Effects of Branch Prediction on Worst Case Execution Time of Programs

Tulika Mitra and Abhik Roychoudhury

Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543.
{tulika,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 microarchitectural features in modern processors, such as pipeline and caches, to obtain tight estimates.

In this paper, we study the effects of speculative execution on WCET analysis. Speculative execution uses branch prediction to 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 [5, 7, 11, 14, 15, 17, 19]. 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 [10]. 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.

Apart from satisfying timing constraints in hard real-time systems, WCET analysis has other applications. Design of an embedded system typically involves additional stringent requirements on size, power consumption, and cost. An accurate WCET analysis enables an embedded system designer to satisfy these additional constraints by exploring a larger design space. For example, a designer may run the program on a cheaper processor with lower performance as long as WCET analysis concludes that the timing constraints are not violated.

Static program analysis techniques have been used for estimating the WCET of a program [17, 19]. However these works disregard the underlying hardware platform on which the program is executing. Modern processors employ advanced microarchitectural features such as pipeline, caches, and branch prediction to speed up program execution. Modeling these features allows us to obtain accurate WCET estimates. Hence, substantial research has been devoted to combining program analysis and microarchitectural modeling for WCET estimation [4, 5, 11, 12, 15, 18]. Among the microarchitectural 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 in modern processors. Speculating the control flow subsequent to a branch instruction can avoid the pipeline stalls. Current generation processors perform control flow speculation through branch prediction, which predicts the outcome of branch instructions [8]. 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 (varies between 3 – 19 clock cycles in different processors).

Branch 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* [16]; 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 [22], which are more accurate. 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.

Need to model branch prediction Many WCET analysis schemes assume perfect branch prediction. This assumption results in *incorrect* WCET, particularly when a hard-to-predict conditional statement (if-then-else) is present inside a loop body. Even though existing prediction schemes have a high average case prediction accuracy, the accuracy in the worst case can be quite low, that is, branch penalties can contribute substantially to a program's WCET. Thus, the incorrect WCET computed assuming perfect branch prediction can be substantially less than the actual WCET of a program. Alternatively, certain works assume that all branches in a program are mispredicted. This pessimism results in significant overestimation for WCET as branch prediction accuracy is high for loop control branches even in the worst case.

Issues in modeling branch prediction Microarchitectural 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 outcome of the branches I and I'.

Summary of contributions In this paper, we model the effects of branch prediction on WCET of a program. Our technique combines program analysis with microarchitectural 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 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.

Our modeling of branch prediction captures *various local and global branch prediction schemes.* The only assumption made by our modeling is the presence of a single prediction table. The various branch prediction schemes then 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 *re-use the same constraints* for modeling the effects of different branch prediction schemes. This gives a framework for studying the effects of branch prediction on WCET.

Ours is one of the first works on modeling branch prediction schemes for WCET analysis. It can be used to model both local and global branch predictions. Earlier, [4] has investigated the effects of local branch prediction schemes for estimating WCET. Conceptually, [4] is a simple modification of cache modeling techniques for WCET. As a result, it cannot be extended to global branch prediction schemes.

Organization The rest of the paper is organized as follows. The next section surveys related work on WCET analysis. Section 3 provides a brief discussion on branch prediction schemes. In section 4, we present our technique for estimating WCET in the presence of a global branch prediction scheme. Section 5 illustrates this technique with a concrete example. In section 6, we extend our method to model the effect of other branch prediction schemes: both local and global. Section 7 presents experimental results and conclusions appear in section 8.

2 Related Work

Analyzing the WCET of a program has been extensively investigated. Earlier works [17, 19] 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 [7], superscalar processors [13] and cache memories [5, 11].

Little work has been done to study the effects of branch prediction on a program's WCET. [13] models a very simple static branch prediction scheme for WCET analysis. In particular, their scheme predicts all forward branches to be not taken, and all backward branches to be taken. Since all current day processors (Intel Pentium, AMD, Alpha, SUN SPARC) implement dynamic branch prediction schemes, it is more realistic to model these schemes.

