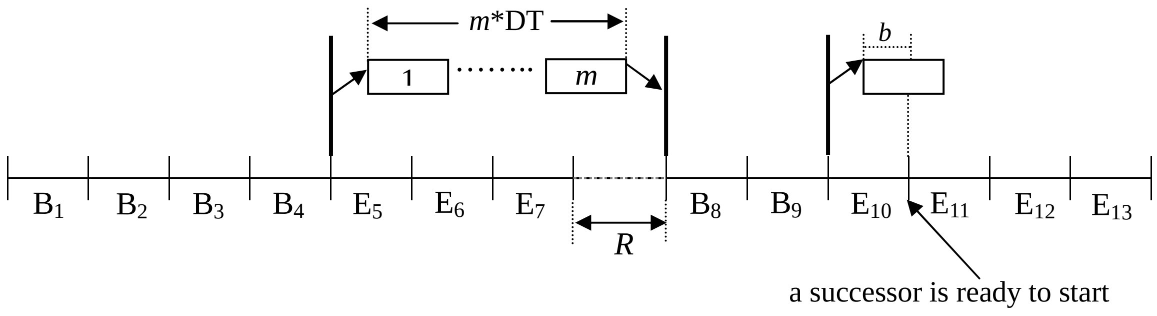


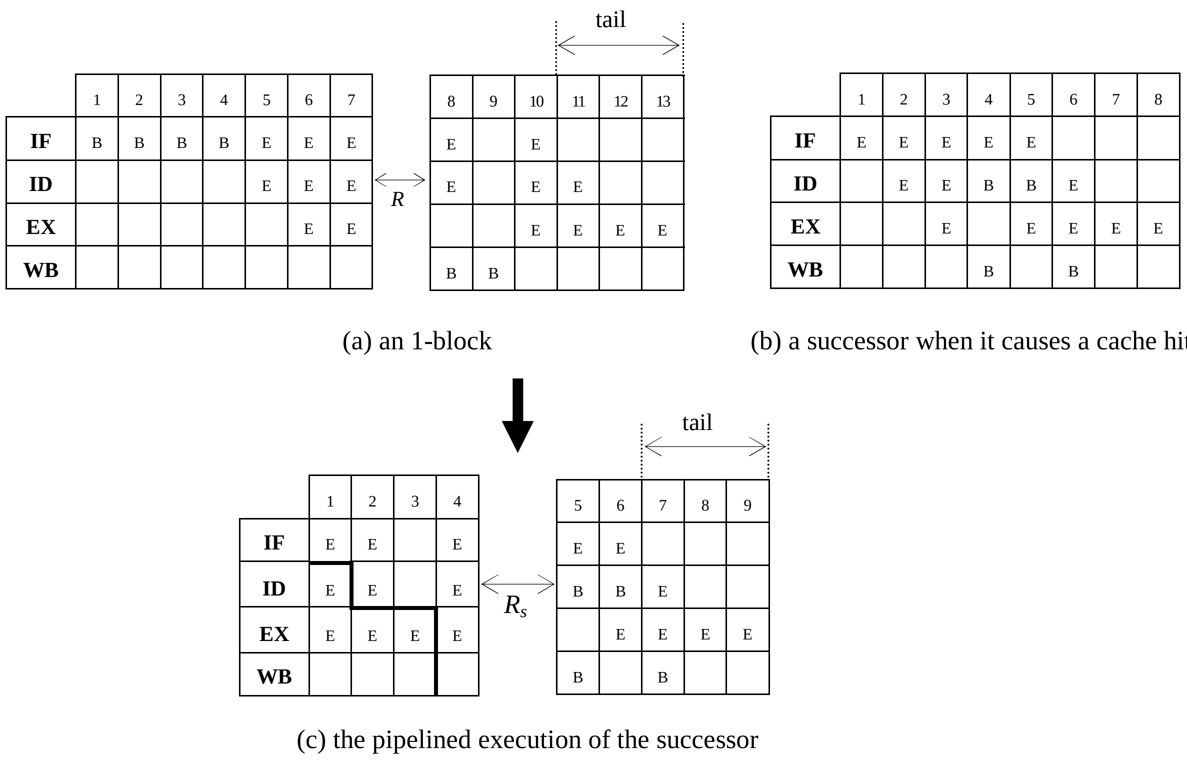
Figure 4: The concurrent execution of the DMAC and a sequence of processor cycles

