Timing Analysis of Embedded Software for Speculative Processors

Tulika Mitra
tulika@comp.nus.edu.sg

Abhik Roychoudhury
abhik@comp.nus.edu.sg

Xianfeng Li
lixianfe@comp.nus.edu.sg

School of Computing
National University of Singapore
Republic of Singapore 117543

ABSTRACT

Static timing analysis of embedded software is important for systems with hard real-time constraints. To accurately estimate time bounds, it is essential to model the underlying micro-architecture. In this paper, we study static timing analysis of embedded programs for modern processors with speculative execution. Speculation of conditional branch outcomes significantly improves processor performance, and hence program execution time. Although speculation is used in most modern processors, its effect on software timing has not been systematically studied before. The main contribution of our work is a parameterized framework to model different control flow speculation schemes. The accuracy of our framework is illustrated through tight timing estimates obtained for benchmark programs.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and Embedded Systems*

General Terms

Measurement, Performance.

Keywords

Branch Prediction, Worst Case Execution Time.

1. INTRODUCTION

An embedded system contains processor(s) running specific application programs which communicate with an external environment in a timely fashion. These application programs thus have real-time requirements, i.e., there are hard deadlines on the execution time of such software. Moreover, many embedded systems are safety critical. Therefore, it is important to perform static analysis of embedded software to guarantee the satisfiability of all timing constraints.

Static timing analysis can provide an upper/lower bound on the execution time of a program. These bounds are useful for schedulability analysis, hardware/software partitioning, choice of processor (design space exploration) etc. Due to its inherent importance in embedded system design, timing analysis of embedded software has been extensively studied [2, 5, 6, 9, 11, 14, 16]. Accurate timing analysis *critically* depends on modeling the effects of the underlying micro-architecture. Ignoring the micro-architecture can produce extremely pessimistic time bounds. This is particularly so because modern processors employ advanced micro-architectural features such as pipeline, caches, and speculative execution to speed up program execution. In the recent past, researchers have studied the effects of pipeline and cache on program execution time [5, 9, 11, 15].

The presence of branch instructions forms control dependency between different parts of the program. This dependency causes pipeline stalls which can be avoided by speculating the control flow subsequent to a branch. Current generation processors perform control flow speculation through branch prediction, which predicts the outcome of branch instructions [7]. If the prediction is correct, then execution proceeds without any interruption. For incorrect prediction, the speculatively executed instructions are undone, incurring a branch misprediction penalty. This penalty varies between 3-19 clock cycles. If branch prediction is not modeled, all the branches in the program must be conservatively assumed to be mispredicted for finding the maximum execution time. *This pessimism results in as much as* $60-70\%$ *over-estimation for some of the benchmarks in this paper*, even assuming a 3 clock cycle branch misprediction penalty.

In this paper, we model the effects of speculation via branch prediction on the *Worst Case Execution Time* of a program, also known as its WCET. Our micro-architectural modeling is completely generic and parameterizable w.r.t. the currently used branch prediction schemes. It automatically derives linear constraints on the total misprediction count from the control flow graph of the program. These constraints can be solved by integer linear programming (ILP) solver to compute bounds on program execution time.

2. MODELING BRANCH PREDICTION

Branch prediction can be *static* or *dynamic*. Static schemes associate a fixed prediction to each branch instruction via

ISSS'02, October 2–4, 2002, Kyoto, Japan.

compile time analysis. Almost all modern processors, however, predict the branch outcome dynamically based on past execution history [7]. Dynamic schemes are more accurate than static schemes, and in this work we study only dynamic branch prediction.

Dynamic schemes predict a branch depending on the execution history. The first dynamic technique proposed is called *local branch prediction* [7, 12], where each branch is predicted only based on its own last few outcomes. This scheme uses a 2^n-entry branch prediction table to store the past branch outcomes, which is indexed by the n lower order bits of the branch address. In the simplest case, each prediction table entry is 1-bit and stores the last outcome of the branch mapped to that entry. When a branch is encountered, the corresponding table entry is looked up and used as the prediction. When a branch is resolved, the corresponding table entry is updated with the outcome.

