A Framework to Model Branch Prediction for Worst Case Execution Time Analysis

Tulika Mitra
Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543.
tulika@comp.nus.edu.sg

Abhik Roychoudhury
Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543.
abhik@comp.nus.edu.sg

ABSTRACT

Estimating the Worst Case Execution Time (WCET) of a program on a given hardware platform is useful in the design of embedded real-time systems. These systems communicate with the external environment in a timely fashion, and thus impose constraints on the execution time of programs. Estimating the WCET of a program ensures that these constraints are met. WCET analysis schemes typically model micro-architectural features in modern processors, such as pipeline and caches, to obtain tight estimates.

In this paper, we study the effects of dynamic branch prediction on WCET analysis. Branch prediction schemes predict the outcome of a branch instruction based on past execution history. This allows the program execution to proceed by speculating the control flow. Branch predictions schemes can be local or global. Local schemes predict a branch outcome based exclusively on its own execution history whereas global schemes take into account the outcome of other branches as well. Current WCET analysis schemes have largely ignored the effect of branch prediction.

Our technique combines program analysis and microarchitectural modeling to estimate the effects of branch prediction. Starting from the control flow graph of the program, we derive linear inequalities for bounding the number of mispredictions during execution (for all possible inputs). These constraints are then solved by any standard integer linear programming solver. Our technique models local as well global branch prediction in a uniform fashion. Although global branch prediction schemes are used in most modern processors, their effect on WCET has not been modeled before. The utility of our method is illustrated through tight WCET estimates obtained for benchmark programs.

Keywords

Worst Case Execution Time, Embedded code, Branch prediction, Program analysis, Microarchitectural modeling

1. INTRODUCTION

Estimating the *Worst Case Execution Time* (WCET) of a program is an important problem [7, 8, 9, 14, 17, 18, 22, 25]. WCET analysis computes an upper bound on the program's execution time on a particular processor for all possible inputs. The immediate motivation of this problem lies in the design of embedded real-time systems [13]. Typically an embedded system contains processor(s) running specific application programs and communicating with an external environment in a timely fashion. Many embedded systems are safety critical, *e.g.*, automobiles and power plant applications. The designers of such embedded systems must ensure that all the real-time constraints are satisfied. Real-time constraints impose hard deadlines on the execution time of embedded software. WCET analysis of the program can guarantee that these deadlines are met.

Static program analysis techniques have been used for estimating the WCET of a program [22, 25]. These works disregard the underlying hardware execution platform of the program. Modern processors employ advanced micro-architectural features such as pipeline, caches, and branch prediction to speed up program execution. By modeling these features we can get accurate WCET estimates. Hence, substantial research has been devoted to combining program analysis and micro-architectural modeling for WCET estimation [4, 8, 14, 15, 18, 23]. Among the micro-architectural features, the effect of pipeline and cache (particularly instruction cache) has been extensively studied. However, the effect of branch prediction has been largely ignored.

Branch prediction. 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 [11]. 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.

A classification of branch prediction schemes appears in Figure 1. Branch prediction can be *static* or *dynamic*. Static schemes associate a fixed prediction to each branch instruction via compile time analysis [1]. Almost all modern processors, however, predict the branch outcome dynamically based on past execution history [11]. Dynamic schemes are

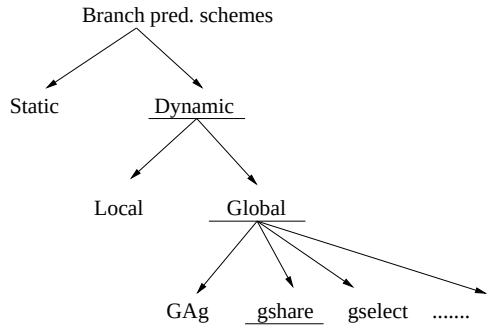

Figure 1: Classification of Branch Prediction Schemes. At each level, the more widely used category is underlined.

more accurate than static schemes, and in this work *we study only dynamic branch prediction.*

Dynamic prediction schemes use prediction table(s) to store the outcomes of the recently executed branches. These past outcomes are used to predict future branches. The simplest branch prediction schemes are *local* [19]; the prediction of a branch instruction I is based exclusively on the past outcomes of I. Most modern processors however use *global* branch prediction schemes [27], which are more accurate. Examples of embedded processors using global branch prediction include IBM PowerPC 440GP [20] and SB-1 MIPS 64 [12], both of which use the popular *gshare* prediction scheme [19]. In these schemes, the prediction of the outcome of a branch I not only depends on I's recent outcomes, but also on the outcome of the other recently executed branches. Global schemes can exploit the fact that behavior of neighboring branches in a program are often correlated.

