Performance analysis of hard-real-time embedded software

Tai-Yi Huang*, Kuang-Li Huang and Yeh-Ching Chung

Department of Computer Science,
National Tsing Hua University, Hsinchu, Taiwan 300, ROC
E-mail: tyhuang@cs.nthu.edu.tw
E-mail: klhuang@cs.nthu.edu.tw
E-mail: ychung@cs.nthu.edu.tw
*Corresponding author

Abstract: The execution time of an instruction depends on its adjacent instructions and I/O activities. Our method first iteratively determines the set of all possible execution times of each instruction. We next construct a set of linear constraints on their execution counts. The maximum value of the cost function is an upper bound of the worst-case execution time. We demonstrate the capability of this method on a machine model where a processor has an instruction cache and pipeline, and cyclestealing DMA I/O is concurrently executing. The experimental results show that our method safely and tightly bounds the worstcase execution time.

Keywords: hard-real-time systems; worst-case execution time; WCET; integer linear programming; ILP; embedded software.

Reference to this paper should be made as follows: Huang, T-Y., Huang, K-L. and Chung, Y-C. (2006) 'Performance analysis of hard-real-time embedded software', *Int. J. Embedded Systems*, Vol. 2, Nos. 3/4, pp.209–221.

Biographical notes: Tai-Yi Huang received the BS Degree in Computer Science and Information Engineering from National Taiwan University in 1991. He received both the MS and PhD Degrees from University of Illinois at Urbana-Champaign in computer science in 1994 and 1996, respectively. From 1996 to 2001, he was a Software Design Engineer in Windows OS Kernel Performance Group, Microsoft Inc. He joined the Computer Science Department at National Tsing Hua University as an Assistant Professor in 2002 and became an Associate Professor in 2006. His research interests include low-power embedded systems, real-time operating systems, and high-performance clustered storages. He is a member of the IEEE and the ACM.

Kuang-Li Huang received the BS Degree in Electrical Engineering from Chung-Yuan Christian University in 2002 and the MS Degree in Computer Science from National Tsing Hua University, Taiwan, in 2004. He is currently a PhD student in the Computer Science Department at National Tsing Hua University. His research interests include P2P look up protocols, distributed systems, and operating systems.

Yeh-Ching Chung received the BS Degree in Information Engineering from Chung Yuan Christian University in 1983, and the MS and PhD Degrees in Computer and Information Science from Syracuse University in 1988 and 1992, respectively. He joined the Department of Information Engineering at Feng Chia University as an Associate Professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the Chairman of the Department. In 2002, he joined the Department of Computer Science at National Tsing Hua University as a Full Professor. His research interests include parallel and distributed processing, pervasive software, and system software for SOC design. He is a member of the IEEE Computer Society and ACM.

1 Introduction

In a hard-real-time embedded system, each task must complete the execution by its deadline. A task that executes longer than its allocated computation time may lead to missed deadlines and the failure of the whole system. In such a system, it is required that the WCET of each program be known in advance. This knowledge is also often required in the schedulability analysis of hard-realtime systems (Liu and Layland, 1973; Sha et al., 1990; Sun et al., 1997). For this reason, the problem of bounding the WCET of a program has received a great deal of attentions in recent years.

This paper presents an iterative I-IPET for bounding the WCET of a hard-realtime embedded program executed on a dynamic architecture where the execution time of an instruction varies, depending on the execution of the instruction itself, its adjacent instructions, and concurrent I/O activities. The IPET, first developed by Li and Malik (1995), converted the problem of bounding the WCET of a program into one of solving a set of Integer Linear Programming (ILP) problems. This method was later extended by Li and Malik (1999) to take into account the interferences of instruction caching (direct-mapped and set-associative) and data caching. Without loss of generality, we illustrate the capability of the I-IPET approach on a machine model with instruction caching, instruction pipelining and cycle-stealing DMA I/O. However, our I-IPET approach can be easily adapted to consider the effect of other modern architectures or tighten the WCET predictions in addition to bounding the interference of cycle-stealing DMA I/O.

Our I-IPET approach builds on top of the cache-aware IPET methodology (Li and Malik, 1999) to bound the WCET of a program executing concurrently with an independent cycle-stealing DMA I/O operation. DMA Controllers (DMAC) are commonly used in hard-real-time embedded systems to reduce the CPU usage on interrupt service routines. A DMAC that operates in the cycle-stealing mode transfers data by 'stealing' bus cycles from the executing program. Bus contention between the executing program and the cycle-stealing DMA I/O operation retards the progress of both and extends their execution times. Our I-IPET approach 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 behaviour 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 called *DMA*-bounding linear constraints. These possible execution times, execution counts, DMA-bounding and cache-bounding linear constraints are used as inputs to an ILP problem, the solution of which is an upper bound of the WCET of the program being analysed.

To demonstrate the efficacy of our method on bounding the WCET, we conducted extensive experiments 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 our machine model. For the sake of simplicity, we present our I-IPET approach in an incremental way. Section 4 describes the methodology for bounding the interference of instruction caching and cycle-stealing DMA I/O. Section 5 extends the methodology in Section 4 to include the effect of instruction pipelining. We present our experimental results in Section 6. Finally, Section 7 gives the concluding remarks.

2 Related work

A number of methods have been developed by different research groups to predict the WCET of a program (Colin and Puaut, 2000; Engblom and Ermedahl, 1999, 2000; Engblom et al., 2001; Ferdinand et al., 1997; Ferdinand and Wilhelm, 1999; Gupta and Gopinath, 1994; Healy et al., 1999; Kim et al., 1999; Lim et al., 1998; Lundqvist and Stenström, 1999a, 1999b; Mueller et al., 1994; Ottosson and Sjödin, 1997; Park and Shaw, 1991; Puschner and Koza, 1989; Stappert and Altenbernd, 2000). Shaw (1989) first proposed a timing schema to represent the execution time of a program. Park and Shaw (1991) later extended the schema to eliminate infeasible execution paths (i.e., paths that can never be executed) and tighten the WCET prediction. Similarly, Puschner and Koza (1989) introduced several new language constructs with which programmers can describe the timing behaviour of a program. Muller et al. (1994) developed a static cache simulation to bound the WCET of a program executed on a contemporary machine with an instruction cache. Lim et al. (1998) proposed a timing analysis technique for modern multiple-issue machines such as superscalar processors. Kim et al. (1999) presented quantitative analysis results on the impacts of various architecture features on the accuracy of WCET predictions. Lundqvist and Stenström (1999a) extended cycle-level architectural simulation techniques to calculate WCET predictions on high-performance processors.

Recently, the IPET methodology has been widely used to determine the WCET of a program (Engblom et al., 2001; Healy et al., 2000; Huang et al., 1996; Li and Malik, 1995, 1999; Ottosson and Sjödin, 1997; Theiling, 2002; Theiling et al., 2000). Li and Malik (1995) presented the first IPET approach to convert the problem of bounding the WCET into one of solving a set of ILP constraints. Users can provide path information in the form of linear constraints, called *path-bounding* linear constraints, to eliminate infeasible paths and tighten the WCET prediction. Li and Malik (1999) extended their IPET approach to include the timing analysis of direct-mapped instruction caches. Engblom et al. (2001) converted complex control-flow information into a set of linear constraints for tighter WCET predictions. Healy et al. (2000) described several complementary methods to automatically bound loop iterations with linear constraints. Theiling et al. (2000) adopted abstract interpretation to analyse the performance of modern hardware architectures and used the IPET approach to find the longest execution path. Among all the IPET extensions, our work is the only one that attempts to consider the interference of cycle-stealing DMA I/O and the first one that used an iterative IPET approach to model the performance of modern hardware architectures.

The first IPET approach (Li and Malik, 1995) decomposes a program into a number of basic blocks.