To the best of our knowledge, [4] is the only other work on estimating WCET under dynamic branch prediction. This work has several limitations. First of all, their analysis techniques classifies the branch instructions in a program into only four categories *e.g.* always mispredicted, first time mispredicted etc. The WCET is then estimated conservatively for each of these categories. Our analysis technique is more fine grained. We derive linear constraints on the misprediction count of each branch instruction, and then estimate the maximum value of the total number of mispredictions (under these constraints). Secondly, their technique is applicable to *only* local branch prediction. The modeling of [4] is very close to cache modeling techniques, and thus cannot be extended to global branch 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 (which acts as a cache). 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, [11, 20] have used ILP to model the effects of instruction cache on a program's WCET.

3 Branch Prediction Schemes

Existing branch prediction schemes can be broadly categorized as *static* and *dynamic*. In a static scheme, a branch is predicted in the same direction every time it is executed. A static scheme exploits the fact that most branches are biased towards one direction [1]. However, static schemes are much less accurate than dynamic schemes. Therefore, most modern processors employ dynamic branch prediction, and in this work we concentrate only on dynamic prediction.

Dynamic schemes predict a branch depending on the execution history. The first dynamic technique proposed was 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 [8]. 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. Notice that for a small table, two or more branches might map to the same table entry and they will affect one another's prediction (constructively or destructively).

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. Prediction of the branch will be same as the last outcome. When a branch is resolved, the corresponding table entry is updated with the outcome. A more accurate 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 can take advantage of this situation [22]. 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*[1] 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 implemented in various processors [16]:

1. *GAg:* The BHR is used as an index to look up the prediction table.

2. *gshare:* 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 different table indices compared to *GAg*.

3. *gselect (GAp):* The BHR is concatenated with the last few bits of the branch address to look up the prediction table. For example, gselect(4/4) concatenates 4 bit BHR with last 4 bits of the branch address.

As global branch prediction schemes can capture the correlation among the outcomes of different branches, they are also called correlation based schemes in literature [22].

Note that even with accurate branch prediction, the processor needs to know the target of a taken branch instruction to proceed. Current processors employ a small branch target buffer to cache this information for recently taken branches. We have not modeled this buffer in our analysis technique as it is straightforwardly handled via instruction cache analysis techniques [11]. 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. In particular, we consider *GAg*, a global branch prediction scheme [16, 22]. This scheme and its minor variations have been implemented in commercial microprocessors *e.g.* AMD, Alpha, Power PC, Ultra SPARC etc. 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.

To recapitulate, the *GAg* branch prediction scheme maintains a branch history register (BHR) which is the outcome of the last k branches. It also maintains a prediction table which is indexed by the contents of the BHR. Each row of the table contains a prediction for branch outcome: taken or not taken.

Control flow graph The starting point of our analysis is the control flow graph 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

[1]For global schemes, this prediction table is also called Pattern History Table (PHT) in architecture literature.

end node, such that all program paths originate at the start node, and terminate at the end node.

Each edge $i \to j$ of the control flow graph is labeled .

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

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.

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

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

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

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

We assume the absence of recursive and dynamic function calls; otherwise WCET analysis is undecidable [17]. Also, we provide constraints on the maximum number of iterations of loops in the programs. These bounds can be computed offline using techniques such as [6]. However these techniques are not applicable to all classes of programs, and in that case the loop bounds are user-provided.

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:

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

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 $i \to j$ 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 $i \to j$ is given by $\Gamma(\pi, i \to j)$ where:*

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

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 $j \to i$
- $\Gamma(\pi', j \to i) = \pi$

Note that for any incoming edge $j \to i$, there can be 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 basic block i (except the start block), from the inflows of i's execution with history π we get:

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

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

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

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

Repetition of a history pattern There can be two reasons for misprediction of the branch in block i with history π *i.e.* mispredictions of block i if the πth row of the prediction table is looked up.

- Certain blocks (including 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 create a prediction different from the current outcome of i

To model mispredictions, we need to capture repeated occurrence of a history π during program execution *i.e.* repeated reading/writing to the πth row of the prediction table. For this purpose, we define the quantity $p^{\pi}_{i \leadsto j}$ 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 $p^{\pi}_{i \leadsto j}$ is the number of times a path is taken from block i to block 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, $p^{\pi}_{i \leadsto j}$ denotes the number of times control flows from block i to block j s.t.