In this paper, we model the effects of branch prediction on WCET of a program. Our technique combines program analysis with micro-architectural modeling to obtain linear constraints on the total misprediction count. These linear constraints are automatically generated from the control flow graph of the program, and are solved by a standard integer linear programming (ILP) solver. For every branch instruction I our constraints bound (a) the number of times I looks up a specific entry in the prediction table (b) the number of times this results in a misprediction. These bounds are then used to estimate the maximum number of mispredictions of I for all possible inputs.

Summary of contributions. The contributions of this paper are summarized as follows.

- We model the effects of local *and* global schemes for branch prediction on a program's WCET. Until recently, effects of dynamic branch prediction have been considered to be difficult to model statically [3]. To the best of our knowledge, ours is one of the first works to estimate the effects of (dynamic) branch prediction on the WCET of a program. Earlier, Colin and Puaut [4] have investigated the effects of local branch prediction schemes for estimating WCET. Conceptually, this work is a simple modification of cache modeling techniques for WCET. As a result, it *cannot* be extended to global branch prediction schemes (which are routinely used in modern processors).
- The only assumption made by our modeling is the presence of a single prediction table. The various prediction schemes differ from each other in how they index into this table (for finding the prediction of a branch instruction). By defining this index appropriately, we have been able to *reuse the same constraints* for modeling the effects of local and global branch prediction schemes (including popular schemes such as *gshare*). This gives a common framework for studying the effects of various prediction schemes on WCET.
- Even though our technique is fully ILP based, the insights gained from our modeling can be useful for other WCET analysis techniques. In Section 6 we discuss which of the constraints in our modeling can be used in a multi-phased WCET analysis (where path analysis and architectural modeling are separated).

2. RELATED WORK

Analyzing the WCET of a program has been extensively investigated. Earlier works [22, 25] estimated the WCET of a program by finding the longest execution path via program analysis. The cost of a path is the sum of the costs of the different instructions, assuming that the execution time of each instruction is constant. Presence of performance enhancing micro-architectural features such as pipeline, cache and branch prediction make this cost model inapplicable. Due to control and data dependences between instructions, the execution time of an instruction in a pipelined processor depends on the past instructions. Recent works on WCET analysis have therefore modeled micro-architectural features such as pipelined processors [6, 9], superscalar processors [16] and cache memories [8, 14].

Little work has been done to study the effects of branch prediction on a program's WCET. Effects of static branch prediction on WCET have been investigated in [3, 16]. However, 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, [4] is the only other work on estimating WCET under dynamic branch prediction. Their technique models the prediction table as a direct mapped cache, and is applicable to *only* local prediction schemes. In particular, [4] considers a scheme where the prediction for a particular branch instruction is either absent or present in a *specific* row of a prediction table. This assumption does not hold for global schemes where a branch instruction's prediction may reside in various rows of the prediction table.

Using Integer Linear Programming (ILP) for WCET analysis is not new. In particular, [14] has reduced the WCET analysis of instruction cache behavior into an ILP problem. In [7, 26], ILP has been used for program path analysis subsequent to abstract interpretation based micro-architectural modeling of instruction cache, pipelines etc.

3. BRANCH PREDICTION SCHEMES

As mentioned before, existing branch prediction schemes can be broadly categorized as *static* and *dynamic* (refer Figure 1). In a static scheme, a branch is predicted in the same direction every time it is executed. Either the compiler can attach a prediction bit to every branch through

analysis, or the hardware can predict using simple heuristics, such as backward branches are predicted taken and forward branches are predicted non-taken. However, static schemes are much less accurate than dynamic schemes.

Dynamic schemes predict a branch depending on the execution history. The first dynamic technique proposed is called *local branch prediction*, where each branch is predicted based on its last few outcomes. This is called "local" because prediction of a branch is *only* dependent on its *own* history [11]. 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. A more accurate version of local scheme uses k-bit counter per table entry.

The local prediction scheme cannot exploit the fact that a branch outcome may be dependent on the outcomes of other recent branches. The *global branch prediction* schemes (also called correlation based schemes in architecture literature) can take advantage of this situation [27]. 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* (also called Pattern History Table (PHT) in architecture literature) 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.

Among the global schemes, three are quite popular and have been widely implemented [19]. In the *GAg* scheme, the BHR is simply used as an index to look up the prediction table. In the popular *gshare* scheme, the BHR is XORed 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*. Finally, in the *gselect (GAp)* scheme, the BHR is concatenated with the last few bits of the branch address to look up the table.