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. Let x_i be the execution count of the basic block B_i. 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. To analyse the effect of instruction caching, Li and Malik (1995) further partition 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. Let $c^h_{i,j}$ and $c^m_{i,j}$ denote the cache-hit and cache-miss execution times of the *l*-block $B_{i,j}$ and let $x^h_{i,j}$ and $x^m_{i,j}$ 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.,

$$x_i = x^h_{i,j} + x^m_{i,j}, \quad j = 1,\ldots,n_i,$$

where n_i is the number of *l*-block in B_i. Let N be the number of basic blocks in a program. The cost function of the execution time of the program is

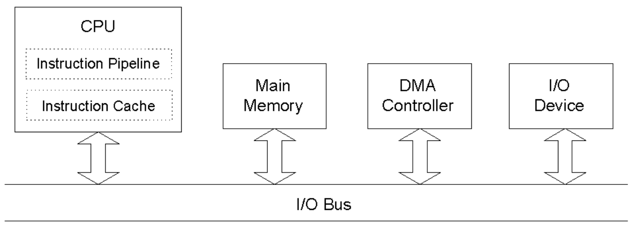
$$\sum_{i=1}^{N}\sum_{j=1}^{n_i}(c^h_{i,j}x^h_{i,j} + c^m_{i,j}x^m_{i,j}). \tag{1}$$

A set of cache-bounding linear constraints on $x^h_{i,j}$ and $x^m_{i,j}$ are generated to bound the relationship of each *l*-block. The maximum value of the cost function in equation (1) under the path-bounding linear constraints on the x_i's and the cache-bounding linear constraints on the $x^h_{i,j}$'s and $x^m_{i,j}$ ' s is an upper bound of the WCET of the program executed alone on a machine model with instruction caching.

3 The machine model

We develop our I-IPET approach on top of the cost function in equation (1) and the path-bounding and the cache-bounding linear constraints. We adopt here the commonly-used machine model shown in Figure 1. The program being analysed is executed in the CPU with instruction caching and pipelining. An independent DMA I/O operation, issued by another program, is executed concurrently by the DMAC. The DMAC operates in the cycle-stealing mode. Either the CPU or the DMAC, but not both, can hold the bus and transfer data at any time instant. For the sake of concreteness, we assume that bus contention between the CPU and the DMAC is regulated according to the VMEbus (Black et al., 1981) bus access protocol. This protocol is sufficiently general that our analysis may be easily applied to many other commonly-used bus protocols.

Figure 1 The machine model



An *instruction cycle* consists of a sequence of operations to fetch and execute an instruction. The sequence takes one or more machine cycles. A machine cycle requires one to several processor clock cycles to execute. We assume that the CPU is synchronous: the beginning of each machine cycle is triggered by the processor clock. We classify all machine cycles into one of two categories: bus-access (B) cycles and execution (E) cycles. B-cycles are those machine cycles during which the CPU uses the I/O bus. In contrast, during E-cycles, the CPU does not use the bus. In general, there may be several consecutive E-cycles in an instruction cycle.

The DMAC transfers data only when the CPU is in E-cycles. Let m be the maximum units of data the DMAC can transfer during the sequence of cycles, $B_i \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots E_k \rightarrow B_{i+1}$. When the CPU enters E_1 from B_i, there is a short delay called the Bus Master Transfer (BMT) time, while the DMAC gains control of the bus. 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_{i+1} from E_k. The DMAC checks whether there is any pending bus request only 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 control of the bus and enters the B_{i+1} 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 the k consecutive E-cycles. We can compute m by the equation

$$m = \left\lceil \frac{T - \text{BMT}}{\text{DT}} \right\rceil. \tag{2a}$$

The worst-case delay suffered by the CPU execution of the sequence of machine cycles is

$$r = \left\lceil \frac{m \times \text{DT} + 2 \times \text{BMT} - T}{T_c} \right\rceil \times T_c, \tag{2b}$$

where T_c is the period of a clock cycle. The details of the derivation for these equations can be found in Huang and Liu (1995).

4 Instruction catching and DMA I/O

In this section we extend the cost function in equation (1) to bound the WCET of a program executing concurrently with DMA I/O on a machine model with instruction caching. To simplify the discussion, we assume that instruction pipelining in the machine model shown in Figure 1 is disabled. We will later enable instruction pipelining in the next section.

When DMA I/O is concurrently executing, the execution time of an *l*-block is equal to the sum of its execution time when it executes alone and the delay caused by DMA I/O. A DMA transfer can cross two *l*-blocks if the first *l*-block ends with an *E*-cycle and the second *l*-block begins with an *E*-cycle (i.e., it causes a cache hit). Consequently, the delay suffered by an *l*-block varies, depending upon the concurrent execution of DMA I/O, the *l*-block itself and its adjacent *l*-blocks. Our I-IPET approach follows the control flow of the program iteratively to determine the set of possible execution times of each *l*-block. The method may examine an *l*-block repeatedly and will terminate only after all possible execution times have been determined.

In the following, we first describe the methodology to determine the set of possible execution times of an *l*-block when it causes a cache miss. We next discuss the case when it causes a cache hit. Suppose that, when DMA I/O is present, our I-IPET approach determines that an *l*-block $B_{k,i}$ has, totally, $t_{k,i}$ possible execution times when it causes a cache hit. Let $c^h_{k,l,1}, c^h_{k,l,2}, \ldots, c^h_{k,l,t_k,l}$ denote these execution times. Let

denote the execution counts of these execution times, respectively. The sum of these execution counts is equal to the cache-hit count of $B_{k,l}$, i.e.,

$$x^h_{k,l} = x^h_{k,l,1} + x^h_{k,l,2} + \cdots + x^h_{k,l,t_k,l}. \tag{3}$$

Similarly, suppose that $B_{k,i}$ has totally $s_{k,i}$ possible execution times when it causes a cache miss and these execution times are denoted by $c^m_{k,l,1}, c^m_{k,l,2}, \ldots, c^m_{k,l,s_k,l}$. Let $x^m_{k,l,1}, x^m_{k,l,2}, \ldots, x^m_{k,l,s_k,l}$ denote their execution counts. The sum of these execution counts is equal to the cache-miss count of $B_{k,l}$, i.e.,

$$x^m_{k,l} = x^m_{k,l,1} + x^m_{k,l,2} + \cdots + x^m_{k,l,s_k,l}. \tag{4}$$

We can, therefore, express the total execution time of $B_{k,l}$ as

$$\sum_{d=1}^{t_{k,l}} c^h_{k,l,d} x^h_{k,l,d} + \sum_{d=1}^{s_{k,l}} c^m_{k,l,d} x^m_{k,l,d}. \tag{5}$$

The cost function in equation (1) should be replaced by the new cost function

$$\sum_{i=1}^{N} \sum_{j=1}^{n_i} \left\{ \sum_{d=1}^{t_{i,j}} c^h_{i,j,d} x^h_{i,j,d} + \sum_{d=1}^{s_{i,j}} c^m_{i,j,d} x^m_{i,j,d} \right\}. \tag{6}$$

when we want to take into account of the effect of DMA I/O. We bound the WCET of the program by the maximum value of the new cost function under a set of path-bounding linear constraints on the x_i's, a set of cache-bounding linear constraints on the $x^h_{i,j}$'s and $x^m_{i,j}$'s and a set of DMA-bounding linear constraints on the $x^h_{i,j,d}$'s and $x^m_{i,j,d}$'s.

The rationale that can be used to determine the set of possible execution times of each *l*-block $B_{k,l}$ and the DMA-bounding linear constraints on their execution count is presented next. We describe the I-IPET approach that implements the rationale at the end of this section.

4.1 Cache-miss execution times

An instruction that executes alone has two possible instruction cycles; one when it causes a cache hit and one when it causes a cache miss. For the sake of simplicity, we assume that querying the status of the instruction cache is instantaneous. Therefore, the cache-hit instruction cycle begins with an *E*-cycle to fetch the instruction from the on-chip instruction cache, and the cache-miss instruction cycle begins with one or more *B*-cycles to fetch the instruction and the subsequent instructions in the *l*-block from the main memory. Due to the iterative analysis, however, our I-IPET approach can be easily adapted to work on other machine models that may require extra *E*-cycles to determine whether the requested *l*-block is in the instruction cache.