- π th row of the prediction table is used for branch prediction at blocks i and j.
- 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, $p^{\pi}_{start \leadsto i}$ models the numbers of times the π th row of the prediction table is looked up for the first time at block i. Similarly, $p^{\pi}_{i \leadsto end}$ is the number of times the π th row of the prediction table is looked up for the 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 $p^{\pi}_{start \leadsto i}$) 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 $p^{\pi}_{i \leadsto end}$) 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:

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

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

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

Furthermore, if $\neg poss(i, \pi)$ or $\neg poss(j, \pi)$ or j is not reachable from i then we set:

$$p^{\pi}_{i \leadsto j} = 0$$

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 $p^{\pi,1}_{i \leadsto j}$ and $p^{\pi,0}_{i \leadsto j}$. Let i be a basic block with a branch instruction. Then it has two outgoing edges $i \to k$ and $i \to l$ labeled 1 and 0 (corresponding to the branch being taken or not taken). For any block j and any index π, let $Allpaths(p^{\pi}_{i \leadsto j})$ denote the set of program paths contributing to the count $p^{\pi}_{i \leadsto j}$. Any such path must either begin with the edge $i \to k$ or the edge $i \to l$. We define the following:

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

By definition:

$$p^{\pi}_{i \leadsto j} = p^{\pi,1}_{i \leadsto j} + p^{\pi,0}_{i \leadsto j}$$

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

$$\sum_j \sum_\pi p^{\pi,1}_{i \leadsto j} \leq e_{i,k}$$

$$\sum_j \sum_\pi p^{\pi,0}_{i \leadsto j} \leq e_{i,l}$$

Recall that $e_{i,k}$ and $e_{i,l}$ denote the execution counts of the edges $i \to k$ and $i \to l$ 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 $\leq \sum_j p^{\pi,1}_{i \leadsto j}$, 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
  - 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
  - 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_i^π

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

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_i^π 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, we can constrain m_i^π 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
- m_i : misprediction count of block i

We now maximize the objective function (denoting the execution time of the program) subject to these constraints using an (integer) linear programming solver. This gives an upper bound of 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 1. 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 four possible history patterns: 00, 01, 10, 11. Also, each row of the prediction table contains one bit to store the last outcome for that pattern: 0 for not taken and 1 for taken.

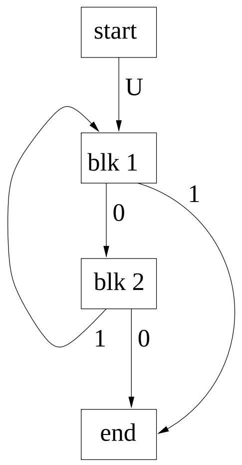

Figure 1: Example Control Flow Graph

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. In our method, this bound is either computed offline or user provided. Let us consider a loop bound of 100. Then,

$$e_{2,1} < 100$$

Defining WCET Let us assume a branch prediction 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:

$$\begin{aligned} poss(1, \pi) &= true \quad \text{if } \pi \in \{00, 01\} \\ &= false \quad \text{otherwise.} \end{aligned}$$

$$\begin{aligned} poss(2, \pi) &= true \quad \text{if } \pi \in \{00, 10\} \\ &= false \quad \text{otherwise.} \end{aligned}$$

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

$$m_1 = m_1^{00} + m_1^{01} \quad v_1 = v_1^{00} + v_1^{01}$$
$$m_2 = m_2^{00} + m_2^{10} \quad v_2 = v_2^{00} + v_2^{10}$$

$$m_1^{00} \le v_1^{00} \quad m_1^{01} \le v_1^{01}$$
$$m_2^{00} \le v_2^{00} \quad m_2^{10} \le v_2^{10}$$

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:

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

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

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

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

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

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:

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

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 $e_{i,j}$ a bound is obtained on the misprediction counts. As described in the last section, we get bounds for m_1^{00}, m_2^{00}, m_1^{01} and m_2^{10}. This gives constraints on the total number of mispredictions $m1 + m2$. The objective function is maximized subject to these constraints to obtain the WCET. The constraints for the misprediction counts for this particular example are not shown here due to space limitations.