Note that even with accurate branch prediction, the processor needs the target of a taken branch instruction. Current processors employ a small branch target buffer to cache this information. We have not modeled this buffer in our analysis technique; its effect can be modeled via instruction cache analysis techniques [14]. Furthermore, the effect of the branch target buffer on a program's WCET is small compared to the total branch misprediction penalty. This is because the target address becomes available at the beginning of the pipeline, whereas the branch outcome becomes available near the end of the pipeline.

4. MODELING BRANCH PREDICTION

In this section, we illustrate how the effects of branch prediction on WCET analysis can be estimated. We consider *GAg*, a global branch prediction scheme [19, 27]. This scheme and its variations (such as *gshare*) have been widely implemented in general-purpose and embedded processors, *e.g.*, IBM PowerPC 440GP [20], AMD, Alpha. However, our modeling is generic and not restricted to *GAg*. In Section 6, we will demonstrate how it easily captures other global prediction schemes as well as local schemes.

4.1 Issues in Modeling Branch Prediction

Micro-architectural features such as pipeline and instruction cache have been successfully modeled for WCET analysis. In the presence of these features, the execution time of an instruction may depend on the past execution trace. For pipeline, these dependencies are typically local. That is, the execution time of an instruction may depend only on the past few instructions, which are still in the pipeline. To model the effect of caches and branch prediction, more complicated *global analysis* is required. This is because both cache hit/miss of an instruction and correct/incorrect prediction of a branch instruction depend on the complete execution trace so far. However, there are two significant differences between global analysis of instruction cache and branch prediction.

Both instruction cache and branch prediction maintain global data structures that record information about past execution trace, namely the cache and the branch prediction table. For instruction cache, a given instruction can reside only in one row of the cache: if it is present it is a cache hit, otherwise it is a cache miss. Local branch prediction is quite similar - outcomes of a given branch instruction is stored only in one entry of the prediction table and prediction of the branch instruction depends on the content of only that entry. However, for global branch prediction schemes, a given branch instruction may use different entries of the prediction table at different points of execution. Given a branch instruction I, a global branch prediction scheme uses the history $\mathcal{H}_I$ (which is the outcome of last few branches before arriving at I) to decide the prediction table entry. Because it is possible to arrive at I with various history, the prediction for I can use different entries of the prediction table at different points of execution.

The other difference between instruction cache and branch prediction modeling is obvious. In case of instruction cache, if two instructions I and I' are competing for the same cache entry, then the flow of control either from I to I' or from I' to I will always cause a cache miss. However, for branch prediction, even if two branch instructions I and I' map to same entry in the prediction table, the flow of control between them does not imply correct or incorrect prediction. Their competition for the same entry may have constructive or destructive effect in terms of branch prediction depending on the outcomes of the branches I and I'.

4.2 WCET Estimation Technique

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 is labeled.

$$
\begin{array}{rrl}
label(i \rightarrow j) = & U \text{ if } i \rightarrow j & \text{denotes unconditional flow} \\
& 0 \text{ if } i \rightarrow j & \text{denotes control flow via} \\
& & \text{not taking the branch at } i \\
& 1 \text{ if } i \rightarrow j & \text{denotes control flow via} \\
& & \text{taking the branch at } i
\end{array}
$$

For any basic block i, if the last instruction of i is a branch

then it has two outgoing edges labeled 0 and 1. Otherwise, basic block i has only one outgoing edge with label U.

We construct the CFG of programs with procedures and functions (recursive or otherwise) in the following way. For every distinct call of a procedure P in the source code, we create a separate copy of the CFG of procedure P. Thus, each call of P transfers control to its corresponding copy. However, we require any basic block b appearing in the CFG of P to have the same instruction addresses in all copies of the CFG. Thus our approach is *not* equivalent to procedure inlining.

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 inflow equals outflow for each basic block i (except the start and end nodes), we have the following equations.

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

Furthermore, as the start and end blocks are executed exactly once, we get:

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

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 via techniques such as [10].

Defining WCET. 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 $m_i = 0$. To find the worst case execution time, we need to maximize the above objective function. For this purpose, we need to 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), then the index would be 10. We define v_i^π and m_i^π: the execution count and the misprediction count of block i when i is executed with Branch History Register $= \pi$. By definition:

$$m_i^\pi \leq v_i^\pi$$

$$m_i = \sum_\pi m_i^\pi \quad \text{and} \quad v_i = \sum_\pi v_i^\pi$$

For each basic block i and history π, we find out whether it is possible to reach block i with history π. This information can be obtained via static analysis of the control flow graph and is denoted by a predicate *poss* where:

$$poss(i, \pi) = \begin{array}{ll} true & \text{if } i \text{ can be reached with history } \pi \\ false & \text{otherwise.} \end{array}$$

Clearly, if $\neg poss(i, \pi)$ then $v_i^\pi = m_i^\pi = 0$. Otherwise, we need to estimate v_i^π and m_i^π.