Most modern processors however use *global branch prediction* schemes [18] (also called correlation based schemes), which are more accurate. Examples of processors using global branch prediction include Intel Pentium Pro, AMD, Alpha as well as embedded processors PowerPC 440GP [13] and SB-1 MIPS 64 [8]. In these schemes, the prediction of the outcome of a branch I not only depends on I's recent outcomes, but also on the outcomes of the other recently executed branches. Global schemes can exploit the fact that behavior of neighboring branches in a program are often correlated. Global schemes uses a single shift register, called *Branch History Register (BHR)* to record the outcomes of n most recent branches. As in local schemes, there is a global *branch prediction table* in which the predictions are stored. The various global schemes differ from each other (and from local schemes) in the way the prediction table is looked up when a branch is encountered.

We now present our timing estimation technique to model effects of speculation. In particular, we consider *GAg*, a global branch prediction scheme [12, 18], which uses the BHR as an index to look up the prediction table. However, our modeling is generic and not restricted to *GAg*. In Section 4, we will demonstrate how it easily captures other global prediction schemes as well as local schemes.

Control flow graph.

The starting point of our analysis is the control flow graph (CFG) of the program. The vertices of this graph are basic blocks, and an edge $i \rightarrow j$ denotes flow of control from basic block i to basic block j. We assume that the control flow graph has a unique *start* node and a unique *end* node, such that all program paths originate at the start node, and terminate at the end node. Each edge $i \rightarrow j$ of the control flow graph has a label, denoted $label(i \rightarrow j)$. For any block i, if the last instruction of i is a branch then it has two outgoing edges labeled 0 and 1. Otherwise, block i has one outgoing edge with label U.

For programs with procedures and functions (recursive or otherwise), we create a separate copy of the CFG of a procedure P for every distinct call site of P in the program. Each call of P transfers control to its corresponding copy.

Flow constraints and loop bounds.

Let v_i denote the number of times block i is executed, and let $e_{i,j}$ denote the number of times control flows through the edge $i \rightarrow j$. As the start and end blocks are executed exactly once,

$$v_{start} = v_{end} = 1 = \sum_{\substack{i \\ start \rightarrow i}} e_{start,i} = \sum_{\substack{i \\ i \rightarrow end}} e_{i,end}$$

As inflow equals outflow for other basic blocks,

$$v_i = \sum_{\substack{j \\ j \rightarrow i}} e_{j,i} = \sum_{\substack{j \\ i \rightarrow j}} e_{i,j}$$

We provide bounds on the maximum number of iterations for loops and maximum depth of recursive invocations for recursive procedures. These bounds can be user provided, or can be computed offline for certain programs [6].

Defining Execution Time bounds.

Let $cost_i$ be the execution time of basic block i assuming perfect branch prediction. Given the program, $cost_i$ is a fixed constant for each i. Then, the total execution time of the program is

$$Time = \sum_i (cost_i * v_i + penalty * m_i)$$

where *penalty* is a constant denoting the penalty for a single branch misprediction; m_i is the number of times the branch in block i is mispredicted. If block i does not contain a branch, then $m_i = 0$. By maximizing/minimizing this objective function we can get upper/lower bounds on execution time. We now derive constraints on v_i and m_i.

Introducing History Patterns.

To determine the prediction of a block i, we first compute the index into the prediction table. In the case of *GAg*, this index is the outcome of last k branches before block i is executed. These k outcomes are recorded in the Branch History Register (BHR). Thus, if $k = 2$ and the last two branches were taken (1) followed by not taken (0), the index would be 10. We define $e^{\pi}_{i,j}$, v^{π}_i and m^{π}_i: execution count of $i \rightarrow j$, execution count and misprediction count of block i when i is executed with BHR $= \pi$, respectively. By definition:

$$e_{i,j} = \textstyle\sum_\pi e^{\pi}_{i,j}; \quad m_i = \textstyle\sum_\pi m^{\pi}_i; \quad v_i = \textstyle\sum_\pi v^{\pi}_i; \quad m^{\pi}_i \leq v^{\pi}_i$$

For each block i and history π, we compute via static analysis of the control flow graph a predicate *poss* where $poss(i, \pi)$ is true if and only if i can be reached with history π. If $\neg poss(i, \pi)$, then we set $e^{\pi}_{i,j} = v^{\pi}_i = m^{\pi}_i = 0$.