4 Cycle-Stealing DMA I/O

We call a processor cycle a B-cycle if at the processor cycle any stage in the instruction pipeline is a B-stage. Otherwise, we call a processor cycle an E-cycle. The CPU uses the system bus only during B-cycles. To analyze the bus contention between the CPU and the DMAC during the cache-miss execution of the l-block shown in Figure 2c, we represent the bus-access pattern of the l-block by a sequence of B-cycles and E-cycles,

$$B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow E_5 \rightarrow E_6 \rightarrow E_7 \rightarrow B_8 \rightarrow B_9 \rightarrow E_{10} \rightarrow E_{11} \rightarrow E_{12} \rightarrow E_{13}.$$

Figure 4 illustrates the concurrent execution of the DMAC and the sequence of processor cycles. Here we assume that bus contention between the CPU and the DMAC is regulated according to the VMEbus [14] bus access protocol. This protocol is sufficiently general that our analysis may be easily applied to many other commonly-used bus protocols. To access the bus, the DMAC asserts the bus request line. Since the bus is used by the CPU during B_1 to B_4 cycles, the DMAC waits. The bus is free after the CPU enters E_5 cycle from B_4 cycle. The DMAC gains the control of the bus after a short delay, called the bus master transfer time (BMT). The DMAC keeps transferring data as long as the CPU continues to be in E-cycles. The CPU sends a bus request when it is ready to enter B_8 cycle from E_7 cycle. Because the DMAC is currently transferring data, the CPU waits and the pipelined execution stalls. The DMAC checks whether there is any pending request at the end of each data transfer. If there is a bus request, the DMAC releases the bus. After another BMT delay, the CPU gains the control of the bus and the pipelined execution resumes.



(a) an 1-block

(b) a successor when it causes a cache hit

(c) the pipelined execution of the successor

Figure 5: The interference of DMA I/O on the pipelined execution of a successor

Let m be the number of units of data the DMAC transfers during the interval between B_4 cycle and B_8 cycle. Let R be the length of time the pipelined execution stalls. In this example, R is the time between E_7 cycle and B_8 cycle. We assume that the transfer of each unit of data by the DMAC takes the same amount of time and denote this time by DT. Let T be the total execution time of three processor cycles, E_5, E_6, and E_7. We can calculate m by the equation

$$m = \left\lceil \frac{\mathrm{T} - \mathrm{BMT}}{\mathrm{DT}} \right\rceil . \tag{3a}$$

Once m is known, we can calculate the time of stall R by the equation

$$R = \left\lceil \frac{m * \mathrm{DT} + 2 * \mathrm{BMT} - \mathrm{T}}{\mathrm{T_c}} \right\rceil * \mathrm{T_c}. \tag{3}$$

where $\mathrm{T_c}$ is the period of a processor clock cycle. The detail of the derivation for these equations can be found in our previous work [5].

Figure 5 illustrates how the cache-miss execution of the l-block shown in Figure 2c affects the