We obtain the cache-miss sequence of machine cycles of the *l*-block $B_{k,l}$ by concatenating the cache-miss instruction cycle of the first instruction and the cache-hit instruction cycles of the rest of the instructions in the *l*-block. We denote this sequence of cycles by $B^m_{k,l}$.

The delay suffered by $B^m_{k,l}$ due to DMA I/O is the sum of the delays suffered by all *E*-cycle sequences in $B^m_{k,l}$. Because $B^m_{k,l}$ begins with a *B*-cycle, a *predecessor* of the sequence (i.e., an instruction executed immediately before this sequence) does not affect the execution time of $B^m_{k,l}$. On the other hand, if $B^m_{k,l}$ ends with an *E*-cycle, a DMA transfer can cross $B^m_{k,l}$ and a *successor* of the sequence (i.e., an instruction executed immediately after the sequence) if the successor causes a cache hit and $B^m_{k,l}$ may affect the execution time of the successor. Consequently, the set of possible execution times of $B^m_{k,l}$ depends on whether $B^m_{k,l}$ ends with a *B*-cycle or an *E*-cycle.

$B^m_{k,l}$ ends with a B-cycle. If $B^m_{k,l}$ ends with a B-cycle, no DMA transfer can cross it or any of its successors. We can, therefore, use equation (2) to calculate the delay suffered by each *E*-cycle sequence in $B^m_{k,l}$. We then obtain an execution time of $B^m_{k,l}$ by summing $c^m_{k,l}$ and the delays of all *E*-cycle sequences. It is the only possible execution time of $B^m_{k,l}$ in this case. In other words, the number $s_{k,l}$ of possible execution times of $B^m_{k,l}$ is one.

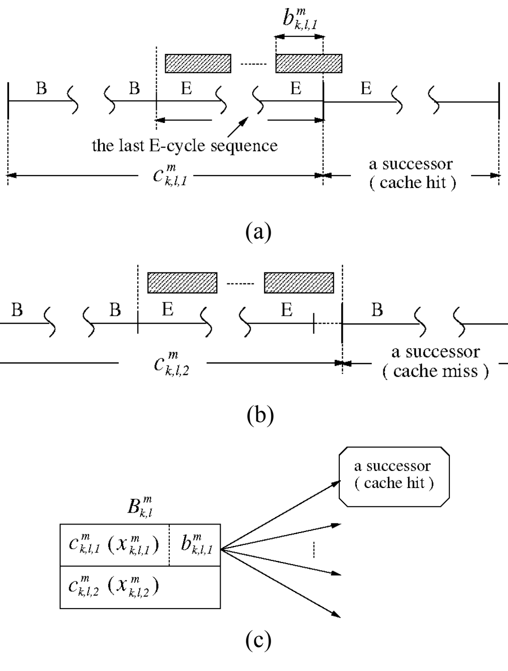
$B^m_{k,l}$ ends with an E-cycle. If $B^m_{k,l}$ ends with an *E*-cycle, the delay suffered by the last *E*-cycle sequence in $B^m_{k,l}$ depends on whether its successor causes a cache miss or a cache hit. This fact is illustrated by Figure 2. We first use equation (2) to calculate the delay suffered by each *E*-cycle sequence except the last one. When its successor causes a cache hit, the last *E*-cycle sequence suffers no delay because a DMA transfer can cross $B^m_{k,l}$ and the successor, as shown in Figure 2(a). In contrast, when its successor causes

a cache miss, as shown in Figure 2(b), the last E-cycle sequence suffers a delay that is given by equation (2). Hence, $B^m_{k,l}$ has totally two possible execution times: $c^m_{k,l,1}$, which is equal to $c^m_{k,l}$, plus the total delays suffered (due to concurrent DMA I/O) by all but the last E-cycle sequence; and $c^m_{k,l,2}$, which is equal to $c^m_{k,l,1}$ plus the delay suffered by the last E-cycle sequence. $s_{k,l}$ is two in this case.

4.2 The execution-time-dependency graph

We represent the dependency of the execution times of $B_{k,l}$ on the behaviour of its predecessors and successors by an *execution-time-dependency graph*, a part of which is illustrated in Figure 2(c). Each node under $B^m_{k,l}$ represents a possible execution behaviour of $B^m_{k,l}$ that leads to a possible execution time of this l-block; the node is labelled with the corresponding execution time and execution count. Here, the node labelled $c^m_{k,l,1}(x^m_{k,l,1})$ represents the case shown in Figure 2(a) and the node labelled $c^m_{k,l,2}(x^m_{k,l,2})$ represents the case shown in Figure 2(b). In the former case (where a successor of $B^m_{k,l}$ causes a cache hit), the last DMA transfer in $B^m_{k,l}$ affects the delay suffered by the first E-cycle sequence of a successor. We represent this fact by a directed edge from this node (the *source* of the edge) to the node representing the cache-hit case of a successor (the *target* of the edge). We will later replace the target of the edge by a node representing a specific execution behaviour of this successor. In general, a directed edge represents the fact that the delay suffered by the first E-cycle sequence in the l-block represented by the target node is affected by the last DMA transfer in the l-block represented by the source node.

Figure 2 The execution times of $B^m_{k,l}$ if it ends with an E-cycle: (a) the execution time of $B^m_{k,l}$ when its successor causes a cache hit; (b) the execution time of $B^m_{k,l}$ when its successor causes a cache miss and (c) the dependence on the successors of $B^m_{k,l}$



$b^m_{k,l,1}$
B
B
E
E
E
the last E-cycle sequence
$c^m_{k,l,1}$
a successor (cache hit)
(a)
B
B
E
E
B
$c^m_{k,l,2}$
a successor (cache miss)
(b)
$B^m_{k,l}$
$c^m_{k,l,1}$ $(x^m_{k,l,1})$
$b^m_{k,l,1}$
$c^m_{k,l,2}$ $(x^m_{k,l,2})$
a successor (cache hit)
(c)

An I-IPET implementation, which we will describe at the end of this section, constructs and uses such a graph. This graph may be modified as each l-block is analysed in the manner described in this section. In our subsequent discussion, rather than saying a DMA transfer (or a cycle) in an l-block when the execution behaviour of the l-block is represented by a node, we will simply say "a DMA transfer (or a cycle) in the node". Let $b^m_{k,l,1}$ denote the length of time from when the last DMA transfer in the node labelled $c^m_{k,l,1}(x^m_{k,l,1})$ starts to when the last E-cycle sequence in the node ends, as shown in Figure 2(a). We call this length the *elapsed time of the last DMA transfer* in the node.

We add a directed edge from the node labelled $c^m_{k,l,1}(x^m_{k,l,1})$ to the cache-hit case of each successor, as shown in Figure 2(c). We assign each directed edge with an execution count that represents the number of times the control flow passes from the source node to the target node of the edge. The execution counts of these edges satisfy the following constraint.

Constraint 1: *The sum of the execution counts of the directed edges leaving the node labelled* $c^m_{k,l,1}(x^m_{k,l,1})$ *is equal to the execution count* $(x^m_{k,l,1})$.

4.3 Cache-hit execution times

We now examine the case when $B_{k,l}$ causes a cache hit. We use $B^h_{k,l}$ to denote the sequence of machine cycles that is the concatenation of the cache-hit instruction cycles of all instructions in the l-block $B_{k,l}$. We again need to consider two cases: when $B^h_{k,l}$ ends with a B-cycle, and when $B^h_{k,l}$ ends with an E-cycle.