Control flow among history patterns.

First, we define constraints on v^{π}_i. This provides an upper bound on m^{π}_i. Recall that our index into the prediction table is simply a history recording the past few branch outcomes. To model the change in history due to control flow, we use the left shift operator; thus $left(\pi, 0)$ shifts pattern π to the left by one position and puts 0 as the rightmost bit. We define:

DEFINITION 1. *Let $i \rightarrow j$ be an edge in the control flow graph and let π be the history pattern at basic block i. The change in history pattern on executing $i \rightarrow j$ is given by $\Gamma(\pi, i \rightarrow j)$ where:*

$$\Gamma(\pi, i \rightarrow j) = \begin{cases} \pi & \text{if } label(i \rightarrow j) = U \\ left(\pi, 0) & \text{if } label(i \rightarrow j) = 0 \\ left(\pi, 1) & \text{if } label(i \rightarrow j) = 1 \end{cases}$$

Now consider all inflows into block i in the control flow graph. Basic block i can execute with history π only if: block j executes with some history π', control flows along the edge $j \to i$, and $\Gamma(\pi', j \to i) = \pi$.

Note that for any incoming edge $j \to i$, there can be at most two history patterns π' such that $\Gamma(\pi', j \to i) = \pi$. For example if $label(j \to i) = 1$, then $\Gamma(011, j \to i) = \Gamma(111, j \to i) = 111$. For any block i (expect start block), from the inflows of i's execution with history π we get:

$$v_i^\pi = \sum_{\substack{j \\ j \to i}} \sum_{\substack{\pi' \\ \pi = \Gamma(\pi', j \to i)}} e_{j,i}^{\pi'}$$

Similarly, for any basic block i (except end block) from the outflows of i's execution with history π we get:

$$v_i^\pi = \sum_{\substack{j \\ i \to j}} e_{i,j}^\pi$$

Repetition of a history pattern. Suppose there is a misprediction of the branch in block i with history π. This means that certain blocks (maybe i itself) were executed with history π, the outcome of these branches appear in the πth row of the prediction table, and the outcome of these branches *must have created* a prediction different from the current outcome of block i. To model mispredictions, we need to capture repeated occurrence of a history π during program execution. For this purpose, we define $p_{i \rightsquigarrow j}^\pi$.

DEFINITION 2. *Let i be the start block of the control flow graph or a basic block with branch instruction. Let j be the end block of the control flow graph, or a basic block with a branch instruction. Let π be a history pattern. Then $p_{i \rightsquigarrow j}^\pi$ is the number of times a path is taken from i to j s.t.*

- *π never occurs at a node with branch instruction between i and j.*
- *If $i \neq$ start block, then π occurs at block i*
- *If $j \neq$ end block, then π occurs at block j*

Intuitively, $p_{i \rightsquigarrow j}^\pi$ denotes the number of times control flows from block i to block j s.t. (a) π th row of the prediction table is used for prediction at blocks i and j, and (b) the π th row of the prediction table is never used for prediction between blocks i and j. In these scenarios, the outcome of block i can affect the prediction of block j (and cause a misprediction). Furthermore, $p_{start \rightsquigarrow i}^\pi$ ($p_{i \rightsquigarrow end}^\pi$) denotes the number of times the π th row of the prediction table is looked up for the first (last) time at block i.