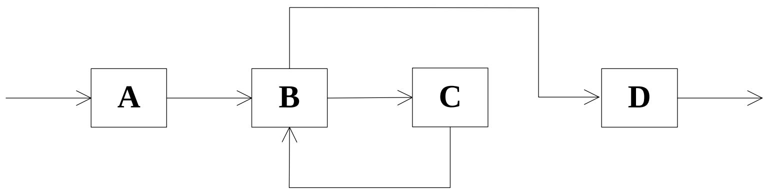


Figure 6: A simple loop

execution time of a successor when DMA I/O is concurrently executing. Figure 5a, a simplified version of Figure 4, shows the pipelined execution of the l-block when DMA I/O is present. Figure 5b shows the cache-hit reservation table of a successor. To determine the execution time of the successor, we concatenate the tail of the l-block's reservation table and the successor's cache-hit reservation table to obtain another reservation table shown in Figure 5c. The first four processor cycles of the new reservation table are E-cycles and the next one is a B-cycle. The pipelined execution stalls due to DMA I/O between the end of the first four E-cycles and the start of the next B-cycle. To determine the length R_s of stall, we need to determine if any DMA transfer crosses the execution of E_{10} cycle of the l-block and the first instruction of the successor. If there is one such DMA transfer, let b denote the length of time from when the DMA transfer starts to when the CPU is ready to fetch the first instruction of the successor, as shown in Figure 4. We call this length the *elapsed time of the crossing DMA transfer* in the l-block. We can calculate R_s using a slight modification of Eq. (3) by setting T to

$$\mathrm{T}_f + \mathrm{BMT} + b$$

where T_f now is the total execution time of the first four E-cycles. In contrast, if there is no such DMA transfer, we can use Eq. (3) directly to calculate R_s. Finally, in this example, the execution time of the l-block is $(R + 10 * \mathrm{T}_c)$ and the execution time of the successor is $(R_s + 6 * \mathrm{T}_c)$, if it has any successor, or $(R_s + 9 * \mathrm{T}_c)$ otherwise.

5 Iterative Integer Linear Programming

A DMA transfer can cross an l-block and its *predecessor* (i.e., an l-block executed immediately before it) when the predecessor ends with an E-cycle and the l-block causes a cache hit (i.e., begins with an E-cycle). As a result, to determine the set of all possible execution times of the l-block we must first analyze all its predecessors that end with an E-cycle. This requirement may lead to a cycle of dependencies. Let us take the loop structure in Figure 6 as an example. Each node in this graph represents an l-block, and each edge represents a control flow edge. Assume that the l-blocks B and C each end with an E-cycle. Because C is a predecessor of B, the execution time of B depends on C. Since B is a predecessor of C, the execution time of C depends on B. When we have a loop of dependencies like the one shown in Figure 6, we follow the control flow of the program iteratively to calculate the set of all possible executions times of each l-block. The iterative process may examine the same l-block repeatedly until its all possible execution times are determined. Since there is only a finite set of these values, this process will eventually terminate.

We describe the iterative algorithm below. This algorithm first determines the set of all possible execution times of each l-block and constructs a directed graph. It next assigns each execution time and each directed edge an execution count. A set of linear constraints on these execution counts are generated. Finally, it constructs the cost function of the program using these possible execution times and execution counts.

5.1 The `main` Procedure

Figure 7 shows the `main` procedure. This procedure requires as input the l-block control structure of the program to be analyzed and the instructions in each l-block. Let the first l-block in the program be denoted $B_{1,1}$. The procedure creates a directed graph $G = (V, D)$. Each node in V represents a reservation table of an l-block with stalls due to DMA I/O. The node is labeled with the corresponding execution time and execution count. Each directed edge between two nodes represents the fact that the pipelined execution of the l-block represented by the target node is affected by the tail of the reservation table of the l-block represented by the source node.

We use a variable L to hold the list of l-blocks waiting to be analyzed. For each l-block there is a list of edges whose targets are the cache-hit case of the l-block. These edges are yet to be

```
Input:   the $l$-block structure of a program, and
         $B_{1,1}$, the first $l$-block to be executed.
Output:  the cost function and a set of linear constraints.

Procedure:
1  Set $G = (V, D)$ to $(\emptyset, \emptyset)$.
   Set $L$ to $\{B_{1,1}\}$.
   Set unprocessed list of each $l$-block to $\emptyset$.
2  While $L$ is not empty do
3      – dequeue an $l$-block from $L$ and assign it to $B_{k,l}$;
4      – call analyze($B_{k,l}$) procedure.
5  For each $l$-block $B_{k,l}$ of the program do
6      – construct two linear constraints that bound the cache-hit count
         and the cache-miss count of the $l$-block.
7      – construct a linear constraint for the directed edges leaving each node.
8      – construct a linear constraint for the directed edges entering each node.
9      – construct the total execution time expression of the $l$-block.
10 Construct a cost function by summing each $l$-block's total execution time expression.
```

Figure 7: The `main` procedure

examined as a crossing DMA transfer may affect the execution time of the l-block; hence this list is called the *unprocessed list* of the l-block. This list is initially empty and is modified each time the l-block is analyzed.