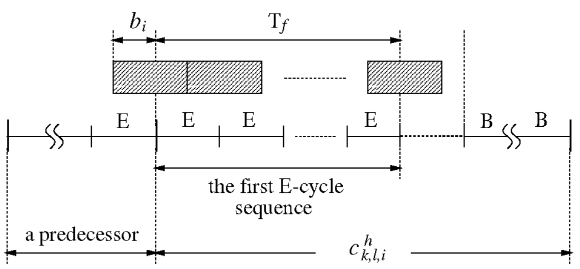
$B^h_{k,l}$ ends with a B-cycle. Again, when $B^h_{k,l}$ ends with a B-cycle, we need not be concerned with its successors. However, because $B^h_{k,l}$ begins with an E-cycle, a predecessor that ends with an E-cycle affects the delay suffered by the first E-cycle sequence in $B^h_{k,l}$. As a result, in order to determine all the possible execution times of $B^h_{k,l}$ we first need to examine its predecessors.

We use equation (2) to calculate the delay suffered by each E-cycle sequence in $B^h_{k,l}$ except the first one. Suppose that, after all the predecessors of $B^h_{k,l}$ have been analysed, there are p directed edges pointing to $B^h_{k,l}$ in the execution-time-dependency graph. We group together the edges from sources in which the elapsed times of the last DMA transfers are the same. Suppose that this procedure partitions the p edges into u disjoint groups. Let b_i denote the elapsed time of the last DMA transfer in the sources of the edges in group i, $i = 1$. Figure 3 shows the case where a predecessor of $B^h_{k,l}$ ends with an E-cycle and the elapsed time of the last DMA transfer in the predecessor is b_i. We can calculate the delay suffered by the first E-cycle sequence of $B^h_{k,l}$ using a slight modification of equation (2); we replace T in equation (2) with the expression

$$T_f + \text{BMT} + b_i, \tag{7}$$

where T_f is the execution time of the first E-cycle sequence when it executes alone. Accordingly, we can obtain an execution time of $B^h_{k,l}$ for group i. Let $c^h_{k,l,i}$ denote this execution time. We add a node in the execution-time-dependency graph to represent the execution behaviour $B^h_{k,l}$ for group i and label the node with the execution time $c^h_{k,l,i}$ and the execution count $x^h_{k,l,i}$. Because there are u such groups, we have u such nodes. We change the targets of the edges in group i to the node labelled with $c^h_{k,l,i}(x^h_{k,l,1})$. The execution counts of the edges in each group i satisfy the following constraint.

Figure 3 The execution time of $B^h_{k,l}$ for the edges in group i

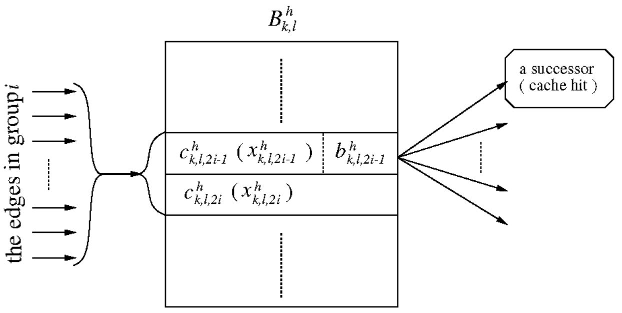


Constraint 2: *The sum of the execution counts of the directed edges entering the node labelled* $c^h_{k,l,i}(x^h_{k,l,i})$ *is equal to the execution count* $x^h_{k,l,i}$.

A predecessor that ends with a B-cycle can not affect the execution time of $B^h_{k,l}$. Hence, there is no directed edge between such a predecessor and $B^h_{k,l}$. We can use equation (2) directly to calculate the delay suffered by the first E-cycle sequence and obtain another execution time $c^h_{k,l,u+1}$ of $B^h_{k,l}$. We add a node to represent this execution behaviour of $B^h_{k,l}$ if it has a predecessor that ends with a B-cycle. In summary, $B^h_{k,l}$ may have u + 1 possible execution times when $B^h_{k,l}$ ends with a B-cycle. There are u + 1 nodes labelled by these execution times and the corresponding execution counts. The sum of these u + 1 execution counts is equal to the cache-hit count of $B^h_{k,l}$, as described in equation (3).

$B^h_{k,l}$ ends with an E-cycle. $B^h_{k,l}$ may end with an E-cycle. In this case, we have to be concerned with its successors as well as its predecessors. We use equation (2) to calculate the delay suffered by each E-cycle sequence except the first one and the last one. As stated above, we must analyse the predecessors of $B^h_{k,l}$ first in order to calculate all the possible values for the delay suffered by the first E-cycle sequence in $B^h_{k,l}$. Again, we consider first the case of predecessors of $B^h_{k,l}$ that end with an E-cycle. For this case, let u denote the number of possible values for the delay suffered by the first E-cycle sequence in $B^h_{k,l}$. In addition, the delay suffered by the last E-cycle sequence in $B^h_{k,l}$ has two possible values; one when a successor causes a cache hit and one when a successor causes a cache miss. Thus, $B^h_{k,l}$ has $2u$ possible execution times in this case. We add two nodes to represent these two execution behaviours of $B^h_{k,l}$ for each group i of incoming edges and label them $c^h_{k,l,2i-1}(x^h_{k,l,2i-1})$ and $c^h_{k,l,2i}(x^h_{k,l,2i})$, as shown in Figure 4. We have $2u$ such nodes. In addition, we replace each incoming edge in group i with two edges which have the same source as the original edge but have as targets the nodes labelled $c^h_{k,l,2i-1}(x^h_{k,l,2i-1})$ and $c^h_{k,l,2i}(x^h_{k,l,2i})$. Both the execution counts of the edges entering the node labelled $c^h_{k,l,2i-1}(x^h_{k,l,2i-1})$ and the execution counts of the edges entering the node labelled $c^h_{k,l,2i}(x^h_{k,l,2i})$ must satisfy a linear constraint similar to Constraint 2.

Figure 4 The execution times of $B^h_{k,l}$ if it ends with an E-cycle


$B^h_{k,l}$
the edges in group$i$
$c^h_{k,l,2i-1}$ ($x^h_{k,l,2i-1}$)
$b^h_{k,l,2i-1}$
$c^h_{k,l,2i}$ ($x^h_{k,l,2i}$)
a successor (cache hit)

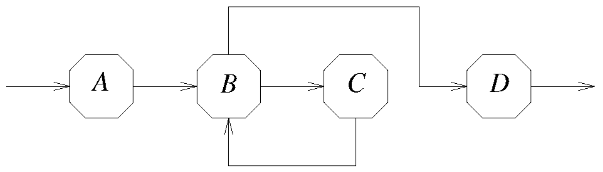
In the case when a predecessor ends with a B-cycle, we can use equation (2) directly to calculate the delay suffered by the first E-cycle sequence *in* $B^h_{k,l}$. Because of this case, $B^h_{k,l}$ has two more possible execution times. Let $c^h_{k,l,2u+1}$ and $c^h_{k,l,2u+2}$ denote these two execution times. We construct two more nodes to represent these two execution behaviours of the l-block. Including the $2u$ nodes described above, $B^h_{k,l}$ has $2u$ + 2 nodes and thus $2u$ + 2 possible execution times.

Finally, we add a directed edge from each node labelled with the execution time $c^h_{k,l,2i-1}, i = 1,\ldots,u+1$, to the node representing the cache-hit case of each successor and label this edge with a corresponding execution count. The execution counts of these edges must satisfy a linear constraint similar to Constraint 1.

4.4 An Iterative Implicit Path Enumeration Technique (I-IPET) implementation

As shown in Section IV-C, to determine the set of possible execution times of an l-block $B^h_{k,l}$, we must first determine the set of possible execution times of all its predecessors that end with an E-cycle. This requirement may lead to a cycle of dependencies. An example is the loop in the control-flow graph shown in Figure 5. Each node in this graph represents an l-block and each edge represents a control flow edge. Suppose 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 G, the execution time of C depends on B. In this section we describe an I-IPET implementation to determine the set of possible execution times of each l-block. Because there is only a finite set for the elapsed times of the last DMA transfer in an l-block, the iterative analysis will eventually terminate.

