Modeling Control Speculation for Timing Analysis *

Xianfeng Li, Tulika Mitra† and Abhik Roychoudhury
School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543.

Abstract. The schedulability analysis of real-time embedded systems requires Worst Case Execution Time (WCET) analysis for the individual tasks. Bounding WCET involves not only language-level program path analysis, but also modeling the performance impact of complex micro-architectural features present in modern processors. In this paper, we statically analyze the execution time of embedded software on processors with speculative execution. The speculation of conditional branch outcomes (branch prediction) significantly improves a program's execution time. Thus, accurate modeling of control speculation is important for calculating tight WCET estimates. We present a parameterized framework to model the different branch prediction schemes. We further consider the complex interaction between speculative execution and instruction cache performance, that is, the fact that speculatively executed blocks can generate additional cache hits/misses. We extend our modeling to capture this effect of branch prediction on cache performance. Starting with the control flow graph of a program, our technique uses integer linear programming to estimate the program's WCET. The accuracy of our method is demonstrated by tight estimates obtained on realistic benchmarks.

Keywords.
Schedulability Analysis, Worst Case Execution Time, Micro-architectural modeling, branch prediction, instruction cache

1. INTRODUCTION

A real-time embedded system contains at least one processor running application-specific programs that communicate with the external environment in a timely fashion. There are hard deadlines on the execution time of such software. Moreover, many embedded systems are safety critical (e.g., automobiles and power plant applications), and the designers of such systems *must* ensure that all the real-time constraints are satisfied. This requires us to estimate the Worst Case Execution Time (*WCET*) of the software on the particular processor. One possibility of computing the WCET is to actually run the program and measure its performance on the processor in question. However, this approach is infeasible for most programs due to the large number of program paths corresponding to the different inputs. Consequently, WCET analysis,

* Preliminary version of parts of this paper have previously been published as [21] and [14].

† Contact Author. E-mail: tulika@comp.nus.edu.sg

which statically analyzes a piece of code to produce an upper bound on the execution time, has received much attention over the last decade [24, 26, 17, 9, 18, 16, 25, 5]. The *estimated WCET* produced by static analysis should be safe (i.e., an overestimation of the actual WCET) but tight (as close to the actual WCET as possible).

WCET analysis comprises two parts: (a) *program path analysis* to eliminate some of the infeasible paths, and (b) *micro-architectural modeling* of different processor features such as pipeline, caches, branch prediction, etc. to estimate the execution time of a given path. Micro-architectural modeling is essential for timing analysis because hardware features (such as cache and pipeline) can alter an instruction's execution time (e.g., an instruction that misses in the cache takes much longer to execute). The effects of cache and pipeline on a program's WCET have been studied extensively in the past decade [30, 1, 17, 9, 16, 25, 27].

Apart from cache and pipeline, current embedded processors also use aggressive control speculation to improve performance. The presence of branch instructions forms control dependency between different parts of the code. This dependency causes pipeline stalls, which can be avoided by speculating the control flow subsequent to a branch instruction. Control flow speculation is done through branch prediction of the outcome of a branch instruction [12]. If the prediction is correct, then the execution proceeds without any interruption. If the prediction is incorrect, the speculatively executed instructions are undone, incurring a branch misprediction penalty. We provide a background on branch prediction schemes in the next section. Apart from misprediction penalties, branch prediction also exerts indirect effects on the performance of other micro-architectural features, such as instruction cache. As the processor caches instructions along the mispredicted path, the instruction cache content is modified by the time the branch is resolved. This prefetching of instructions can have both constructive and destructive effects on cache performance and hence on WCET.

Clearly, we cannot assume perfect branch prediction for the purposes of WCET analysis. This assumption may result in an incorrect WCET (i.e., lower than the actual value), particularly, when a hard-to-predict conditional statement (if-then-else) is present inside a loop body and contributes substantially to a program's WCET. Alternatively, certain works assume that all branches in a program are mispredicted. This pessimism results in the significant overestimation of the WCET as branch prediction accuracy is quite high for loop control branches.