Initially, L contains only the l-block $B_{1,1}$. The unprocessed list of $B_{1,1}$ is empty, denoted by $\emptyset$. Similarly, both the node set V and edge set D of the graph G are empty. During each iteration of the while loop (lines 2 to 4), the l-block (called $B_{k,l}$) at the head of L is dequeued and its execution time is analyzed by the procedure `analyze` (described below). The `analyze` procedure checks each successor of the l-block $B_{k,l}$. A successor is added to the list L if the successor is not yet examined (i.e., visited), or if `analyze` has added a new directed edge to the cache-hit case of the successor and the edge is not yet processed. This iterative process continues until L becomes empty, at which time all l-blocks and all directed edges between pairs of l-blocks have been analyzed.

Once a complete directed graph $G = (V, D)$ is obtained, we construct a set of linear constraints and the total execution time expression for each l-block in the program (lines 5 to 9). We first construct two linear constraints that bound the sum of the execution counts of all possible cache-hit

Input: an l-block $B_{k,l}$, and a directed graph $G = (V, D)$.
Output: an updated directed graph G.

Procedure:
```
1   If ($B_{k,l}$ is the first $l$-block in the program and has never been visited)
2     – update the node set $V$ accordingly.
3   Process each edge in $B_{k,l}$'s unprocessed list in the following manner:
4     (a) determine the execution behavior of $B_{k,l}$ when it causes a cache hit for the edge;
5     (b) check if there is a node in $V$ representing this execution behavior;
6     (c) if (such a node cannot be found) update $V$ accordingly;
7     (d) add a new edge to the edge set $D$;
8     (e) repeat the steps (a-d) when $B_{k,l}$ causes a cache miss;
9     (f) remove the edge from the unprocessed list and $D$.
10  Update $D$ accordingly for each node added to $V$.
11  Update the list $L$ appropriately.
```

Figure 8: The `analyze` procedure

(cache-miss) execution times of an l-block $B_{k,l}$ to be equal to its cache-hit count $x^h_{k,l}$ (cache-miss count $x^m_{k,l}$). We next construct a linear constraint that bounds the sum of the execution counts of the directed edges leaving each node in V to be equal to the execution count of the node, and a linear constraint that bounds the sum of the execution counts of the directed edges entering each node to be equal to the execution count of the node. Finally, we construct the total execution time expression of $B_{k,l}$ by summing the products of each possible execution time by its execution count.

The cost function for the program is constructed at the end of this procedure (line 10). The maximum value of the cost function under the set of linear constraints is an upper bound of the WCET of this program.

5.2 The `analyze` Procedure

This `analyze` procedure first checks if the l-block $B_{k,l}$ is the first l-block of the program and has never been visited. If $B_{k,l}$ is such an l-block, it adds two nodes of $B_{k,l}$ to the node set V: a node that represents the cache-hit reservation table and a node that represents the cache-miss reservation table, both including the stalls due to DMA I/O. Each node is labeled with the corresponding execution time and its execution count (lines 1 to 2).

If there are edges in the unprocessed list of $B_{k,l}$, the procedure processes each edge in the list in turn and removes it from the list and the edge set D (lines 3 to 9). It determines the pipelined execution behavior of $B_{k,l}$ in the way described in Section 4 for each edge in the list. In the cache-hit case, the procedure concatenates the tail of the reservation table represented by the source of an edge and $B_{k,l}$'s cache-hit reservation table to obtain another reservation table. It next uses Eq. (3) to calculate the amount of time the CPU stalls during the execution of the new reservation table. The procedure then checks whether this execution behavior is the same as the one represented by any of $B_{k,l}$'s nodes in V. If it finds such a node in V, it adds to D a new edge from the source of the original edge to the node. Otherwise, it adds to V a new node representing the new reservation table and add to D a new edge from the source node of the original edge to the new node. It next determines the pipelined execution behavior of $B_{k,l}$ when it causes a cache miss in a similar manner.