Figure 5 A simple loop



We now describe in detail the algorithm to generate the expression of the cost function (equation (6)) and the set of DMA-bounding linear constraints. Let K denote the number of the possible values for the elapsed time of the last DMA transfer in an l-block. Let J denote the maximum number of predecessors of each l-block. The complexity of this algorithm is $O(KJN)$ for a program with N l-blocks.

- *The* `main` *procedure.* Figure 6 shows the main procedure. This procedure requires as input the l-block control structure of the program to be analysed and the instructions in each l-block. Let the first l-block in the program be denoted. The procedure creates an execution-time-dependency graph G = (V, D); the nodes and edges in this graph were described earlier. It uses a variable L to hold the list of l-blocks waiting to be analysed. Moreover, 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 examined; 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 analysed.

Figure 6 The `main` procedure

```
Input:  the l-block structure of a program, and
        $B_{1,1}$, the first l-block to be executed.
Output: the cost function (6) and the set of
        DMA-bounding 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 a set of linear constraints.
7  Construct a cost function by summing each l-block's
   total execution time expression.
```

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 2–4), the l-block (called $B^h_{k,l}$) at the head of L is dequeued and its execution time is analysed by the procedure `analyze` (described below). The `analyze` procedure checks each successor of the l-block $B^h_{k,l}$. 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 analysed. We next construct a set of linear constraints for each l-block (lines 5 and 6). These linear constraints are ones for bounding the cache-hit and cache-miss counts (equations (3) and (4)) and ones for bounding the execution counts of the directed edges entering and leaving each node (Constraints 1 and 2). We also construct the total execution time expression for each l-block (equation (5)). Finally, the cost function for the program is constructed (line 7). The cost function and linear constraints thus obtained, in addition to path-bounding and cache-bounding linear constraints, are inputs to an ILP program which computes the maximum of the cost function for all values of execution counts that satisfy the constraints.

- *The* `analyze` *procedure.* The analyse procedure adds to the graph G nodes that represent (some of) $B_{k,l}$*'s* possible execution behaviours and directed edges that represent the dependencies of the execution times of its predecessors and successors on the l-block, in the manner described earlier.

Figure 7 gives the pseudo code of the `analyze` procedure. If this is the first time that $B_{k,l}$ is visited, we add two sets of nodes to the node set V*:* a set of nodes representing the execution behaviours of $B^m_{k,l}$ and a set of nodes representing the execution behaviours of $B^h_{k,l}$ if any predecessor ends with a B-cycle. We label each node with the corresponding execution time and its execution count. (Lines 1–5)