In this paper, we propose an Integer Linear Programming (ILP) based framework to estimate WCET by taking into account speculative execution as well as its interaction with the instruction cache. The ILP formulation obtains highly accurate WCET estimates since it combines

program path analysis and micro-architectural modeling into a single framework [15, 16]. Our modeling of branch prediction is generic and parameterizable w.r.t. the currently used branch prediction schemes. Effects of branch misprediction on cache performance are integrated into our framework by extending previous work on instruction cache modeling [16]. Based on the branch prediction scheme and cache organization, our modeling derives linear constraints from the control flow graph of a program. These constraints are fed to an ILP solver for computing an upper bound on the program's execution time. We provide experimental results on the accuracy of our modeling for moderate to large-sized embedded benchmarks.

Micro-architectural modeling for WCET analysis is typically achieved through either Abstract Interpretation (AI) [9] or Integer Linear Programming (ILP) [16]. To model a micro-architectural feature, AI based schemes develop a static categorization of the program instructions. For example, to model instruction cache, AI based schemes categorize program instructions as *always hit*, *always miss* etc., which is clearly conservative. ILP based works seek to remove this inaccuracy by developing constraints to bound the worst case execution time. Thus, for instruction cache modeling, an ILP based approach will develop linear constraints to bound the maximum number of misses for each instruction.

Our modeling of branch prediction is based on Integer Linear Programming. Modeling several micro-architectural features is easy in the abstract interpretation based approach, since this is achieved by considering the effect of each feature separately (in other words, effects of the interaction between different features is overestimated). However, the resultant modeling is less accurate. In the ILP based approach, modeling several interacting micro-architectural features is non-trivial since we have to modify the constraints modeling the individual features. In this paper, we develop an ILP based modeling of two interacting micro-architectural features: branch prediction and instruction cache.

Contributions The contributions of this paper can be summarized as follows.

- We model various branch prediction schemes for WCET analysis including widely used schemes such as *gshare* [20, 29]. Modeling branch prediction for timing analysis is a largely unexplored topic. Existing works such as [5] only model simple branch predictors in a manner similar to instruction cache modeling. We discuss the difficulties of modeling more complex branch predictors used in modern processors. We also present a parameterized framework which models various existing branch prediction schemes.

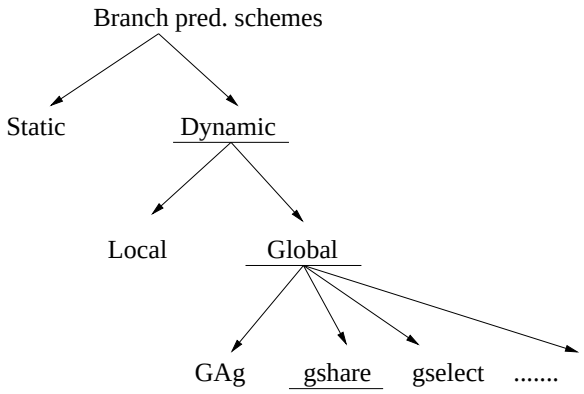

Figure 1. Categorization of Branch Prediction Schemes

- We study the interaction of branch prediction with instruction cache behavior, and integrate our modeling of branch prediction with existing modeling of instruction cache [16]. To the best of our knowledge, ours is the only work on ILP based WCET analysis that models several micro-architectural features and their interaction.
- We present experimental results to demonstrate the accuracy of our modeling and its impact on WCET estimation. We also discuss evidence of the scalability of our estimation technique w.r.t. (a) different branch prediction schemes, (b) cache organizations, and (c) different values of branch misprediction and cache miss penalties.

Paper Organization The remainder of this paper is organized as follows. The next section provides a brief overview of the existing branch prediction schemes in modern processors. Section 3 discusses related work on WCET analysis. Section 4 presents our modeling of branch prediction; an example is discussed at the end of the section. In Section 5, we examine the combined effects of branch prediction and instruction cache on WCET and integrate them into our modeling. In Section 6, we present the experimental results on the accuracy, retargetability and scalability of our analysis technique. We conclude in Section 7.