For each node added to V, the procedure adds to the edge set D an edge from the node to each successor and adds the edge to the unprocessed list of the successor (line 10). Finally, if it added any node to V, it adds all $B_{k,l}$'s successor l-blocks to the list L (line 11).

6 Experimental Results

We conducted extensive experiments to demonstrate the effectness of our method on bounding the WCET of programs executing concurrently with DMA I/O on a dynamic architecture. We evaluated the performance of our method by comparing our WCET predictions with the traditional pessimistic predictions for several sample programs. In the following we first describe the control flow of the experiment. We next describe the experimental results.

6.1 The Control Flow

Figure 9 describes the control flow of the experiment. Table 1 lists the sample programs in our tested workload. For each sample program, we compiled it into a MC68030 assembly program and executed the assembly program on a MC68030 simulator with the worst-case data set to obtain the worst-case execution trace. We identified the worst-case data set of each sample program by a careful study of the program. We used the MC68030 in this experiment because it is a widely-used

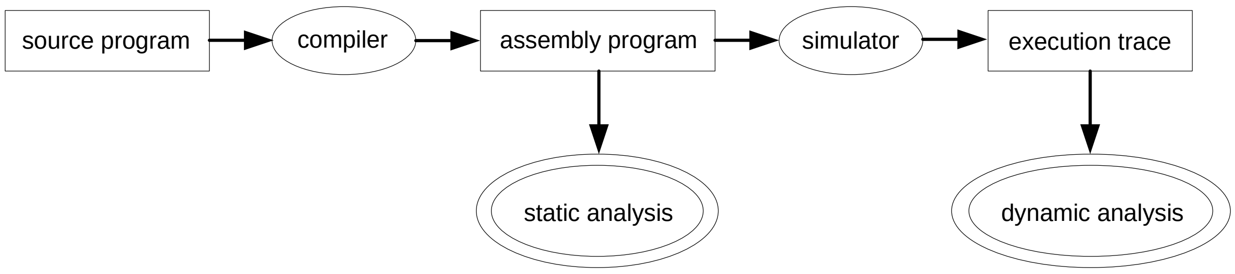


Figure 9: The control flow of the experiment

Name	Description
Sels	a selection-sort function for 50 elements.
Bubs	a bubble-sort function for 50 elements.
Mtxm	a matrix-multiplication function for two 10x10 matrices.
Mtx2	similar to **Mtxm** but unrolls the whole innermost loop.

Table 1: The tested set of programs

embedded microprocessor for which instruction timing information is available.

This experiment is divided into two parts: *static analysis* and *dynamic analysis*. In the static analysis part, we compared our WCET prediction with the pessimistic WCET prediction of the program. In the dynamic analysis part, we compared our WCET prediction with the pessimistic WCET prediction of the worst-case execution trace of the program.

We implemented our methods in the form of a timing tool. The timing tool contains about 15,000 lines of C++ code, part of which is obtained from a tool called `Cinderella`, developed by Li *et al* [9]. Our timing tool takes as input an MC68030 assembly program. As required by all previous work [1, 3–6, 8–13], the user needs to provide loop bound information on the analyzed program. The user can also provide additional information in form of linear constraints to tighten the WCET prediction.

The MC68030 microprocessor contains an on-chip 256-byte direct-mapped instruction cache which is organized as 16 16-byte lines. It also uses an instruction pipeline which allows as many as two instructions to be executed simultaneously. We obtained the timing information of each instruction from the Motorola 68030 manual [2]. The clock frequency of the microprocessor was 20 MHz, and the period of a clock cycle T_c was 50 ns. A 0-wait memory was used, and each DMA transfer took two clock cycles. Hence, we set DT to 100 ns. Finally, BMT was 5 ns.