6 Extending to other prediction schemes

In Section 4, we derived bounds on misprediction counts under a global branch prediction scheme (*GAg*). This models the effects of such a branch prediction scheme on the WCET of a program. The different prediction schemes differ from each other primarily in how they index into the prediction table. Thus, to predict a branch b, the index computed is a function of:

- the past execution trace (history)
- address of the branch instruction b

In the *GAg* scheme, the index computed is simply the outcome of the last k branches (where k is fixed a-priori). Thus, the index 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.

Our modeling is independent of the definition of the prediction table index, so far called as the history pattern π. All our constraints regarding history patterns, branch outcomes, and mispredictions only assume the following:

1. the presence of a global prediction table
2. π, the index into this prediction table.
3. 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).

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 We now discuss two other global prediction schemes: *gshare* and *gselect* [16, 22]. In *gshare*, the index π used for a branch instruction is defined as:

$$\pi = history_k \oplus address_k$$

where k is a constant, $\oplus$ is exclusive-OR, $history_k$ denotes the most recent k branch outcomes and $address_k$ denotes the least significant significant k bits of the branch instruction address. The updation of π due to control flow is modeled by the Γ_{gshare} function

$$\Gamma_{gshare}(\pi, i \to j) = \Gamma(history_k, i \to j) \oplus address_k$$

where $i \to j$ 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 is similar. Here the index π into the prediction table is defined as:

$$\pi = history_k \bullet address_k$$

where k is some constant and $\bullet$ denotes concatenation. The updation of π due to control flow is given by function $\Gamma_{gselect}$

$$\Gamma_{gselect}(\pi, i \to j) = \Gamma(history_k, i \to j) \bullet address_k$$

Again, $i \to j$ is an edge in the control flow graph and Γ is the function described in Definition 1.

Local schemes Local branch prediction schemes can be modeled as a simple instance of our framework. Note that local prediction schemes use only the branch instruction address to index into the prediction table. To predict the outcome of a branch instruction I, the index π into the prediction table is:

$$\pi = address_k$$

where k is a constant and $address_k$ denotes the least significant significant k bits of the address of branch instruction I. Thus, any particular branch instruction *always* indexes into a particular row of the prediction table. Two instructions share the same row if their last k address bits are identical. Updation of the index π due to control flow is given by the function Γ_{local}

$$\Gamma_{local}(\pi, i \to j) = lsb_k(j)$$

where $i \to j$ is an edge in the control flow graph and $lsb_k(j)$ is the least significant k 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 k 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 *i.e.* it is ignored.

7 Experimental Results

In this section, we evaluate the accuracy and performance of our worst case execution time analysis method.

7.1 Methodology

We selected six different benchmark programs for our experiments. The characteristics of these benchmark programs are given in Table 1. Note that some of our benchmarks, such as *sort* and *eqnt*, contain hard to predict branch instructions (*i.e.* branches whose behavior depend on program input) within nested loops. These branches make it challenging to analyze the worst case branch behavior.

In all our experiments we assumed a perfect processor pipeline with no data dependency and cache misses, except for control dependency via conditional branch instructions. More concretely, we assumed that each instruction takes a fixed number of clock cycles to execute and branch misprediction penalty is 3 clock cycles. This assumption enabled us to separate out the effect of branch prediction on WCET.

Note that we need to estimate the program's actual WCET to evaluate the accuracy of our analysis. Therefore, we need to choose a target hardware platform so that we can measure the actual WCET. Unfortunately, no single hardware platform will allow us measure actual WCET for different branch prediction schemes. Therefore, we decided to use the SimpleScalar architectural simulation platform [3]. 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.

In general, given a branch prediction scheme and a benchmark program, we attempted to identify the program input that will generate the WCET. Note that for the same program, the worst case input is different for different branch prediction schemes. We chose parameters for SimpleScalar to simulate the given branch prediction scheme, and then ran the program on SimpleScalar with the worst case input to measure actual WCET. However, for some benchmarks such as *eqnt*, it is non-trivial to determine the worst case input as well. For each of our benchmarks, the actual WCET was computed with human guidance.