2. BACKGROUND ON BRANCH PREDICTION

Branch prediction schemes can be broadly categorized as *static* and *dynamic* (see Figure 1; the most popular category in each level is

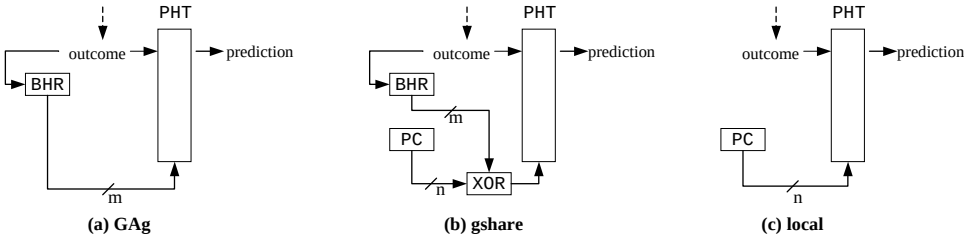

Figure 2. Illustration of Branch Prediction Schemes. The branch prediction table is shown as PHT, denoting Pattern History Table.

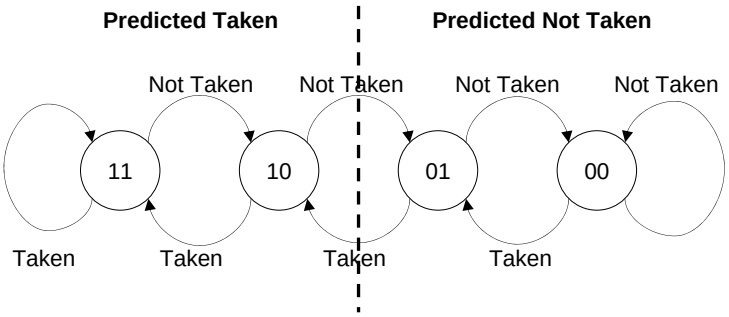

Figure 3. Two-bit Saturating Counter Predictor

underlined). In the 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 perform the prediction 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* (illustrated in Figure 2(c)), where each branch is predicted based on its last few outcomes. It is called "local" because the prediction of a branch is *only* dependent on its *own* history. This scheme uses a 2^n-entry *branch prediction table* to store past branch outcomes, which is indexed by the n lower order bits of the branch address. Obviously, two or more branches with the same lower order address bits will map to the same table entry and they will affect each other's predictions (constructively or destructively). This is known as the *aliasing effect*. In the simplest case, each prediction table entry is one-bit and stores the last outcome of the branch mapped to that entry.

Throughout this paper, for simplicity of disposition, we only discuss our modeling for the one-bit scheme. 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. In practice, two-bit saturating counters are often used for prediction, as show in Figure 3; furthermore, the two-bit counter can be extended to n-bit scheme straightforwardly. We are aware that subsequent to our work, Bate and Reutemann [2] have developed techniques to extend the state-of-the art for modeling an n-bit saturating counter (in each row of the prediction table).

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 [29]. Global schemes use a single shift register called *branch history register (BHR)* to record the outcomes of the n most recent branches. As in local schemes, there is a branch prediction table in which 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 [20]. In the *GAg* scheme (refer to Figure 2(a)), the BHR is simply used as an index to look up the prediction table. In the popular *gshare* scheme (refer to Figure 2(b)), the BHR is XOR-ed with the last n bits of the branch address (the PC register in Figure 2(b)) for prediction table look-up. Usually, *gshare* results in a more uniform distribution of table indices compared to *GAg*. Finally, in the *gselect (GAp)* scheme (not illustrated in Figure 2 but straightforward to derive from the gshare scheme), the BHR is concatenated with the last few bits of the branch address to look up the table. Our technique presented in section 4.3 can model each of these schemes simply with an adjustment of the parameters of the analyzer.

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 easily modeled via techniques similar to instruction cache analysis [16]. 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 is available at the beginning of the pipeline whereas the branch outcome is available near the end of the pipeline.