Name	# of instructions	% of long instructions
Sels	11,713	0%
Bubs	21,335	0%
Mtxm	40,789	10%
Mtx2	10,592	20%

Table 2: The features of tested traces

6.1.1 Static Analysis

Given an assembly program, we first computed its WCET when it executes alone. We denote this value by $\mathcal{A}_s$. We next used our method to compute the WCET of the program when it executes concurrently with DMA I/O and denote this value by $\mathcal{W}_s$. We compared $\mathcal{W}_s$ with $\mathcal{A}_s$ to measure how much DMA I/O extends the WCET of the program. In addition, we computed the maximum units of data the DMAC can transfer during the execution of the worst-case execution path. We denote this value by $\mathcal{M}_s$. We also used a pessimistic method to predict the WCET of the concurrent execution of the program and a DMA I/O operation that transfers $\mathcal{M}_s$ units of data. The pessimistic method bounds the WCET by the sum of $\mathcal{A}_s$ and the execution time of the DMA I/O operation when it is carried out alone. We denote this pessimistic prediction by $\mathcal{W}_s^a$. We measure the effectiveness of our method by the percentage P_s of reduction from the pessimistic prediction, i.e.,

$$P_s = \frac{\mathcal{W}_s^a - \mathcal{W}_s}{\mathcal{W}_s^a} * 100\%$$

6.1.2 Dynamic Analysis

The approach we took to demonstrate the improvement of our method on each program's worst-case execution trace is similar to the one used in the static analysis. We first computed the execution time of a trace when it executes alone and denote this value by $\mathcal{A}_d$. We next simulated the concurrent execution of the trace and DMA I/O to find the execution time of the trace when it executes concurrently with DMA I/O and the number of units of data the DMAC transfers. We denote them by $\mathcal{W}_d$ and $\mathcal{M}_d$, respectively. We compared $\mathcal{W}_d$ with $\mathcal{A}_d$ to measure how much DMA I/O shows down the execution of the trace. The trace can be treated as a program with only straight-line code. Since the program contains only one execution path, our WCET prediction was exactly the same as $\mathcal{W}_d$. In other words, given the trace and a DMA I/O operation that transfers $\mathcal{M}_d$ units of data, both of which are ready at the same time, our method bounds the WCET by $\mathcal{W}_d$. Let $\mathcal{W}_d^a$ denote the pessimistic WCET prediction of the concurrent execution of the trace and the DMA I/O operation. We measure the effectiveness of our method by the percentage of reduction from the pessimistic prediction, i.e.,

$$P_d = \frac{\mathcal{W}_d^a - \mathcal{W}_d}{\mathcal{W}_d^a} * 100\%$$

In addition, we evaluated the accuracy of our method by comparing the execution time $\mathcal{W}_d$ of the worst-case execution trace and our WCET prediction $\mathcal{W}_s$ of the structured program.

Column 2 of Table 2 lists the number of instructions in the worst-case execution trace of each program. To investigate the relationship between the performance of our method and the computational requirement of a program, we classify all instructions into two categories: long instructions and short instructions. An instruction is a long instruction if, during its execution, the CPU does not need the bus for 10 processor clock cycles or more. In contrast, during the execution of a short instruction, the CPU never allows any I/O device to have the bus for such a long period. Generally speaking, long instructions do intensive computation, and short instructions do data movement or simple computation. For example, the instructions `MULU.W D1,D2` and `DIVU.W D2,D0` are long instructions and `MOVE.L (A3)+,D0` and `ADD.L D0,D1` are short instructions. We tested programs with different computational requirement in the experiment. Column 3 of Table 2 gives the percentage of long instructions in each trace. Among the tested program, **Mtx2** is obtained by unrolling the innermost loop of **Mtxm**. As shown in Table 2, this loop-unrolling procedure significantly increases the percentage of long instructions.