Figure 7 The `analyze` procedure

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

Procedure:
1   If ($B_{k,l}$ has not been visited)
2       – mark $B_{k,l}$ as visited;
3       – determine the execution behaviors of $B^m_{k,l}$;
4       – determine the execution behaviors of $B^h_{k,l}$
          if any predecessor ends with a B-cycle;
5       – update the node set $V$ accordingly.
6   Process each edge $e$ in $B_{k,l}$'s unprocessed list
    in the following manner:
7       – remove $e$ from the unprocessed list;
8       – determine the set of execution behaviors of
          $B^h_{k,l}$ for the edge $e$;
9       – check if there exists a set of nodes in $V$
          representing the execution behaviors;
10      – if (such a set of nodes cannot be found)
          update $V$ accordingly;
11      – if ($B_{k,l}$ ends with an E-cycle) replace $e$
          with a pair of new edges in $D$;
12      – update the target (targets) of the edge
          (the pair of new edges).
13  Update the edge set $D$ appropriately.
14  Update the list $L$ appropriately.
```

If there are edges in the unprocessed list of $B_{k,l}$, we process each edge in the list in turn and remove it from the list (lines 6–12). If $B_{k,l}$ ends with an E-cycle, we replace each edge with a pair of new edges which has the same source as the original edge. We then process each edge to achieve two goals: first, to put the edge in one of the groups of edges so that the elapsed time of the last DMA transfer in the same group are the same and second, to change the target of the edge to the node labelled with the corresponding execution time and execution count of $B_{k,l}$.

For each node added to *V*, if the last DMA transfer in the node affects the execution time of a successor, we add to *D* a directed edge from the node to the cache-hit case of the successor. We then add this edge to the unprocessed list of the successor (Line 13).

Finally, we add to the list *L* any of the successor *l*-blocks of $B_{k,l}$ which have not yet been visited or to which we just constructed an edge (Line 14).

5 Instruction caching, pipelining and DMA I/O

We now enable instruction pipelining in the machine model which allows multiple instructions to be overlapped in execution. We represent the execution of an *l*-block when it executes alone 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 (Kogge, 1981). We classify all pipeline stages into two categories: *B*-stages and *E*-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.

Figure 8 shows an *l*-block and its cache-hit and cache-miss reservation tables for example. The instruction pipeline consists of four 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 are written to the memory during the Write Back (*WB*) stage. The cache-miss reservation table shown in Figure 8(c) begins with several *B*-stages to fetch the first and subsequent instructions in the *l*-block from the main memory. Each of the subsequent instructions begins with an *E*-stage to fetch 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.

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 analyse the bus contention between the CPU and the DMAC during the cache-miss execution of the *l*-block shown in Figure 8(c), we represent the pipelined execution of the *l*-block by a sequence of *B*-cycles and *E*-cycles, as shown in Figure 9. Let m be the number of units of data the DMAC transfers between B_4 cycle and B_8 cycle. Let *R* be the length of time the pipelined execution stalls. The DMAC operates as described in Section III. Similarly, we can use equation (2) to calculate *m* and *R*.

Figure 8 An *l*-block and its cache-hit and cache-miss reservation tables: (a) an *l*-block; (b) its cache-hit reservation table and (c) its cache-miss reservation table

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

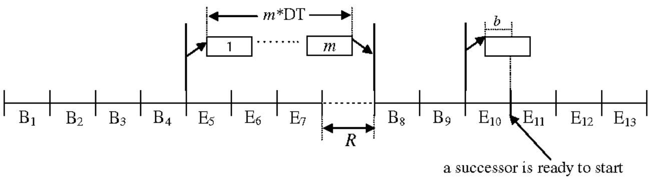
	1	2	3	4	5	6	7	8	9	10
IF	E	E	E	E	E		E			
ID		E	E	E	E		E	E		
EX			E	E			E	E	E	E
WB					B	B				

(a)

	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				

(b)

Figure 9 The concurrent execution of the DMAC and a sequence of processor cycles

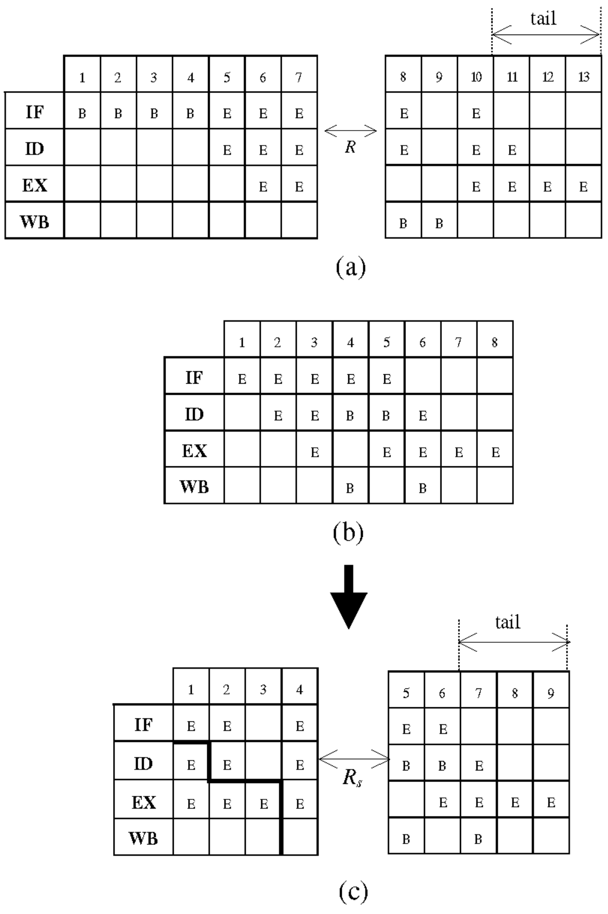


5.1 The execution time of an l-block

Figure 10 illustrates how the cache-miss execution of the *l*-block shown in Figure 8(c) affects the execution time of a successor when DMA I/O is present. Figure 10(a), a simplified version of Figure 9, shows the pipelined execution of the *l*-block when it executes concurrently with DMA I/O. Here 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. For example, the tail of the cache-miss reservation table shown in Figure 10(a) consists of Columns 11–13.

Figure 10(b) 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 10(c). 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 time the pipelined execution stalls, 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 elapsed time of the last DMA transfer of the *l*-block, as shown in Figure 9. We can calculate R_s using a slight modification of equation (7). In contrast, if there is no such DMA transfer, we use equation (2) directly to calculate R_s.

Figure 10 The interference of DMA I/O on the pipelined execution of a successor: (a) an *l*-block; (b) a successor when it causes a cache hit tail and (c) the pipelined execution of the successor



Finally, we define the execution time of an *l*-block to be 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 to 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). In the example shown in Figure 10, the execution time of the *l*-block is ($R + 10 \times T_c$) and the execution time of the successor is ($R_s + 6 \times T_c$), if it has any successor, or ($R_s + 9 \times T_c$), otherwise.

5.2 Iterative timing analysis

We represent the dependency of the execution times of an *l*-block $B_{k,l}$ on the behaviour of its predecessors and successors by an execution-time-dependency graph similar to the one described in Section 4.2. Each node under $B_{k,l}$ now represents a reservation table with stalls due to DMA I/O that leads to a possible execution time of this *l*-block and the node is labelled with the corresponding execution time and execution count. Each directed edge between two nodes represents the fact that the execution time of the *l*-block represented by the target node is affected by the elapsed time of the last DMA transfer of the *l*-block represented by the source node.

To bound the WCET of a program executing concurrently with DMA I/O on an advanced architecture with instruction caching and pipelining, we apply the I-IPET approach described in Section 4.4 to determine the set of possible execution times of each *l*-block, construct a set of DMA-bounding linear constraints on the execution counts and generate a new cost function. The maximum value of the new cost function under the path-bounding, cache-bounding and DMA-bounding linear constraints bounds the WCET of the program.

6 Experimental results

We conducted extensive experiments to demonstrate the efficacy of our I-IPET approach on bounding the WCETs of programs. We evaluated the performance of our method by comparing our WCET predictions with the traditional pessimistic WCET predictions for several sample programs. In the following we first describe the experiments. We next describe the experimental results on each of the two architectures discussed in this paper.

6.1 The experiments

Figure 11 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 path. We identified the worst-case data set of each sample program by a careful study of the program. Column 2 of Table 1 lists the number of instructions in the worst-case execution path of each program. From the tested program, *Mtx2* is obtained by unrolling the innermost loop of *Mtxm*. We used the MC68030 in this experiment because it is a widely-used embedded microprocessor for which instruction timing information is available.

Figure 11 The control flow of the experiment

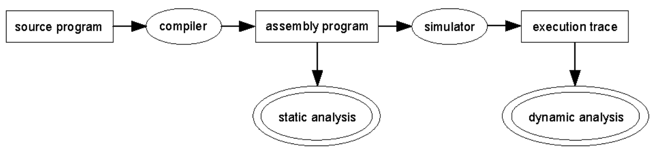


Table 1 The tested set of programs

Name	*Worst-case path insts.*	*Description*
Sels	11,713	Selection sort
Gaus	47,272	Gaussian elimination
Mtxm	40,789	Matrix multiplication
Tdsm	8,450	DMA I/O simulator
Mtx2	10,592	Loop-unrolled *Mtxm*

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 structured *program*. In the dynamic analysis part, we compared our WCET prediction with the pessimistic WCET prediction of the worst-case execution *path*.

Static analysis. Given an assembly program, we first computed its WCET when it executes without any interference of DMA I/O. Here we use the cache-aware IPET solution (Li and Malik, 1999) to obtain this prediction. We denote this value by $\mathcal{A}_s$. We next used our I-IPET approach to compute the WCET of the program when it executes concurrently with DMA I/O and denote this value by $\mathcal{W}_s$. In addition, we computed the maximum units of data which the DMAC can transfer during the execution of $\mathcal{W}_s$. 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.,