We wrote a prototype analyzer that accepts assembly language program code annotated with loop bounds. The analyzer then generates the objective function and the linear constraints. We used a public domain LP solver *lp_solve* [2] to solve the generated constraints and obtain the estimated WCET. This LP solver produced integral bounds for all our benchmark programs *i.e.* an ILP solver was not needed.

7.2 Accuracy

To evaluate the accuracy of our branch prediction modeling, we present the experiments for two different branch prediction schemes: *GAg* and *Local*. We do not report the experiments for *gshare* and *gselect*, which are very similar to *GAg*. In order to stress the capability of our analysis method, we need to have as many conflicts as possible among the conditional branches in the program. Therefore, we decided to use only 4-entry branch prediction table for both the schemes. As a side remark, we note that *GAg* performs comparably to local schemes with a 4-entry prediction table. In practice however prediction tables of both the schemes are larger and global schemes perform significantly better [16].

GAg branch prediction Tables 2 and 3 show the results of our experiment for *GAg* branch predictor. Table 2 compares the actual number of mispredictions for WCET with mispredictions estimated by ILP solver. Table 3 shows the effect of branch prediction on WCET. The second column presents WCET in terms of processor clock cycles when all the branches are assumed to be mispredicted. The third column presents the other extreme of WCET calculation where all the branches are assumed to be correctly predicted. The fourth column shows the actual WCET. The table shows the importance of accurately modeling branch prediction for WCET analysis. The fifth column shows the estimated WCET via our analysis method and the final column gives the ratio of actual versus estimated WCET. The tables show that our method is very effective in obtaining tight bound for WCET with branch prediction. For *FFT*, the bound has more pessimism. The reason is that the innermost loop of *FFT* has variable maximum number of iterations, which was not precisely captured by our constraints.

Local branch prediction Tables 4 and 5 show the results of our experiment for local branch predictor. Again, our analysis method obtains a very tight bound on WCET. These results show that our analysis framework produces tight WCET estimates for both local and global branch prediction schemes.

7.3 Performance

Finally, Table 6 shows the time required by lp_solve to optimize the objective function under the linear constraints generated by our analyzer. On a Pentium IV 1.3 GHz processor with 1GByte of main memory, lp_solve requires less than 30 msec for all the benchmark programs. Note that lp_solve takes less time for local scheme than the global

Benchmark	Description	Code size in bytes
check	Checks if any element of 100-element array is negative	144
matsum	Summation of 2 100 × 100 matrix	184
matmul	Multiplication of 2 10 × 10 matrix	288
sort	Insertion sort of 100-element array	144
eqnt	Kernel of Eqntott program from SPEC'92	216
FFT	1024-point Fast Fourier Transform	856

Table 1: Characteristics of benchmark programs.

Benchmark	Actual	Estimated	Ratio
check	3	3	1.00
matsum	204	204	1.00
matmul	223	223	1.00
sort	587	598	1.02
eqnt	202	206	1.02
FFT	3398	5175	1.52

Table 2: Actual and estimated branch misprediction count with *GAg*.

Bench.	Pess.	Opt.	Actual	Est.	Ratio
check	1202	602	611	611	1.00
matsum	131105	100805	101417	101417	1.00
matmul	18396	15066	15735	15735	1.00
sort	75550	45250	46526	46554	1.00
eqnt	3007	1804	2358	2370	1.00
FFT	237653	203840	214034	219365	1.02

Table 3: Effects of *GAg* branch prediction scheme on WCET. All the numbers are given in terms of processor cycles. *Pessimistic* is WCET with 100% misprediction, *Optimistic* with no misprediction. *Actual* is the correct WCET and *Estimated* is the WCET with our analysis.

scheme. This is because each branch instruction has only one possible index into the prediction table under the local scheme as opposed to multiple possible indices under the global scheme. This generates significantly smaller number of variables to be solved and results in faster solution.