When the πth row of the prediction table is used at block i for branch prediction, either it is the first use of the π th row (denoted by $p_{start \rightsquigarrow i}^\pi$) or the π th row was used for branch prediction last time in some block $j \neq start$. Similarly, for every use of the π th row of the prediction table at block i, either it is the last use of the πth row (denoted by $p_{i \rightsquigarrow end}^\pi$) or it is used for branch prediction next time in block $j \neq end$. Since v_i^π denotes the number of times block i uses the πth row of prediction table, therefore:

$$v_i^\pi = \sum_j p_{j \rightsquigarrow i}^\pi = \sum_j p_{i \rightsquigarrow j}^\pi$$

Also, there can be at most one first use, and at most one last use of the π th row of the prediction table:

$$\sum_i p_{start \rightsquigarrow i}^\pi \leq 1 \quad \text{and} \quad \sum_i p_{i \rightsquigarrow end}^\pi \leq 1$$

Furthermore, if $\neg poss(i, \pi)$ or $\neg poss(j, \pi)$ or j is not reachable from i then we set: $p_{i \rightsquigarrow j}^\pi = 0$.

Introducing branch outcomes. Misprediction occurs on differing branch outcomes for the same history pattern. We define two new variables $p_{i \rightsquigarrow j}^{\pi,1}$ and $p_{i \rightsquigarrow j}^{\pi,0}$ corresponding to the two outcomes of branch at i. Let $Allpaths(p_{i \rightsquigarrow j}^\pi)$ denote the set of program paths contributing to the count $p_{i \rightsquigarrow j}^\pi$. Any such path must either begin with i's outgoing edge labeled 1 (say $i \to k$) or i's outgoing edge labeled 0 (say $i \to l$). We now define:

- $p_{i \rightsquigarrow j}^{\pi,1}$ denotes the execution count of those paths in $Allpaths(p_{i \rightsquigarrow j}^\pi)$ which begin with the edge $i \to k$
- $p_{i \rightsquigarrow j}^{\pi,0}$ denotes the execution count of those paths in $Allpaths(p_{i \rightsquigarrow j}^\pi)$ which begin with the edge $i \to l$

By definition $p_{i \rightsquigarrow j}^\pi = p_{i \rightsquigarrow j}^{\pi,1} + p_{i \rightsquigarrow j}^{\pi,0}$

$$\textstyle\sum_j p_{i \rightsquigarrow j}^{\pi,1} = e_{i,k}^\pi \quad \text{and} \quad \sum_j p_{i \rightsquigarrow j}^{\pi,0} = e_{i,l}^\pi$$

Modeling mispredictions. For simplicity of exposition, let us assume that each row of the prediction table contains a one bit prediction: 0 denotes a prediction that the branch will not be taken, and 1 denotes a prediction that the branch will be taken. However, our technique for estimating the mispredictions is generic. It can be extended if the prediction table maintains ≥ 2 bits per entry.

Recall that m_i^π denotes the number of mispredictions of the branch in block i when block i is executed with history pattern π. There can be two scenarios for misprediction.

- Case 1: Branch of block i is taken
  The number of such outcomes is $\leq \sum_j p_{i \rightsquigarrow j}^{\pi,1}$, since this denotes the total outflow from block i when it is executed with history π and the branch at i is taken. Since branch at i was mispredicted, the prediction in row π of the prediction table must have been 0 (not taken). This is possible only if: another block j was executed with history π, branch of block j was not taken, and history π never appeared between blocks j and i. The total number of such inflows into block i is at most $\sum_j p_{j \rightsquigarrow i}^{\pi,0}$.
- Case 2: Branch of block i is not taken
  Number of such outcomes is $\leq \sum_j p_{i \rightsquigarrow j}^{\pi,0}$. Total number of inflows into block i s.t. the branch i can be misprediction with history pattern π is at most $\sum_j p_{j \rightsquigarrow i}^{\pi,1}$.

From the above, we derive the following bound on m_i^π

$$m_i^\pi \leq min(\sum_j p_{i \rightsquigarrow j}^{\pi,1}, \sum_j p_{j \rightsquigarrow i}^{\pi,0}) + min(\sum_j p_{i \rightsquigarrow j}^{\pi,0}, \sum_j p_{j \rightsquigarrow i}^{\pi,1})$$

This constraint can be straightforwardly rewritten into linear inequalities by introducing new variables. Also, we

derived the bound on m_i^π assuming that each row of the prediction table contains one bit. If each row of the prediction table contains $k > 1$ bits (in practice at most 2 bits), we then consider the outcomes at block i, and last k uses of the π th row of the prediction table before arriving at i.