Control flow among history patterns. First, we define constraints on v_i^π. This provides an upper bound on m_i^π. 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:*

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

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 $j \rightarrow i$, and $\Gamma(\pi', j \rightarrow i) = \pi$.

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

$$v_i^\pi \leq \sum_{\substack{j \\ j \rightarrow i}} \sum_{\substack{\pi' \\ \pi = \Gamma(\pi', j \rightarrow i)}} v_j^{\pi'}$$

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

$$v_i^\pi \leq \sum_{\substack{j \\ i \rightarrow j}} v_j^{\pi'}$$

where $\pi' = \Gamma(\pi, i \rightarrow j)$. In this case, for any outgoing edge $i \rightarrow j$, there can be only one such π'.

Repetition of a history pattern. Let us suppose there is a misprediction of the branch in block i with history π. This means that certain blocks (including i itself) were executed with history π; the outcome of these branches appear in the πth row of the prediction table. Furthermore, 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 the quantity $p^\pi_{i \rightsquigarrow j}$ below. Note that if history π occurs at a basic block with a branch instruction, then the πth row of the prediction table is looked up for branch prediction.

DEFINITION 2. *Let i be either the start block of the control flow graph or a basic block with branch instruction. Let j be*

either the end block of the control flow graph, or a basic block with a branch instruction. Let π be a history pattern. Then $p^{\pi}_{i \rightsquigarrow j}$ 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, $p^{\pi}_{i \rightsquigarrow j}$ 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 branch prediction at blocks i and j, and (b) the π th row of the prediction table is never used for branch 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, $p^{\pi}_{start \rightsquigarrow i}$ ($p^{\pi}_{i \rightsquigarrow end}$) 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 $p^{\pi}_{start \rightsquigarrow i}$) 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 $p^{\pi}_{i \rightsquigarrow end}$) or it is used for branch prediction next time in block $j \neq end$. Since v^{π}_i denotes the number of times block i uses the πth row of prediction table (the execution count of block i with history π), therefore:

$$v^{\pi}_i = \sum_j p^{\pi}_{j \rightsquigarrow i} = \sum_j p^{\pi}_{i \rightsquigarrow j}$$

Also, there can be at most one first use, and at most one last use of the π th row of the prediction table during program execution. Therefore we get

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

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

Introducing branch outcomes. To model mispredictions, we not only need to model the repetition of history patterns, but also the branch outcomes. Misprediction occurs on differing branch outcomes for the same history pattern. For this reason, we define two new variables $p^{\pi,1}_{i \rightsquigarrow j}$ and $p^{\pi,0}_{i \rightsquigarrow j}$. Let i be a basic block with a branch instruction. Then it has two outgoing edges $i \rightarrow k$ and $i \rightarrow l$ labeled 1 and 0 (corresponding to the branch being taken or not taken). For any block j and any index π, let $Allpaths(p^{\pi}_{i \rightsquigarrow j})$ denote the set of program paths contributing to the count $p^{\pi}_{i \rightsquigarrow j}$. Any such path must either begin with the edge $i \rightarrow k$ or the edge $i \rightarrow l$. We define the following:

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

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

Also, for any block j and history pattern π, a path contributing to the count $p^{\pi,1}_{i \rightsquigarrow j}$ must begin with the edge $i \rightarrow k$ *i.e.* branch at block i is taken. A similar constraint holds for the count $p^{\pi,0}_{i \rightsquigarrow j}$. Thus:

$$\sum_j \sum_\pi p^{\pi,1}_{i \rightsquigarrow j} \leq e_{i,k} \quad \text{and} \quad \sum_j \sum_\pi p^{\pi,0}_{i \rightsquigarrow j} \leq e_{i,l}$$

Recall that $e_{i,k}$ and $e_{i,l}$ denote the execution counts of the edges $i \rightarrow k$ and $i \rightarrow l$ respectively.

Modeling mispredictions. As mentioned before, misprediction occurs at block i if there are differing branch outcomes for a repeating history pattern. 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 the prediction table is indexed by π, the history pattern. For every basic block i with a branch instruction, for every index π, we need to estimate m^{π}_i. It 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 in which block i is mispredicted with history π.