6.2 Experimental Results

We first used the method described in Section 5 to compute the WCET of a program when it executes concurrently with DMA I/O on a dynamic architecture. A similar method without considering the stalls caused by DMA I/O can be used to compute the WCET of the program when it executes alone. Table 3 shows the experimental results of the static analysis. In order to study the relationship between the reduction percentage P_s and the size of the instruction cache, we

4 line instruction cache

Name	$\mathcal{W}_s/\mathcal{A}_s$	$\mathcal{W}_s^a/\mathcal{A}_s$	P_s
Sels	1.05	1.84	43%
Bubs	1.07	1.84	42%
Mtxm	1.03	1.64	37%
Mtx2	1.02	1.71	40%

8 line instruction cache

Name	$\mathcal{W}_s/\mathcal{A}_s$	$\mathcal{W}_s^a/\mathcal{A}_s$	P_s
Sels	1.06	1.88	44%
Bubs	1.07	1.85	42%
Mtxm	1.03	1.89	46%
Mtx2	1.02	1.71	40%

16 line instruction cache

Name	$\mathcal{W}_s/\mathcal{A}_s$	$\mathcal{W}_s^a/\mathcal{A}_s$	P_s
Sels	1.06	1.88	44%
Bubs	1.07	1.85	42%
Mtxm	1.03	1.94	47%
Mtx2	1.02	1.81	44%

32 line instruction cache

Name	$\mathcal{W}_s/\mathcal{A}_s$	$\mathcal{W}_s^a/\mathcal{A}_s$	P_s
Sels	1.06	1.88	44%
Bubs	1.07	1.85	42%
Mtxm	1.03	1.94	47%
Mtx2	1.02	1.92	47%

Table 3: The static-analysis experimental results

conducted the same experiment on processor configurations with instruction caches of 4, 8, 16, and 32 16-byte cache lines. Columns 2 and 3 give the values of $\mathcal{W}_s$ and $\mathcal{W}_s^a$, respectively, after each is normalized to $\mathcal{A}_s$. Column 4 gives the value of P_s for each program. A program has a higher cache-hit ratio when it executes on a process configuration with more cache lines. Consequently, when a program executes on a processor configuration with more cache lines, it has a larger value of P_s.

Table 4 shows the dynamic-analysis results of the four experiments with 4, 8, 16, and 32 cache lines. Column 3 gives the cache-hit ratio of each trace. Column 4 gives the bus utilization of each trace when it executes alone. The bus utilization of a trace is the amount of time the CPU uses the system bus to the execution time of the trace. Because a trace that has a higher percentage of long instructions spends more time in computation, it has a lower bus utilization. Column 5 and 6 give the values of $\mathcal{W}_d$ and $\mathcal{W}_d^a$, respectively, after each is normalized to $\mathcal{A}_d$. Column 7 gives the value of P_d for each trace. We can see that our method produces a larger reduction percentage P_d for a trace with a higher hit ratio and a larger percentage of long instructions. Column 8 gives the value of $\mathcal{W}_d/\mathcal{W}_s$ on each processor configuration. The fact $\mathcal{W}_d/\mathcal{W}_s \leq 1$ for any tested program shows that our WCET prediction $\mathcal{W}_s$ safely bounds the execution time $\mathcal{W}_d$ of the worst-case execution path of the program. A program is *deterministic* if it contains only an execution path. Among the tested programs, **Mtxm** and its loop-unrolled version **Mtx2** are deterministic. An execution trace

4 line instruction cache

Name	% of long instructions	cache-hit ratio	bus utilization	$\mathcal{W}_d/\mathcal{A}_d$	$\mathcal{W}_d^a/\mathcal{A}_d$	P_d	$\mathcal{W}_d/\mathcal{W}_s$
Sels	0%	0.98	0.25	1.05	1.79	41%	0.54
Bubs	0%	1.00	0.23	1.07	1.82	41%	0.49
Mtxm	10%	0.80	0.39	1.03	1.64	37%	1.00
Mtx2	20%	0.80	0.30	1.02	1.71	40%	1.00