Putting it all together. We have derived linear inequalities on v_i (execution count of block i) and m_i (misprediction count of block i). We now maximize the objective function subject to these constraints using an (integer) linear programming solver to give an estimate of the Worst Case Execution Time (WCET) of the program.

3. AN EXAMPLE

We illustrate our estimation technique with a simple example. Consider the CFG in Figure 1. All edges of the graph are labeled. Recall that the label U denotes unconditional control flow and the label 1 (0) denotes control flow by taking (not taking) a conditional branch. We assume that a 2 bit history pattern is maintained, *i.e.*, the prediction table has four rows for the history patterns 00, 01, 10, 11.

Flow constraints and loop bounds. The *start* and *end* nodes execute only once. Hence

$$v_{start} = v_{end} = 1 = e_{start,1} = e_{2,end} + e_{1,end}$$

From the inflows and outflows of blocks 1 and 2, we get:

$$v_1 = e_{start,1} + e_{2,1} = e_{1,2} + e_{1,end}$$

$$v_2 = e_{1,2} = e_{2,end} + e_{2,1}$$

Furthermore, the edge $2 \rightarrow 1$ is a loop, and its bound must be given. Let us consider a bound of 100. Then, $e_{2,1} < 100$.

Defining WCET. Let us assume a branch misprediction penalty of 3 clock cycles. The WCET of the program is obtained by maximizing

$$Time = 2v_{start} + 2v_1 + 4v_2 + 2v_{end} + 3m_1 + 3m_2$$

assuming $cost_{start} = cost_1 = 2$, $cost_2 = 4$, $cost_{end} = 2$. Recall that $cost_i$ is the execution time of block i (assuming perfect prediction); m_i is the number of mispredictions of block i. There are no mispredictions for executions of *start* and *end* blocks as they do not have branches.

Introducing History Patterns. We find out the possible history patterns π for each basic block i via static analysis of the CFG. This information is denoted by the predicate $poss(i, \pi)$. The initial history at the beginning of program execution is assumed to be 00, *i.e.*, $poss(start, \pi)$ is true iff $\pi = 00$. In our example, we obtain that $poss(1, \pi)$ is true iff $\pi \in \{00, 01\}$ and $poss(2, \pi)$ is true iff $\pi \in \{00, 10\}$.

We introduce the variables v_i^π and m_i^π: the execution count and misprediction count of block i with history π.

$$m_1^{00} \leq v_1^{00} \text{ and } m_1^{01} \leq v_1^{01} \quad m_2^{00} \leq v_2^{00} \text{ and } m_2^{10} \leq v_2^{10}$$
$$m_1 = m_1^{00} + m_1^{01} \qquad v_1 = v_1^{00} + v_1^{01}$$
$$m_2 = m_2^{00} + m_2^{10} \qquad v_2 = v_2^{00} + v_2^{10}$$

The variables v_{start}^π, v_{end}^π and $e_{i,j}^\pi$ are defined similarly.

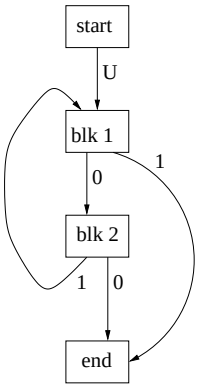

Figure 1: Example Control Flow Graph

Control flow among history patterns. We now derive the constraints on v_i^π based on flow of the pattern π. Let us consider the inflows and outflows of block 1 with history 01. From the inflow we get: $v_1^{01} = e_{2,1}^{10} + e_{2,1}^{00}$

Note that the inflow from block *start* to block 1 is automatically disregarded in this constraint since it cannot produce a history 01 when we arrive at block 1. Also, for the inflows from block 2 the history at block 2 can be either 00 or 10. Both of these patterns produce history 01 at block 1 when control flows via the edge $2 \rightarrow 1$. From the outflows of block 1 with history 01 we have: $v_1^{01} = e_{1,2}^{01} + e_{1,end}^{01}$ Constraints for other blocks and patterns are similar.