3. RELATED WORK ON WCET ANALYSIS

Research on WCET analysis was initiated more than a decade ago. Early research activities can be traced back to Shaw and Park's *timing schema* [22, 26]. They analyzed the program source code and did not consider hardware speed-up features such as caching or pipelining. Their analysis of program source code is compositional; thus, to estimate the worst case execution time of a loop, the method first finds out the worst case execution time of *any* loop iteration. Needless to say that ignoring hardware modeling results in loose bounds. Puschner and Koza [24] studied the conditions for decidability of WCET analysis: availability of loop bounds, absence of dynamic function calls, etc.

As far as research on micro-architectural modeling is concerned, most existing work focuses on modeling *instruction cache* and *pipeline*, either individually or combined. Zhang et al. [30] modeled a simple pipeline structure which has only two stages. Schneider and Ferdinand [25] used *abstract interpretation* to examine a more sophisticated pipeline of the SuperSPARC I processor.

Modeling of instruction cache behavior for WCET analysis has received considerable attention [9, 16, 28]. The difficulty in analyzing cache behavior is its global nature (as compared to the timing effects of a pipeline). One instruction may cause another (spatially or temporally) remote instruction to miss/hit in the cache. Arnold et al. [1] modeled caching by first doing simulation on cache behavior and then categorizing cache accesses as *always hit*, *always miss*, *first hit* or *first miss*. After the category information is obtained, the execution time of each instruction (or basic block) of the program is determined. Theiling et al. [27] used abstract interpretation to perform analysis on cache behavior. Their approach shares the two-phase strategy of Arnold et al. [1]. The difference is that they used abstract interpretation (instead of cache simulation) to obtain the categorization of instructions.

The combined effects of caching and pipelining on timing analysis have been studied extensively in the last decade. Lim et al. [17] extended the timing schema approach while Healy et al. [10] incorporated pipeline modeling into their previous work of cache modeling [1]. They first categorized cache behaviors of instructions, and used the cache information to analyze the performance of the pipeline. Thus, instruction timing is fixed so that path analysis can be performed as the last step. Lundqvist and Stenström [19] performed WCET analysis based on *symbolic execution*, which can actually be deemed as a hybrid of simulation and static analysis.

Li et al. [15, 16] proposed another integrated method in the context of modeling instruction cache behavior for WCET analysis. They

used an *integer linear programming* (ILP) formulation and did the path enumeration implicitly with linear constraints derived from the *control flow graph* (CFG) of the program. The cache behavior is also modeled by linear constraints from another set of graphs called *cache conflict graphs* (CCG). The two sets of constraints are integrated and submitted to an ILP solver. One advantage of this approach is that it solves the following dilemma: we cannot decide the worst case path without knowing individual instructions' execution time; on the other hand, an instruction's execution time might be unknown until its execution context/path is determined. By modeling the problem in terms of constraints and then letting an ILP solver explore the possible paths and corresponding instruction timing simultaneously, this dilemma can be solved. However, in the separated approach [27], as the execution paths for instructions are unknown, many instructions which can be either hits or misses in the cache are conservatively categorized as cache misses, resulting in overestimation.

Papers investigating branch prediction effects have emerged in recent years. Engblom [8] performed a purely empirical study; his work shows the importance of modeling branch prediction for timing analysis. However, the work offers no formal analysis technique. Colin and Puaut [5] presented their work on modeling a *local* branch prediction scheme. Their work is basically a separated approach (as are the works on abstract interpretation based micro-architectural modeling). Branch instructions are first classified into four classes, a strategy similar to some of the modeling methods for caching [9]. Note that they considered 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 popular global prediction schemes like gshare [20, 29]. In these schemes, a branch instruction's prediction may reside in different rows of the prediction table at different points in execution.

The impact of branch prediction on instruction cache contents is often called the wrong-path cache effect. The wrong-path cache effect involves prefetching instructions due to misprediction. Note that instruction prefetching has previously been modeled for WCET analysis [4, 13]. However, wrong-path prefetch modeling is much more involved as the prefetching is controlled by branch mispredictions (which by themselves are difficult to capture).