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

Dynamic analysis. The approach we took to demonstrate the improvement of our method on each program's worst-case execution path 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 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 that the DMAC transfers. We denote them by $\mathcal{W}_d$ and $\mathcal{M}_d$, respectively. 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 $\mathcal{W}_d$. 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.,

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

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

6.2 Results on an instruction-cache architecture

For each program in the test set, we first used equation (1) to compute the WCET of the program when it executes alone. We next used the I-IPET approach described in Section 4 to compute the WCET when the program executes concurrently with DMA I/O. Table 2 shows the static-analysis experimental results when the on-chip instruction cache is organised as 16 16-byte lines. Columns 2 and 3 give the values of $\mathcal{W}_s$ and $\mathcal{W}_s^a$, respectively, after each is normalised to $\mathcal{A}_s$. Column 4 gives the value of P_s for each program. For example, according to our predictions, DMA I/O extends the WCET of the program *Sels* for up to 15%. Assuming that the DMA I/O operation is carried out alone, the pessimistic method estimates the delay caused by DMA I/O to be 104% on the same program, which results in a 44% reduction by our method. Among the tested programs, our method produces up to 47% reduction from the pessimistic WCET prediction on the *Mtxm* program.

Table 2 The 16-line static-analysis results on the instruction-cache architecture

Name	$\mathcal{W}_s/\mathcal{A}_s$	$\mathcal{W}_s^a/\mathcal{A}_s$	P_s (%)
Sels	1.15	2.04	44
Gaus	1.18	2.00	41
Mtxm	1.02	1.92	47
Tdsm	1.04	1.93	46
Mtx2	1.03	1.85	44

In order to study the relationship between the reduction percentage P_s and the size of the instruction cache, we conducted the same experiment on processor configurations with instruction caches of 4, 8 and 32 16-byte cache lines. Figure 12 gives the result P_s recorded in each of these experiments. A program has a higher cache-hit ratio when it executes on a processor configuration with more cache lines. The higher cache-hit ratio increases the percentage of *E*-cycles in the execution of the program and thus allows more concurrent DMA I/O transfers. Consequently, when a program executes on a processor configuration with more cache lines, it has a larger value of P_s.

Figure 12 The static-analysis results on the instruction-cache architecture

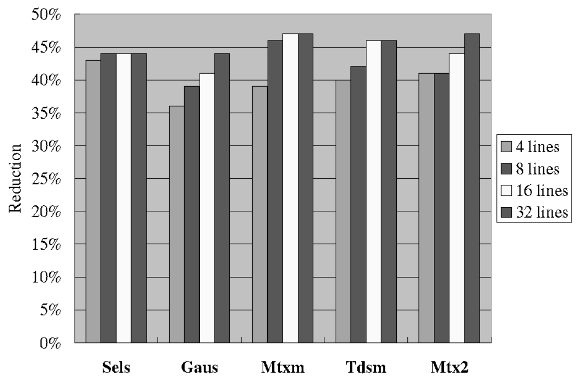
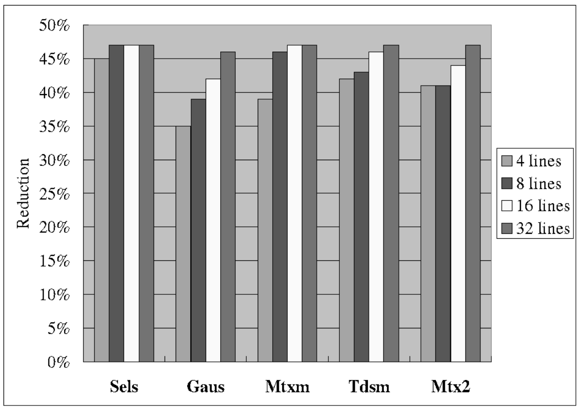


Table 3 shows the dynamic-analysis experimental results on a processor configuration with 16 16-byte cache lines. Column 2 gives the cache-hit ratio of each trace. Column 3 gives the bus utilisation of each trace when it executes alone. The bus utilisation of a trace is the amount of time the CPU uses the system bus to the execution time of the trace. In general, a trace with a higher cache-hit ratio has a larger percentage of *E*-cycles. Thus, it will have a lower bus utilisation and allow more concurrent DMA I/O transfers. Columns 4 and 5 give the values of $\mathcal{W}_d$ and $\mathcal{W}_d^a$, respectively, after each is normalised to $\mathcal{A}_d$. Column 6 gives the value of P_d for each trace. Our method produces a larger reduction percentage P_d on a trace with a higher cache-hit ratio and a lower bus utilisation. Specifically, our method produces a 47% reduction from the pessimistic WCET prediction on the traces of *Sels* and *Mtxm*, each with a 100% cache-hit ratio. We conducted the same experiment on processor configurations with instruction caches of 4, 8 and 32 16-byte lines. Figure 13 gives the results P_d in each experiment. A trace has a higher cache-hit ratio, a lower bus utilisation and thus a larger reduction percentage P_d when it executes on a processor configuration with more cache lines.

Table 3 The 16-line dynamic-analysis results on the instruction-cache architecture

Name	*Cache-hit ratio*	*Bus utilisation*	$\mathcal{W}_d/\mathcal{A}_d$	$\mathcal{W}_d^a/\mathcal{A}_d$	P_d(%)	$\mathcal{W}_d/\mathcal{W}_s$
Sels	1.00	0.11	1.01	1.90	47	0.45
Gaus	0.81	0.23	1.04	1.79	42	0.58
Mtxm	1.00	0.09	1.02	1.92	47	1.00
Tdsm	0.95	0.12	1.04	1.93	46	0.83
Mtx2	0.90	0.17	1.03	1.85	44	1.00

Figure 13 The dynamic-analysis results on the instruction-cache architecture



Column 7 of Table 3 gives the value of $\mathcal{W}_d/\mathcal{W}_s$, which is smaller than or equal to one for each of the five tested programs. In fact, throughout the whole experiment on the instruction-cache architecture, $\mathcal{W}_d/\mathcal{W}_s$ is less than or equal to one for any of the tested programs for any processor configuration. This fact shows that our WCET prediction $\mathcal{W}_s$ of any tested program safely bounds the execution time $\mathcal{W}_d$ of the worst-case execution path of the program when DMA I/O is present. 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 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., $\mathcal{W}_d/\mathcal{W}_s = 1$, at any processor configuration. 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.

6.3 *Results on an advanced architecture*

In this experiment both the instruction cache and the instruction pipeline on MC68030 were enabled. We used the method described in Section 5 to compute the WCET of a program when it executes concurrently with DMA I/O. 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 4 shows the static-analysis results with 16 16-byte cache lines, and Figure 14 shows the reduction percentage P_s on processor configurations with 4, 8, 16 and 32 cache lines. As explained earlier, a program has a higher hit ratio and therefore, has a larger value of P_s when it executes on a processor configuration that has more cache lines. Because pipelined execution decreases the fraction of the execution time of a program which can be overlapped with DMA I/O transfers, our method produces a smaller P_s on an advanced architecture than on an instruction-cache architecture. However, the reduction P_s observed in this experiment is only slightly smaller. This is because there are only two stages in the instruction pipeline on MC68030. We expect that our method will produce a much smaller P_s on an advanced architecture that has more stages in the instruction pipeline.

Table 4 The 16-line static-analysis results on the advanced architecture

Name	$\mathcal{W}_s/\mathcal{A}_s$	$\mathcal{W}_s^a/\mathcal{A}_s$	P_s (%)
Sels	1.06	1.88	44
Gaus	1.08	1.77	39
Mtxm	1.03	1.94	47
Tdsm	1.04	1.93	46
Mtx2	1.02	1.81	44

Figure 14 The static-analysis results on the advanced architecture

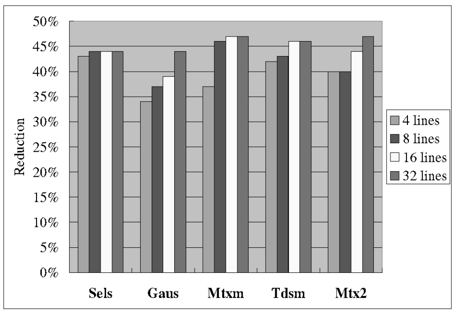
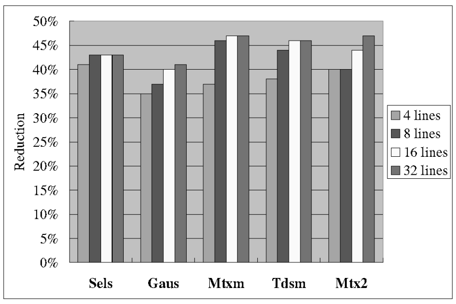


Table 5 shows the dynamic-analysis experimental results on a processor configuration with 16 16-byte cache lines. Figure 15 shows the reduction percentage P_d for all processor configurations. The cache-hit ratio of each trace is equal to the one we obtained on an instruction-cache architecture. Again, our method produces a larger reduction percentage P_d for a trace with more cache lines. Because of pipelined execution, each trace has a larger bus utilisation. Therefore, each trace has a smaller reduction percentage P_d. Finally, that fact that $\mathcal{W}_d/\mathcal{W}_s \leq 1$ in all our experiments demonstrates that our method safely and tightly bounds the WCET.

Table 5 The 16-line dynamic-analysis results on the advanced architecture

Name	*Cache-hit ratio*	*Bus utilisation*	$\mathcal{W}_d/\mathcal{A}_d$	$\mathcal{W}_d^a/\mathcal{A}_d$	P_d (%)	$\mathcal{W}_d/\mathcal{W}_s$
Sels	1.00	0.17	1.06	1.87	43	0.52
Gaus	0.81	0.26	1.05	1.75	40	0.57
Mtxm	1.00	0.10	1.03	1.94	47	1.00
Tdsm	0.95	0.14	1.03	1.91	46	0.78
Mtx2	0.90	0.20	1.02	1.82	44	1.00

Figure 15 The dynamic-analysis results on the advanced architecture



7 Concluding remarks

A hard-real-time embedded system is required to process tasks with timing requirements that must be met to ensure the correctness of the system. The analysis which determines whether a particular system can meet its timing requirements relies on prior information on the WCET of each task. The problem of determining the WCET of a program has received a lot of attention recently. However, all of the previous research on bounding the WCET assumes that the program being analysed executes without any interference of I/O activities. Our work is the first one that considers the interference of concurrently executing cycle-stealing DMA I/O in bounding the WCET of a program.