Repetition of a history pattern. To model the repetition of history pattern along a program path, the variables $p_{i \rightsquigarrow j}^\pi$ are introduced (refer Definition 2). We now present the constraints for the pattern 01. Corresponding to the first and last occurrence of the history pattern 01 we get:

$$p_{start \rightsquigarrow 1}^{01} \leq 1 \;\; and \;\; p_{1 \rightsquigarrow end}^{01} \leq 1$$

Corresponding to the repetition of the pattern 01 we get:

Exec. with01 Inflow from last 01 Outflow to next 01

$$v_1^{01} \quad = p_{1 \rightsquigarrow 1}^{01} + p_{start \rightsquigarrow 1}^{01} \quad = p_{1 \rightsquigarrow 1}^{01} + p_{1 \rightsquigarrow end}^{01}$$

Constraints for the other patterns are derived similarly.

Modeling mispredictions. Using variables v_i^π, $p_{i,j}^\pi$ and $e_{i,j}$ we get constraints for m_1^{00}, m_2^{00}, m_1^{01} and m_2^{10} (not shown due to space limitations). This bounds the total number of mispredictions $m_1 + m_2$. The objective function is maximized subject to these constraints to obtain the WCET.

4. MODELING OTHER SCHEMES

We now discuss the extensions of the technique for modeling other branch prediction schemes. The prediction schemes differ from each other primarily in how they index into the prediction table. To predict a branch I, the index computed can be a function of: (a) the past execution trace (history) and (b) address of the branch instruction I. In the *GAg* scheme, the index computed depends solely on the history and not on the branch instruction address. Other global prediction schemes (*gshare*, *gselect*) use both history and

Program	Description
`check`	-ve number search of 100-element array
`matsum`	Summation of two 100×100 matrices
`matmul`	Multiplication of two 10×10 matrices
`fft`	1024-point Fast Fourier Transform
`fdct`	Fast Discrete Cosine Transform
`isort`	Insertion sort of 100-element array
`bsearch`	Binary search of 100 element array
`eqntott`	Drawn from SPEC'92 integer benchmarks
`dhry`	Dhrystone benchmark

Table 1: Description of benchmark programs.

branch address, while local schemes use only the branch address. Our modeling is independent of the definition of the prediction table index, so far called as the history pattern π. To model the effect of other branch prediction schemes, we only alter the meaning of π, and show how π is updated with the control flow (the Γ function of Definition 1). *No change is made to the linear constraints* described before.

In the popular *gshare* [12] scheme, the BHR is XOR-ed with last n bits of the branch address to look up the prediction table. Usually, *gshare* results in a more uniform distribution of table indices compared to *GAg*. We define the index π as $\pi = history_m \oplus address_n(I)$ where m, n are constants, $n \geq m$, $\oplus$ is XOR, $address_n(I)$ denotes the lower order n bits of branch instruction I in block i, and $history_m$ denotes the most recent m branch outcomes (which are XOR-ed with higher-order m bits of $address_n(I)$). And,

$$\Gamma_{gshare}(\pi, i \rightarrow j) = \Gamma(history_m, i \rightarrow j) \oplus address_n(J)$$

In *gselect (GAp)* [18], the BHR is concatenated with the last few bits of the branch address to look up the table. The modeling is similar and is omitted for space considerations.

In local schemes, the index π for branch instruction I is the least significant n bits of I's address, denoted $address_n(I)$ (n is a constant). Here π is independent of the past execution history of other branches. The update of π due to control flow is given by $\Gamma_{local}(\pi, i \rightarrow j) = address_n(J)$, where $address_n(J)$ denotes the least significant n bits of the branch instruction J in basic block j.