4. MODELING BRANCH PREDICTION

In this section, we discuss the modeling of branch prediction schemes for WCET analysis. Section 4.1 shows the core of our modeling using Integer Linear Programming and Section 4.2 illustrates our modeling with an example program. In Section 4.3, we demonstrate the flexibility of our modeling technique by capturing the effects of various existing branch prediction schemes.

4.1. Core Modeling

Issues in modeling branch prediction We proceed to examine the difficulties in modeling branch prediction for worst case execution time analysis. So far, micro-architectural features such as pipelining and instruction caching have been modeled for WCET analysis. In the presence of these features, the execution time of an instruction may depend on the past execution trace. For pipelining, 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 instruction caching and branch prediction, *global analysis* is required. This is because the effect of an instruction's execution on caches and branch predictors could affect the execution of remote instructions. However, there are two significant differences between the global analysis of the instruction caching and of branch prediction.

Both instruction caching and branch prediction maintain global data structures that record information about the past execution trace, namely the cache and the branch prediction table. For instruction caching, 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[1]. Local branch prediction is quite similar – outcomes of a given branch instruction are stored only in one fixed entry of the prediction table where predictions are made. 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 the last few branches before arriving at I) to decide the prediction table entry. Because it is possible to arrive at I with various histories, the prediction for I can use different entries of the prediction table at different points of execution.

The other difference between instruction caching and branch prediction modeling is obvious. In the case of instruction caching, if two

[1] To be precise, in associative caches, an address can be present in only one cache *set*.

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 the 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'.

For ease of description, we model the *GAg* global branch prediction scheme as an example. This scheme has been described in Section 2. However, our modeling is generic and not restricted to *GAg* (as will be shown in Section 4.3). In fact, the default scheme in our experiments is the more popular *gshare* scheme.

Control Flow Graph (CFG) The starting point of our analysis is the control flow graph of the program. The vertices of this graph are the basic blocks, and an edge $i \rightarrow j$ denotes the flow of control from basic block B_i to basic block B_j. We assume that the CFG 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 CFG has a label, denoted $label(i \rightarrow j)$.

$$\begin{aligned} label(i \rightarrow j) = \; & U \text{ if } i \rightarrow j \text{ implies unconditional flow} \\ & 1 \text{ if } i \rightarrow j \text{ implies branch at i is } \textit{taken} \\ & 0 \text{ if } i \rightarrow j \text{ implies branch at i is } \textit{non-taken} \end{aligned}$$

For any block B_i, if its last instruction is a conditional branch, then it has two outgoing edges labeled 0 and 1. Otherwise, B_i has one outgoing edge with label U.

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

Flow constraints and loop bounds Let v_i denote the number of times B_i is executed, and let $e_{i \rightarrow j}$ denote the number of times control flows through the edge $i \rightarrow j$. As inflow equals outflow for each basic block (except the start and end nodes), we have the following equations:

$$v_i = \sum_j e_{j \rightarrow i} = \sum_j e_{i \rightarrow j}$$

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

$$v_{start} = v_{end} = 1 = \sum_i e_{start \to i} = \sum_i e_{i \to end}$$

Upper bounds on v_i are provided through the maximum number of iterations for loops and maximum depth of invocations for recursive procedures. These bounds can be user provided, or can be computed offline for certain programs [11].

Defining WCET Let $cost_i$ be the execution time of B_i, assuming perfect branch prediction and no cache misses. Given the program, $cost_i$ is a fixed constant for each i. Then, the total execution time of the program is:

$$Time = \sum_{i=1}^{N} (cost_i * v_i + bmp * bm_i)$$

where N is the number of basic blocks in the program, bmp is a constant denoting the penalty for a single branch misprediction, and bm_i is the number of times the branch in B_i is mispredicted. If B_i does not contain a branch, then $bm_i = 0$. To find the worst case execution time, we need to maximize the above objective function. For this purpose, we need to derive constraints on bm_i.