To check the scalability of our solution, we generated the linear constraints for *FFT* (the biggest of our benchmarks) with 512 entry *GAg* branch prediction table. For about 11,300 constraints generated by our analyzer, *lp_solve* took only 11 min to obtain the solution.

8 Conclusions and Future Work

In this paper, we presented a framework to study the effects of branch prediction on WCET of a program. In particular, we use program analysis and microarchitectural modeling to derive bounds on the misprediction count of conditional branch instructions in a program. Our modeling is generic, and captures a host of local and global branch prediction schemes. The only assumption we make is the presence of a single global prediction table. Using our technique, we have obtained tight WCET estimates for benchmark programs under various branch prediction schemes. This is achieved by automatically generating the linear constraints from a program's control flow graph and then solving them using an (Integer) Linear Programming solver.

There are several avenues for future research on this

Benchmark	Actual	Estimated	Ratio
check	198	198	1.00
matsum	200	200	1.00
matmul	200	200	1.00
sort	399	399	1.00
eqnt	203	204	1.00
FFT	4129	5154	1.25

Table 4: Actual and estimated branch misprediction count with *Local*.

Bench	Pess.	Opt.	Actual	Est.	Ratio
check	1202	602	1196	1196	1.00
matsum	131105	100805	101405	101405	1.00
matmul	18396	15066	15666	15666	1.00
sort	75550	45250	46447	46447	1.00
eqnt	3007	1804	2311	2314	1.00
FFT	237653	203840	216227	219302	1.01

Table 5: Effects of *Local* branch prediction scheme on WCET. All the numbers are given in terms of processor cycles. *Pessimistic* is WCET with 100% misprediction, *Optimistic* with no misprediction. *Actual* is the correct WCET and *Estimated* is the WCET with our analysis.

Benchmark	*GAg* Time (msec)	*Local* Time (msec)
check	5	4
matsum	5	4
matmul	6	4
insertion	19	5
eqnt	26	7
FFT	22	8

Table 6: Time to solve the linear programming problem for *GAg* and *Local* branch prediction schemes.

topic. First of all our technique assumes that the loop bounds of the program are computed offline or provided by the user. However, techniques for estimating loop bounds, such as [6], are not applicable to all programs. Therefore, in future we plan to integrate the loop bound estimation into our framework for analyzing WCET under branch prediction. This can significantly reduce the pessimism in our estimated WCET for benchmarks such as *FFT*. Work in the direction of such an integrated analysis has been reported in [14, 15].

In the area of micro-architectural modeling, we plan to study other branch prediction schemes such as *PAg* [22]. 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, an important direction of our future work will be to use our WCET analysis for code optimization. Conditional branch instructions can severely affect the performance of a processor pipeline in practice. For this reason, recent research has been directed towards developing semantics preserving transformations for reordering branch instructions [21] or replacing branches with predicated execution [9]. In our analysis technique, the ILP solver not only returns the WCET, but also bounds on the misprediction count of every basic block (and other relevant information). It will be interesting to see whether code transformations can utilize this information to reduce the actual WCET of a program.

References

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

[2] M. R. C. M. Berkelaar. *lp_solve: (mixed integer) linear programming problem solver.* Available from `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

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

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

[6] C. Healy et al. Bounding loop iterations for timing analysis. In *Proceedings of IEEE Real-time Appplications Symposium (RTAS)*, 1998.

[7] C. Healy et al. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), 1999.

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

[9] R. Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms and Tools.* Kluwer Academic Publishers, 2000.

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

[11] 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.

[12] S-S. Lim et al. An accurate worst-case timing analysis technique for RISC processors. *IEEE Transactions on Software Engineering*, 21(7), 1995.

[13] 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.*

[14] Y.A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the International Workshop on Languages, Compilers and Tools for Embedded System (LCTES), LNCS 1474*, 1998.

[15] T. Lundqvist and P. Stenstrom. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of the International Workshop on Languages, Compilers and Tools for Embedded System (LCTES), LNCS 1474*, 1998.

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

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

[18] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded System (LCTES)*, 1999.

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

[20] 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.

[21] M. Yang, G-R. Uh, and D. Whalley. Improving performance by branch reordering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[22] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 1992.