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

From the above, we derive the following bound on m^{π}_i

$$\begin{aligned} m^{\pi}_i \leq \quad & min(\sum_j p^{\pi,1}_{i \rightsquigarrow j} \ , \ \sum_j p^{\pi,0}_{j \rightsquigarrow i}) \\ + \quad & min(\sum_j p^{\pi,0}_{i \rightsquigarrow j} \ , \ \sum_j p^{\pi,1}_{j \rightsquigarrow i}) \end{aligned}$$

This constraint can be straightforwardly rewritten into linear inequalities by introducing new variables (which we do not show here). Also, we derived the above bound on m^{π}_i by assuming that each row of the prediction table contains one bit. For this, we considered (a) possible outcomes at block i with history π (b) possible last use of the π th row of the prediction table before arriving at i. If each row of the prediction table contains $k > 1$ bits (in practice at most 2 bits are used) we can constrain m^{π}_i similarly. In particular, 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. In the above, 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

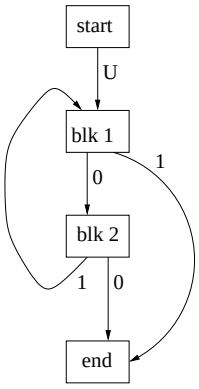

Figure 2: Example Control Flow Graph

function subject to these constraints using an (integer) linear programming solver. This gives an upper bound on the program's WCET.

5. AN EXAMPLE

In this section, we illustrate our WCET estimation technique with a simple example. Consider the control flow graph in Figure 2. The start and end blocks are called "*start*" and "*end*" respectively. 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 $cost_{start} = cost_1 = 2$, $cost_2 = 4$ and $cost_{end} = 2$. 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, since 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 control flow graph. 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 now 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 are also defined similarly.

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:

$$v_1^{01} \leq v_2^{10} + v_2^{00}$$

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 $2 \rightarrow 1$ *i.e.* $\Gamma(00, 2 \rightarrow 1) = \Gamma(10, 2 \rightarrow 1) = 01$ from Definition 1.

From the outflows of the executions of block 1 with history 01 we have:

$$v_1^{01} \leq v_2^{10} + v_{end}^{11}$$

If the branch at block 1 is taken, then control flows to block *end* and the history is 11 *i.e.* $\Gamma(01, 1 \rightarrow end) = 11$. Otherwise the branch is not taken and the control flows to block 2 with history 10 *i.e.* $\Gamma(01, 1 \rightarrow 2) = 10$.

The constraints for the other blocks and patterns are derived similarly.

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, the constraints are as follows:

Exec. with pattern 01	Inflow from last occurrence of 01	Outflow to next occurrence of 01
v_1^{01}	$= p_{1 \rightsquigarrow 1}^{01} + p_{start \rightsquigarrow 1}^{01}$	$= p_{1 \rightsquigarrow 1}^{01} + p_{1 \rightsquigarrow end}^{01}$

Constraints for the other patterns are derived similarly.

Modeling mispredictions. Using the 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 dues 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.

6. EXTENSIONS

We now discuss the extensions of the technique for modeling other branch prediction schemes. We also discuss how (some of) our constraints can be used in a WCET analysis which is not exclusively based on ILP.

6.1 Modeling other prediction schemes

Our modeling is independent of the definition of the prediction table index, so far called as the history pattern π. All our constraints only assume the following: (a) the presence of a global prediction table, (b) the index π into this prediction table and (c) every time the π th row is looked up for branch prediction, it is updated subsequent to the branch outcome. These constraints continue to hold even if π does not denote the history pattern (as in the *GAg* scheme).

In fact, the different branch prediction schemes differ from each other primarily in how they index into the prediction table. Thus, to predict a branch I, the index computed is 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 branch address, while local schemes use only the branch address.

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 in Section 4. These constraints then bound a program's WCET (under the new branch prediction scheme).

Other global schemes. In *gshare* [19, 27], the index π used for a branch instruction I is defined 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 I's address, and $history_m$ denotes the most recent m branch outcomes (which are XORed with higher-order m bits of $address_n(I)$). The updating of π due to control flow is modeled by the function:

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

where $i \rightarrow j$ is an edge in the control flow graph and Γ is the function on history patterns described in Definition 1. The modeling of the *gselect* prediction scheme [27] is similar and is omitted for space considerations.

Local prediction schemes. In local schemes the index π into the prediction table for predicting the outcome of instruction I is $\pi = address_n(I)$. Here n is a constant and $address_n(I)$ denotes the least significant significant n bits of the address of branch instruction I. Updating of the index π due to control flow is given by $\Gamma_{local}(\pi, i \rightarrow j) = lsb_n(j)$. Again $i \rightarrow j$ is an edge in the control flow graph and $lsb_n(j)$ is the least significant n bits of the last instruction in basic block j. If j contains a branch instruction I, it must be the last instruction of j. Thus the least significant n bits of the address of I are used to index into the prediction table (as demanded by local schemes). If j does not contain any branch instruction, then the index computed is never used to lookup the prediction table.