In this paper we presented an iterative I-IPET approach for bounding the WCET of a program executing concurrently with cycle-stealing DMA I/O. Our I-IPET approach follows the control flow of the program iteratively to determine the set of possible execution times of each instruction and construct a set of linear constraints on their execution counts. The maximum value of the cost function under the set of linear constraints bounds the WCET of the program. We conducted extensive experiments on a widely-used microprocessor to demonstrate the efficacy of our approach. The experimental results show that our method safely and tightly bounds the WCET of a program. As we do not impose any architecture-specific restriction in our I-IPET approach, we believe our method can be easily adapted to accurately bound the WCET of a program executed on other modern architectures.

The number of linear constraints generated by our I-IPET approach depends heavily on the complexity of the program being analysed as well as the dynamic architectural features.

Due to the simplicity of the tested programs and the architecture used in the experiments, the iterative procedure shown in Section 4.4 generates <100 constraints for each of the tested programs given in Table 1 and a standard ILP solver can calculate the WCET in less than a couple of seconds. We expect the number of constraints to increase significantly when the complexity of programs and the machine model increases. Obtaining a WCET estimation in an efficient way at the input of a large number of constraints requires an optimised ILP solver, which is beyond the scope of this paper.

Our I-IPET approach is an extension of the cache-aware IPET solution developed by Li and Malik (1999). Our I-IPET approach significantly improves the analysis capability of the IPET methodology in two directions: to include the timing analysis of cycle-stealing DMA I/O and to model the performance of modern hardware architectures such as instruction pipelining. In addition, our DMA I/O analysis can be easily incorporated with other IPET extensions developed later to produce tighter WCET predictions. For example, we can integrate with the extension developed by Li and Malik (1999) to consider the interference of data caching, the extension developed by Engblom and Ermedahl (2000) to analyse the control flow information, the extension by Theiling (2002) to take into account the effect of different invocations to the same function and the extension by Theiling et al. (2000) to reduce the complexity of the ILP problems by using abstract interpretation. In summary, our work advances the IPET methodology significantly and encourages the inclusion of I/O activities and modern hardware architectures in hard-real-time embedded systems.

Acknowledgements

The authors would like to thank Jane Liu and David Hull for their comments on this paper. We also would like to thank Steve Li and Sharad Malik for providing their timing tool, with which we obtained the pessimistic WCET predictions. This research was supported in part by National Science Council, ROC, under Grant NSC 92-2213-E-007-015 and by the Ministry of Education, ROC, under Grant MOE 89-E-FA04-1-4.

References

Black, J., McKenna, C. and Kaplinsky, C. (1981) *The VMEbus Specification*, Motorola.

Colin, A. and Puaut, I. (2000) 'Worst case execution time analysis for a processor with branch prediction', *Journal of Real-Time Systems*, Vol. 18, Nos. 2-3, May, pp.249–274.

Engblom, J. and Ermedahl, A. (1999) 'Pipeline timing analysis using a tracedriven simulator', *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, December, Hong Kong, China, pp.88–95.

Engblom, J. and Ermedahl, A. (2000) 'Modeling complex flows for worst-case execution time analysis', *Proceedings of the 21st Real-Time System Symposium*, November, Orlando, Florida, USA, pp.163–174.

Engblom, J., Ermedahl, A., Sjoedin, M., Gubstafsson, J. and Hansson, H. (2001) 'Worst-case execution-time analysis for embedded real-time systems', *Journal of Software Tools for Technology Transfer*, Vol. 4, No. 4, February, pp.437–455.

Ferdinand, C. and Wilhelm, R. (1999) 'Efficient and precise cache behavior prediction for real-time systems', *Journal of Real-Time Systems*, Vol. 17, Nos. 2–3, pp.131–181.

Ferdinand, C., Martin, F. and Wilhelm, R. (1997) 'Applying compiler techniques to cache behavior prediction', *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pp.37–46.

Gupta, R. and Gopinath, P. (1994) 'Correlation analysis techniques for refining execution time estimates of real-time applications', *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May, pp.54–58.

Healy, C., Arnold, R., Mueller, F., Whalley, D. and Harmon, M. (1999) 'Bounding pipeline and instruction cache performance', *IEEE Transactions on Computers*, January, Vol. 48, No. 1, pp.53–70.

Healy, C.A., Sjodin, M., Rustagi, V., Whalley, D.B. and van Engelen, R. (2000) 'Supporting timing analysis by automatic bounding of loop iterations', *Journal of Real-Time Systems*, Vol. 18, Nos. 2–3, pp.129–156.

Huang, T-Y. and Liu, J.W-S. (1995) 'Predicting the worst-case execution time of the concurrent execution of instructions and cycle-stealing DMA I/O operations', *ACM SIGPLAN Notices*, Vol. 30, No. 11, November, pp.1–6.

Huang, T-Y., Liu, J.W-S. and Hull, D. (1996) 'A method for bounding the effect of DMA I/O interference on program execution time', *Proceedings of the 17th Real-Time System Symposium*, December, Washington, DC, USA, pp.275–285.

Kim, S-K., Ha, R. and Min, S.L. (1999) 'Analysis of the impacts of overestimation sources on the accuracy of worst case timing analysis', *Proceedings of the 20th Real-Time System Symposium*, December, Phoenix, AZ, USA, pp.22–31.

Kogge, P.M. (1981) *The Architecture of Pipelined Computers*, Hemisphere Publishing Corp., New York, NY.

Li, Y-T.S. and Malik, S. (1995) 'Performance analysis of embedded software using implicit path enumeration', *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, June, pp.456–561.

Li, Y-T.S. and Malik, S. (1999) *Performance Analysis of Real-Time Embedded Software*, Kluwer Academic Publishers, Princeton, NJ, USA.

Lim, S-S., Han, J.H., Kim, J. and Min, S.L. (1998) 'A worst case timing analysis technique for multiple-issue machines', *Proceedings of the 19th Real-Time System Symposium*, December, Washington, DC, USA, pp.334–345.

Liu, C.L. and Layland, J. (1973) 'Scheduling algorithms for multiprogramming in a hard real-time environment', *Journal of the ACM*, Vol. 10, No. 1, pp.46–61.

Lundqvist, T. and Stenström, P. (1999) 'An integrated path and timing analysis method based on cycle-level symbolic execution', *Journal of Real-Time Systems*, Vol. 17, Nos. 2–3, pp.183–207.

Lundqvist, T. and Stenström, P. (1999) 'Timing anomalies in dynamically scheduled microprocessors', *Proceedings of the 20th Real-Time System Symposium*, December, pp.12–21.

Mueller, F., Whalley, D. and Harmon, M. (1994) 'Predicting instruction cache behavior', *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, June, pp.23–40.

Ottosson, G. and Sjödin, M. (1997) 'Worst-case execution time analysis for modern hardware architectures', *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, June, pp.47–55.

Park, C-Y. and Shaw, A.C. (1991) 'Experiments with a program timing tool based on source-level timing schema', *IEEE Computer*, Vol. 24, No. 5, May, pp.48–57.

Puschner, P. and Koza, C. (1989) 'Calculating the maximum execution time of real-time programs', *Journal of Real-Time Systems*, Vol. 1, pp.159–176.

Sha, L., Rajkumar, R. and Lehoczky, J.P. (1990) 'Priority inheritance protocols: an approach to real-time synchronisation', *IEEE Transactions on Computers*, Vol. 39, No. 9, pp.1175–1185.

Shaw, A.C. (1989) 'Reasoning about time in higher-level language software', *IEEE Transactions on Software Engineering*, July, Vol. 15, No. 7, pp.875–889.

Stappert, F. and Altenbernd, P. (2000) 'Complete worst-case execution time analysis of straight-line hard real-time programs', *Journal of Systems Architecture*, Vol. 46, No. 4, pp.339–355.

Sun, J., Gardner, M. and Liu, J.W-S. (1997) 'Bounding completion times of jobs with arbitrary release times, variable execution times, and resource sharing', *IEEE Transactions on Software Engineering*, October, Vol. 23, No. 10, pp.603–615.

Theiling, H. (2002) 'ILP-based interprocedural path analysis', *Proceedings of the Second International Workshop on Embedded Software*, December, Grenoble, France, pp.349–363.

Theiling, H., Ferdinand, C. and Wilhelm, R. (2000) 'Fast and precise WCET prediction by separated cache and path analyses', *Journal of Real-Time Systems*, Vol. 18, Nos. 2–3, pp.157–179.