Introducing History Patterns To predict the direction of the branch in B_i, first, the index into the prediction table is computed. In the case of *GAg*, this index is the outcome of the last k branches before B_i is executed and recorded in the Branch History Register (BHR) with k bits. Thus, if $k = 2$ and the last two branches are taken (1) followed by not taken (0), then the index will be 10. We define annotated execution counts and misprediction counts v_i^π and bm_i^π, corresponding to the execution of B_i with BHR $= \pi$ when B_i is reached. Similarly, $e_{i \to j}^\pi$ denotes the number of times the edge $e_{i \to j}$ is passed with $BHR = \pi$ at the beginning of basic block B_i. Thus,

$$bm_i^\pi \leq v_i^\pi; \;\; e_{i \to j} = \textstyle\sum_\pi e_{i \to j}^\pi; \;\; bm_i = \textstyle\sum_\pi bm_i^\pi; \;\; v_i = \textstyle\sum_\pi v_i^\pi.$$

For each B_i and history π, we find out whether it is possible to reach B_i with history π. This information can be obtained via a terminating least fixed point analysis on the control flow graph. Clearly, if it is not possible to reach B_i with π, then $e_{i \to j}^\pi = v_i^\pi = bm_i^\pi = 0$.

Control flow among history patterns To provide an upper bound on bm_i^π, we first define constraints on v_i^π (since $bm_i^\pi \leq v_i^\pi$). This is done by modeling the change in history along the control flow graph.

DEFINITION 1.

Let π be a history pattern with k bits (the width of the Branch History Register) at B_i. It is composed of the sequence of outcomes of the most recent k branches with the latest outcome at the rightmost bit. The change in history pattern along $i \rightarrow j$ is given by:

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

where $left(\pi, 0)$ ($left(\pi, 1)$) shifts pattern π to the left by one bit (the old leftmost bit is therefore discarded) and puts 0 (1) as the rightmost bit.

Now, B_i can execute with history π only if there exists B_j executing with history π' such that $\Gamma(\pi', j \rightarrow i) = \pi$. Note that for any such incoming edge $j \rightarrow i$, there can be two history patterns π' such that $\Gamma(\pi', j \rightarrow i) = \pi$. For example, if $label(j \rightarrow i) = 1$, then $\Gamma(011, j \rightarrow i) = \Gamma(111, j \rightarrow i) = 111$. Therefore, from the inflows of B_i's execution with history π we get:

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

Similarly, from the outflows of B_i's execution with history π, we get:

$$v_i^{\pi} = \sum_{j} e_{i \rightarrow j}^{\pi}$$

Repetition of a history pattern Let us assume a misprediction of the branch in B_i with history π. This means that certain blocks (perhaps B_i itself) were executed with history π such that the outcomes of these branches created a prediction different from the current outcome of B_i. Thus, to model mispredictions, we need to capture repeated occurrences of a history π during the program's execution. For this purpose, we define $p_{i \rightsquigarrow j}^{\pi}$.

DEFINITION 2. *Let B_i and B_j be two basic blocks with branch instructions and π be a history pattern. Then $p_{i \rightsquigarrow j}^{\pi}$ is the number of times a path is taken from B_i to B_j s.t.*

- *π never occurs at a node with a branch instruction between B_i and B_j.*

– *If $B_i \neq$ start block, then π occurs at B_i*

– *If $B_j \neq$ end block, then π occurs at B_j*

Intuitively, $p^{\pi}_{i \rightsquigarrow j}$ denotes the number of times control flows from B_i to B_j s.t. the πth row of the prediction table is only used for branch prediction at B_i and B_j, and is never accessed in between. In these scenarios, the outcome of B_i may cause a misprediction at B_j. Furthermore, $p^{\pi}_{start \rightsquigarrow i}$ ($p^{\pi}_{i \rightsquigarrow end}$) models the number of times the π th row of the prediction table is looked up for the first (last) time at B_i.