6.2 Two phase WCET Analysis

In the technique presented in Section 4, the search for the longest path as well as the architectural modeling has been fused into a single step. In general, this may lead to a large number of variables and constraints in the ILP problem. Moreover, when various architectural features such as pipeline, cache and branch prediction are modeled together, the resulting ILP problem size can potentially blow up.

In this situation, we can sacrifice some of the accuracy of the ILP based approach and perform a faster, multi-phased analysis. This approach has been successfully applied for WCET analysis of (combinations of) instruction cache and pipeline behavior [7, 26]. In the case of branch prediction, we can achieve this separation of analysis as follows. In the first phase, we conservatively classify the branch instructions. For each history π, we find which branches are always correctly predicted when reached via history π. In the second phase we use ILP to maximize the execution time which is defined as $\sum_i v_i^\pi * time_i^\pi$. Here, v_i^π is the number of times block i is executed with history π and $time_i^\pi$ is the worst case execution time of block i with history π. Note that $time_i^\pi$ is set to be a constant. Let t_i be the execution time of block i without branch misprediction. Then $time_i^\pi = t_i$ if the branch in block i was inferred (in the first phase) to be always correctly predicted with history π. Otherwise we set $time_i^\pi = t_i +$ branch penalty. Of course, if block i has no branch instruction we set $time_i^\pi = t_i$ for all π.

Note that in the two-phased approach for WCET analysis of branch prediction, v_i^π are variables of the ILP problem in the second phase. Thus, all constraints on v_i, v_i^π, $e_{i,j}$ and $p_{i \rightsquigarrow j}^\pi$ developed in Section 4 can be directly used in the second phase of the analysis. Note that this includes constraints which model the control flow among the history patterns, a concept central to branch prediction. The constraints modeling mispredictions via the variables m_i^π will no longer be needed.

The above outlines a method for a two-phase WCET analysis of branch prediction behavior. We have discussed it here mainly to show how certain aspects of our modeling can be used in other WCET analysis techniques. We have not implemented the two-phase approach. We now present experimental results for our ILP-based WCET analysis.

7. EXPERIMENTAL RESULTS

We selected eight different benchmarks for our experiments (refer Table 1). Out of these benchmarks, `check`, `matsum`, `matmult`, `fft` and `fdct` are loop intensive programs where almost all the executed conditional branches are loop branches. Among these programs, the `fft` program is challenging because its innermost loop has a variable number of iterations. Thus the loop branch instruction of the innermost loop of `fft` is hard to predict.

We also chose three other programs: `isort`, `bsearch` and `eqntott`. These programs execute large number of *hard-to-predict* conditional branches arising from if-then-else statements within nested loops. More importantly, the behavior of the conditional branches arising from the if-then-else statements are correlated in `bsearch` and `eqntott`. In fact, `eqntott` was one of the benchmarks used in branch prediction research to show the utility of global prediction schemes (also called correlation-based prediction) [21]. Including such programs into our benchmark suite allows us to test the accuracy of our global branch prediction modeling.

7.1 Methodology

Processor Model. In our experiments we assumed a perfect processor pipeline with no data dependency and cache misses, except for control dependency via conditional branch

Benchmark	Description	No. of lines in source code
`check`	Checks for negative elements in 100-element array	20
`matsum`	Summation of two 100×100 matrices	12
`matmul`	Multiplication of two 10×10 matrices	16
`fft`	1024-point Fast Fourier Transform	53
`fdct`	Fast Discrete Cosine Transform	346
`isort`	Insertion sort of 100-element array	28
`bsearch`	Binary search of 100 element array	27
`eqntott`	Drawn from SPEC'92 integer benchmarks Translates Boolean equation to Truth table	44

Table 1: Description of benchmark programs.

instructions. More concretely, we assumed that all instructions (except conditional branches) take a fixed number of clock cycles to execute. These assumptions, although simplistic, allow us to separate out and measure the effect of branch prediction on WCET. We assumed that the branch misprediction penalty is 3 clock cycles (as in the Intel Pentium processor). Needless to say, any other choice of misprediction penalty can also be used.