5. EXPERIMENTAL RESULTS

We selected nine different benchmarks for our experiments (refer Table 1): `check`, `matsum`, `matmult`, `fft` and `fdct` are loop intensive programs; `isort`, `bsearch`, `dhry` and `eqntott` execute hard-to-predict conditional branches arising from if-then-else statements within nested loops.

Methodology. We assumed zero cache misses and a perfect processor pipeline with no stalls except for penalty due to misprediction of conditional branches. We assumed that the branch misprediction penalty is 3 clock cycles (as in the Intel Pentium processor). We used the SimpleScalar architectural simulation platform [1] in the experiments. SimpleScalar instruction set architecture (ISA) is a superset of MIPS ISA - a popular embedded processor. By changing SimpleScalar parameters, we could change the branch prediction scheme for the experiments.

Note that our technique is estimating the Worst Case Execution Time (called *estimated WCET*). To find the accuracy of our estimation technique, we need the actual Worst Case Execution Time (called *actual WCET*). Clearly, *estimated WCET* $\geq$ *actual WCET*. In addition, we must ensure that the difference (*estimated WCET* - *actual WCET*) should be small. Among the benchmarks, `matsum`, `matmult`, `fft`, `fdct`, and `dhry` which have only one possible input, the actual WCET can be computed via SimpleScalar simulation. For other programs, finding actual WCET is computationally infeasible. So, we used human guidance to select certain inputs which are suspected to increase execution time via mispredictions. We then simulated the programs with these selected inputs and reported the maximum observed execution time (called *observed WCET*). Therefore *estimated WCET* $\geq$ *actual WCET* $\geq$ *observed WCET*.

We wrote a prototype analyzer that accepts assembly language code annotated with loop bounds. Our analyzer is parameterized w.r.t. predictor table size, choice of prediction schemes and misprediction penalty. This makes our branch prediction analyzer retargetable w.r.t. various processor micro-architectures. The analyzer first disassembles the code, identifies the basic blocks and constructs the control flow graph (CFG). From the CFG, our analyzer automatically generates the objective function and the linear constraints. These constraints are then submitted to an ILP solver. For our experiments, we used CPLEX [4], a commercial ILP solver distributed by ILOG.

Accuracy. To evaluate the accuracy of our branch prediction modeling, we present the experiments for three different branch prediction schemes: *gshare*, *GAg* and *local*. Since finding the worst case input of a benchmark (which produces the actual WCET) is a human guided and tedious process, we only measured the actual WCET assuming a 4-entry prediction table. The results appear in Table 2. Even though not shown here due to space shortage, the estimation accuracy was independent of the prediction table size. Our estimation technique obtains a very tight bound on the WCET and misprediction count in all benchmarks except `fft`. The reason is that the number of iterations of the innermost loop of `fft` depends on the loop iterator variable value of the outer loops. This can be captured by providing inequations obtained from data-flow analysis of the loop iterator variables.

Performance. We formulated the timing analysis problem (for *gshare* scheme) with larger branch prediction table sizes varying from 32–1024 entries. Recall that in *gshare*, the branch instruction address is XOR-ed with the global branch history bits. In practice, *gshare* scheme uses smaller number of history bits than address bits, and XORs the history bits with the higher order address bits [12]. The choice of the number of history bits in a processor depends on the expected workload. In our experiments, we used a maximum of 4 history bits as it produces the best overall branch prediction performance across all our benchmarks.

On a Pentium IV 1.3 GHz processor with 1 GByte of main memory, our timing estimation technique requires less than 0.5 second for all the benchmarks.