8 line instruction cache

Name	% of long instructions	cache-hit ratio	bus utilization	$\mathcal{W}_d/\mathcal{A}_d$	$\mathcal{W}_d^a/\mathcal{A}_d$	P_d	$\mathcal{W}_d/\mathcal{W}_s$
Sels	0%	1.00	0.17	1.06	1.87	43%	0.52
Bubs	0%	1.00	0.21	1.07	1.84	42%	0.48
Mtxm	10%	0.99	0.13	1.03	1.89	46%	1.00
Mtx2	20%	0.80	0.30	1.02	1.71	40%	1.00

16 line instruction cache

Name	% of long instructions	cache-hit ratio	bus utilization	$\mathcal{W}_d/\mathcal{A}_d$	$\mathcal{W}_d^a/\mathcal{A}_d$	P_d	$\mathcal{W}_d/\mathcal{W}_s$
Sels	0%	1.00	0.17	1.06	1.87	43%	0.52
Bubs	0%	1.00	0.21	1.07	1.84	42%	0.48
Mtxm	10%	1.00	0.10	1.03	1.94	47%	1.00
Mtx2	20%	0.90	0.20	1.02	1.82	44%	1.00

32 line instruction cache

Name	% of long instructions	cache-hit ratio	bus utilization	$\mathcal{W}_d/\mathcal{A}_d$	$\mathcal{W}_d^a/\mathcal{A}_d$	P_d	$\mathcal{W}_d/\mathcal{W}_s$
Sels	0%	1.00	0.17	1.06	1.87	43%	0.52
Bubs	0%	1.00	0.21	1.07	1.84	42%	0.48
Mtxm	10%	1.00	0.11	1.03	1.93	47%	1.00
Mtx2	20%	1.00	0.08	1.02	1.92	47%	1.00

Table 4: The dynamic-analysis experimental results

of a deterministic program is its only execution path, and the execution time of the trace is the actual WCET of the program. For each of the **Mtxm** and **Mtx2** programs, the execution time $\mathcal{W}_d$ of the trace is equal to our WCET prediction $\mathcal{W}_s$ of the program, i.e., $\mathcal{W}_d/\mathcal{W}_s = 1$. This fact shows that our method does not impose any pessimistic assumptions and, therefore, tightly bounds the WCET of a program executing concurrently with DMA I/O.

7 Concluding Remarks

In this paper we presented an iterative integer linear programming method that can be used to bound the performance of a program executed on a hard-real-time embedded system where the execution time of each instruction depends on not only itself but also its adjacent instructions. We illustrate the capability of this method with an architecture where a processor has an instruction cache and an instruction pipeline, and cycle-stealing DMA I/O is concurrently executing. The experimental results show that our method safely and tightly bound the WCET of a program executed on such an architecture. As we do not impose any architecture-specific restriction in our iterative integer linear programming method, we believe our method can be easily adapted to accurately bound the WCET of a program executed on other dynamic architectures.

References

[1] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, May 2000.

[2] *MC68030 Enhanced 32-bit Microprocessor: User's Manual.* Motorola, 1987.

[3] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1997.

[4] C. Healy, R. Arnold, F. Muller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions of Computers*, 48(1):53–70, January 1999.

[5] Tai-Yi Huang and Jane W.-S. Liu. Predicting the worst-case execution time of the concurrent execution of instructions and cycle-stealing DMA I/O operations. *ACM SIGPLAN Notices*, 30(11), November 1995.

[6] Tai-Yi Huang, Jane W.-S. Liu, and David Hull. A method for bounding the effect of DMA I/O interference on program execution time. In *Proceedings of the 17th Real-Time System Symposium*, pages 275–285, December 1996.

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

[8] Yan-Tsun Steve Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–561, June 1995.

[9] Yan-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software.* Kluwer Academic Publishers, 1999.

[10] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th Real-Time System Symposium*, pages 334–345, December 1998.

[11] Frank Muller, David Whalley, and Marison Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1994.

[12] Chang-Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, pages 48–57, May 1991.

[13] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th Real-Time System Symposium*, pages 144–153, December 1998.

[14] *The VMEbus Specification.* Motorola, 1985.