Simulation Platform. To evaluate the accuracy of our analysis, we need to choose a target hardware platform so that we can measure the actual WCET. Unfortunately, no single hardware platform will allow us to measure actual WCET for different branch prediction schemes. Therefore, we decided to use the SimpleScalar architectural simulation platform [2]. SimpleScalar instruction set architecture (ISA) is a superset of MIPS ISA - a popular embedded processor. By changing simulator parameters, we could change the branch prediction scheme and measure the actual WCET. The programs were compiled for SimpleScalar ISA using gcc.

Finding Actual WCET. Given a branch prediction scheme and a benchmark program, we attempted to identify the program input that will generate the WCET. Once this input is found, the actual WCET and worst case profile can be computed via SimpleScalar simulation. Among the benchmarks, `matsum`, `matmult`, `fft` and `fdct` have only one possible input. Since the other programs have many possible inputs, determining the worst-case input can be tedious. In these programs, 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 profile for the input which maximizes the execution time. The input that produces this maximum observed execution time (called "Observed WCET" in Table 2) is the "*observed worst case input*".

Interestingly, for the same program, the observed worst case input is different for different branch prediction schemes. For example, in the insertion sort program `isort`, the observed worst case inputs are (assuming a four entry prediction table in all schemes): $\langle 100, 99, \ldots, 2, 1\rangle$ for *gshare* and *local* schemes and $\langle 1, 100, 99, \ldots, 2\rangle$ for *GAg* scheme. Thus in the *GAg* scheme, the effects of branch prediction make the input $\langle 1, 100, 99, \ldots, 2\rangle$ perform worse than the *longest* execution path resulting from $\langle 100, 99, \ldots, 2, 1\rangle$.

Tool for Estimating Mispredictions. 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 microarchitectures. The analyzer first disassembles the code, identifies the basic blocks and constructs the the control flow graph (CFG). Note that we take care of procedure and function calls at this stage in the manner described in Section 4.2. 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 [5], a commercial ILP solver distributed by ILOG.

7.2 Accuracy

To evaluate the accuracy of our branch prediction modeling, we present the experiments for three different branch prediction schemes: *gshare*, *GAg* and *local*. For this purpose, we need the actual WCET of a program. As explained earlier, this depends on the program input for some of our benchmarks. 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 for all the benchmarks and all the prediction schemes assuming a 4-entry prediction table. The results appear in Table 2.

In Table 2, the observed WCET is a lower bound on the actual WCET which is obtained by SimpleScalar simulation of the observed worst-case input. The estimated WCET is an upper bound of the actual WCET which is computed by our estimation technique. The observed (estimated) misprediction count is the total misprediction count that produces the observed (estimated) WCET. 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. Hence the relationship between the loop iterator variable values for the different loops has to be inferred via data-flow analysis of these variables [4]. This is currently not captured by our technique which focuses on analyzing control flow.

For the benchmark programs with only one possible input, we used SimpleScalar simulation to measure the actual misprediction count for the *gshare* scheme with larger prediction tables (upto 1024 entries). The estimation accuracy of our technique (estimated count : actual count) with a 4-entry prediction table is roughly the same as the estimation accuracy with larger sized prediction tables. These numbers are not shown due to space considerations.

7.3 Scalability

One major concern with any ILP formulation of WCET is the scalability of the resulting solution. To check the scal-

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	1196	1196	198	198
`matsum`	101,417	101,417	204	204	101,417	101,417	204	204	101,405	101,405	200	200
`matmul`	15,735	15,735	223	223	15,735	15,735	223	223	15,666	15,666	200	200
`fdct`	2493	2493	7	7	2493	2493	7	7	2484	2484	4	4
`fft`	214,874	222,335	3678	6165	214,034	219,365	3398	5175	216,227	219,302	4129	5154
`isort`	74,225	74,871	9687	9952	46,526	46,554	587	598	46,447	46,447	399	399
`bsearch`	104	104	9	9	104	107	9	10	95	98	6	7
`eqntott`	2311	2317	203	205	2308	2320	202	206	2311	2314	203	204

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

ability of our solution, we formulated the WCET problem for the popular *gshare* scheme with branch prediction table size varying from 4–1024 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 [19, 24]. 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. As Figure 3 shows, the number of variables generated for the ILP problem initially increases and then decreases. With increasing number of history bits, number of possible patterns per branch increases. But with fixed history size and increasing prediction table size, the number of cases where two or more branches have the same pattern starts to decrease. This significantly reduces the number of $p^{\pi}_{i \rightsquigarrow j}$ variables.

Finally, Table 3 shows the time required by CPLEX to optimize the objective function under the linear constraints generated by our analyzer. On a Pentium IV 1.3 GHz processor with 1 GByte of main memory, CPLEX requires less than 0.11 second for all the benchmarks with prediction table size varying from 4–1024 entries.