When the πth row is used for branch prediction at B_i, either the πth row is used for the first time (denoted by $p^{\pi}_{start \rightsquigarrow i}$) or the πth row was used for branch prediction last time in some block $B_j \neq B_{start}$. Similarly, for every use of the πth row of the prediction table at B_i, either it is the last use (denoted by $p^{\pi}_{i \rightsquigarrow end}$) or it will be used the next time in $B_j \neq B_{end}$. Since v^{π}_i denotes the number of times B_i uses the πth row of the prediction table, we have:

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

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:

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

Introducing branch outcomes To model mispredictions, we not only need to model the repetition of history patterns, but also branch outcomes. A misprediction occurs on differing branch outcomes for the same history pattern. Therefore, we partition the paths contributing to the count $p^{\pi}_{i \rightsquigarrow j}$ based on the branch outcome at B_i: $p^{\pi,1}_{i \rightsquigarrow j}$ and $p^{\pi,0}_{i \rightsquigarrow j}$, which denote the execution count of those paths that begin with the outgoing edge of B_i labeled 1 (i.e., outcome 1) and 0, respectively. By definition:

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

$$\textstyle\sum_j p^{\pi,1}_{i \rightsquigarrow j} = e^{\pi}_{i \rightarrow k} \quad \text{and} \quad \sum_j p^{\pi,0}_{i \rightsquigarrow j} = e^{\pi}_{i \rightarrow l}$$

where $label(i \rightarrow k) = 1$ and $label(i \rightarrow l) = 0$. In other words, $i \rightarrow l$ and $i \rightarrow k$ are the outgoing edges of basic block B_i with labels 0 and 1, respectively.

Modeling mispredictions For simplicity of exposition, let us assume that each row of the prediction table contains a one-bit prediction: 0

denotes a prediction that the branch will not be taken, and 1 denotes a prediction that the branch will be taken. However, our technique for estimating mispredictions is generic. It can be extended if the prediction table maintains more than one bit per entry. In particular, a recent work [2] has modeled a n-bit saturating counter (in each row of the prediction table).

Recall that bm_i^π denotes the number of mispredictions of the branch in B_i when it is executed with history pattern π. There can be two scenarios in which B_i is mispredicted with history π:

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

From the above, we derive the following bound on bm_i^π:

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

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

Putting it all together We have derived linear inequalities on v_i (execution count of B_i) and bm_i (misprediction count of B_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.

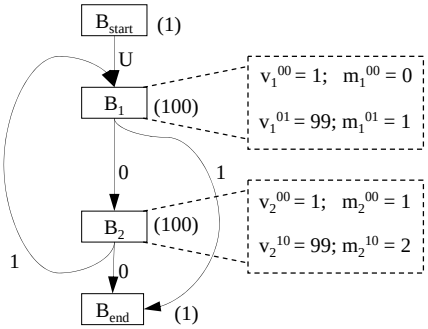

Figure 4. Example of the Control Flow Graph

4.2. An Example

In this part, we illustrate our WCET estimation technique with a simple example. Consider the control flow graph in Figure 4. The start and end blocks are called B_{start} and B_{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 two-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.

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

$$v_{start} = v_{end} = 1 = e_{start \to 1} = e_{2 \to end} + e_{1 \to end}$$

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

$$v_1 = e_{start \to 1} + e_{2 \to 1} = e_{1 \to 2} + e_{1 \to end}$$

$$v_2 = e_{1 \to 2} = e_{2 \to end} + e_{2 \to 1}$$

Furthermore, the edge $2 \to 1$ is a loop, and its bound must be given. In our method, this bound is either computed off-line or provided by the user. Let us consider a loop bound of 100. Then,

$$e_{2 \to 1} < 100$$

Defining WCET Let us assume a branch misprediction penalty of three clock cycles. The WCET of the program is obtained by maximiz-

ing:

$$Time = 2v_{start} + 2v_1 + 4v_2 + 2v_{end} + 3bm_1 + 3bm_2$$

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); bm_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 B_i via static analysis of the control flow graph. The initial history at the beginning of program execution is assumed to be 00. In our example, the possible history patterns for the different basic blocks are as follows:

$$\begin{array}{ll} B_{start}: & \{00\} \\ B_1: & \{00, 01\} \\ B_2: & \{00, 10\} \\ B_{end}: & \{00, 01, 11\} \end{array}$$