6. RELATED WORK

Little work has been done to study the effects of branch prediction on a program's execution time. Effects of static branch prediction have been investigated in [2, 10]. How-

Pgm.	gshare				GAg				local			
	WCET		Mispred		WCET		Mispred		WCET		Mispred	
	Obs.	Est.	Obs.	Est.	Obs.	Est.	Obs.	Est.	Obs.	Est.	Obs.	Est.
check	611	611	3	3	611	611	3	3	1,196	1,196	198	198
matsum	101,417	101,417	204	204	101,417	101,417	204	204	101,405	101,405	200	200
matmul	14,732	14,732	223	223	14,732	14,732	223	223	14,663	14,663	200	200
fdct	2,493	2,493	7	7	2,493	2,493	7	7	2,484	2,484	4	4
fft	213,052	223,640	3,110	6,865	217,048	231,336	4,442	9,205	219,220	219,279	5,166	5,192
isort	74,225	74,742	9,687	9,954	46,526	46,548	587	596	46,447	46,447	399	399
bsearch	104	104	9	9	104	107	9	10	95	98	6	7
eqntott	2,311	2,314	203	204	2,308	2,319	202	205	2,311	2,314	203	204
dhry	122,026	124,297	2,207	2,812	122,617	123,479	2,404	2,606	122,005	122,276	2,200	2,205

Table 2: Observed and estimated WCET (in number of processor cycles) and misprediction count with gshare, GAg, and local schemes.

ever, most current day processors (Intel Pentium, AMD, Alpha, SUN SPARC) implement dynamic branch prediction schemes, which are more difficult to model. To the best of our knowledge, [3] is the only other work on timing estimation under dynamic branch prediction. Their technique is similar to cache modeling techniques [5] and cannot be used to model global branch prediction schemes.

Using Integer Linear Programming (ILP) for WCET analysis is not new. In particular, [9] has reduced the WCET analysis of instruction cache behavior into an ILP problem. In [17], ILP has been used for program path analysis subsequent to abstract interpretation based micro-architectural modeling of instruction cache, pipelines etc.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a framework to measure the effects of speculative execution on the Worst Case Execution Time of a program. Our modeling extends existing work on modeling static branch prediction [2] and uniformly captures various dynamic branch prediction schemes (which are used in both general-purpose and embedded processors [8, 13]). Using our technique, we have obtained tight timing estimates for benchmark programs under various branch prediction schemes. In future we plan to integrate our modeling with existing micro-architectural modeling of pipeline and cache for analyzing program execution time.

8. ACKNOWLEDGMENTS

This work was partially supported by National University of Singapore research grant R-252-000-088-112.

9. REFERENCES

[1] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR96-1308, University of Wisconsin - Madison, 1996.

[2] K. Chen, S. Malik, and D.I. August. Retargetable static software timing analysis. In *IEEE/ACM Intl. Symp. on System Synthesis (ISSS)*, 2001.

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

[4] CPLEX. The ilog cplex optimizer v7.5, 2002. Commercial software, http://www.ilog.com.

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

[6] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *IEEE Real-time Applications Symposium (RTAS)*, 1998.

[7] J.L. Hennessy and D.A. Patterson. *Computer Architecture- A Quantitative Approach*. Morgan Kaufmann, 1996.

[8] SiByte Inc. SiByte SB-1 MIPS64 embedded CPU Core. In *Embedded Processor Forum*, 2000.

[9] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.

[10] S-S. Lim, J.H. Han, J. Kim, and S.L. Min. A worst case timing analysis technique for in-order superscalar processors. Technical report, Seoul National University, 1998. *Earlier version published in IEEE Real Time Systems Symposium(RTSS) 1998.*

[11] T. Lundqvist and P. Stenstrom. Integrating path and timing analysis using instruction-level simulation techniques. In *Intl. Workshop on Languages, Compilers and Tools for Embedded Systems*, 1998.

[12] S. McFarling. Combining branch predictors. Technical report, DEC Western Research Laboratory, 1993.

[13] IBM Microelectronics. PowerPC 440GP Embedded Processor. In *Embedded Processor Forum*, 2001.

[14] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-time Systems*, 1(2), 1989.

[15] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *ACM Intl. Workshop on Languages, Compilers and Tools for Embedded System*, 1999.

[16] A.C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 1(2), 1989.

[17] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analysis. *Journal of Real Time Systems*, May 2000.

[18] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM Intl. Symp. on Computer Architecture (ISCA)*, 1992.