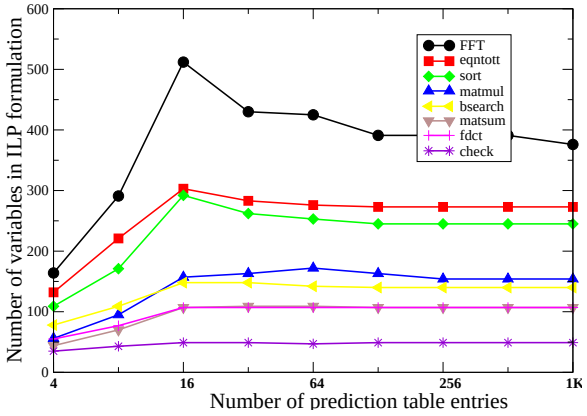

Figure 3: Change in ILP problem size with increase in number of entries in the branch prediction table for *gshare* scheme

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a framework to study the effects of dynamic branch prediction on the WCET of a program. To the best of our knowledge, it is the first work to model local and global branch prediction schemes for WCET analysis. Indeed, the only assumption made in our modeling is the presence of a single prediction table. In particular, we use program analysis and micro-architectural modeling to derive bounds on the misprediction count of conditional branch instructions in a program. Using our technique, we have obtained tight WCET estimates for benchmark programs under various branch prediction schemes.

There are several avenues for future research on this topic. As discussed in Section 6.2, we plan to investigate the the two-phase approach for estimating branch prediction behavior. This work will be similar to the study for instruction cache reported in [26]. It would be interesting to evaluate the accuracy performance tradeoff for the two-phased WCET analysis (as compared to the ILP based technique of this paper). Also, note that our technique requires the loop bounds of the program as user input. To automate this process, one can either employ a specialized technique for computing the loop bounds offline [10] or integrate the loop bound estimation into the WCET analysis framework [17, 18]. Integrating the loop bound estimation into our framework can significantly reduce the pessimism in our estimated WCET for benchmarks such as `fft`.

In the area of micro-architectural modeling, we plan to study other branch prediction schemes such as *PAg* [27]. This scheme uses two separate tables for branch prediction, the effect of which cannot be modeled by the read/write of a single global table. Extending our framework to model such schemes will increase the applicability of our technique.

Finally, this work focused on modeling branch prediction by assuming a perfect cache and a perfect pipeline where the only stalls arise from branch misprediction. This initiates the task of integrating branch prediction behavior with other micro-architectural features (pipeline, cache etc.) for analyzing WCET.

9. REFERENCES

[1] T. Ball and J. Larus. Branch prediction for free. In *ACM SIGPLAN Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 1993.

[2] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset.

Benchmark	Number of prediction table entries								
	4	8	16	32	64	128	256	512	1024
`check`	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
`matsum`	0.00	0.01	0.01	0.01	0.01	0.01	0.00	0.00	0.01
`matmul`	0.00	0.00	0.02	0.00	0.02	0.01	0.01	0.00	0.01
`fdct`	0.01	0.01	0.01	0.00	0.01	0.00	0.01	0.00	0.01
`fft`	0.01	0.02	0.11	0.03	0.06	0.03	0.02	0.02	0.03
`isort`	0.01	0.02	0.03	0.03	0.04	0.01	0.04	0.03	0.02
`bsearch`	0.01	0.00	0.07	0.04	0.03	0.01	0.00	0.00	0.02
`eqntott`	0.01	0.04	0.03	0.03	0.05	0.04	0.03	0.06	0.02

Table 3: Time to solve ILP problem in seconds.

Technical Report CS-TR96-1308, University of Wisconsin - Madison, 1996.

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

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

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

[6] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 1999.

[7] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Intl. Workshop on Embedded Software (EmSoft)*, 2001.

[8] 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.

[9] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), 1999.

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

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

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

[13] Y-T. S. Li and S. Malik. *Performance Analysis of Real-time Embedded Software*. Kluwer Academic Publishers, 1999.

[14] 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.

[15] S.-S. Lim, Y.H. Bae, G.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, and C.S. Kim. An accurate worst-case timing analysis technique for RISC processors. *IEEE Transactions on Software Engineering*, 21(7), 1995.

[16] 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.*

[17] Y.A. Liu and G. Gomez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12), 2001.

[18] 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 (LCTES)*, 1998.

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

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

[21] S-T. Pan, K. So, and J.T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

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

[23] 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 (LCTES)*, 1999.

[24] S. Sechrest, C-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *ACM International Symposium on Computer Architecture (ISCA)*, 1996.

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

[26] 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.

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