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

$$\begin{array}{ll} v_{start} = v_{start}^{00} = 1 & bm_{start} = 0 \\ v_1 = v_1^{00} + v_1^{01} & bm_1 = bm_1^{00} + bm_1^{01} \\ v_2 = v_2^{00} + v_2^{10} & bm_2 = bm_2^{00} + bm_2^{10} \\ v_{end} = v_{end}^{00} + v_{end}^{01} + v_{end}^{11} = 1 & bm_{end} = 0 \end{array}$$

$$\begin{array}{ll} bm_1^{00} \leq v_1^{00} & bm_1^{01} \leq v_1^{01} \\ bm_2^{00} \leq v_2^{00} & bm_2^{10} \leq v_2^{10} \end{array}$$

We also define variables of the form $e_{i\to j}^\pi$ as follows (by using the set of patterns possible at each basic block):

$$\begin{array}{ll} e_{start\to 1} = e_{start\to 1}^{00} & \\ e_{1\to 2} = e_{1\to 2}^{00} + e_{1\to 2}^{01} & e_{1\to end} = e_{1\to end}^{00} + e_{1\to end}^{01} \\ e_{2\to 1} = e_{2\to 1}^{00} + e_{2\to 1}^{10} & e_{2\to end} = e_{2\to end}^{00} + e_{2\to end}^{10} \end{array}$$

Control flow among history patterns We now derive the constraints on v_i^π based on the flow of the pattern π. Let us consider the inflows and outflows of block 1 with history 01. From the inflows we get:

$$v_1^{01} = e_{2\to 1}^{00} + e_{2\to 1}^{10}$$

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:

$$v_1^{01} = e_{1\rightarrow 2}^{01} + e_{1\rightarrow end}^{01}$$

Constraints for inflows/outflows of block 1 with history 00, block 2 with history 00, and block 2 with history 10 are derived similarly.

Repetition of history pattern To model the repetition of a history pattern along a program path, variables $p_{i\rightsquigarrow j}^{\pi}$ are introduced (refer to 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:

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

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

Similarly, we provide constraints for the other patterns.

Introducing branch outcomes For each $p_{i\rightsquigarrow j}^{\pi}$, we define the variables $p_{i\rightsquigarrow j}^{\pi,0}$ and $p_{i\rightsquigarrow j}^{\pi,1}$ via the equation $p_{i\rightsquigarrow j}^{\pi} = p_{i\rightsquigarrow j}^{\pi,0} + p_{i\rightsquigarrow j}^{\pi,1}$. More importantly, we relate $p_{i\rightsquigarrow j}^{\pi}$ variables to $e_{i\rightarrow j}^{\pi}$ variables via $p_{i\rightsquigarrow j}^{\pi,0}$ and $p_{i\rightsquigarrow j}^{\pi,1}$. For example we have $p_{2\rightsquigarrow 2}^{10,1} + p_{2\rightsquigarrow end}^{10,1} = e_{2\rightarrow 1}^{10}$ in Figure 4. In our simple example, we only derive trivial constraints in this category. In general, a sum of $p_{i\rightsquigarrow j}^{\pi,1}$ (or $p_{i\rightsquigarrow j}^{\pi,0}$) variables equals an $e_{i\rightarrow j}^{\pi}$ variable.

Modeling mispredictions Let us now derive the constraints for bm_1^{01}, the number of mispredictions of block 1 with history 01. For this, we consider two cases corresponding to the outcome of the branch at block 1.

– Case 1: The branch at block 1 is taken, and the last branch using the 01 row of the predictor table is not taken.

  The number of times the branch at block 1 is taken is $p^{01,1}_{1\rightsquigarrow end}$. Recall that this is the number of times the control flows as follows: *(a)* block 1 is executed with history 01, *(b)* the branch at block 1 is taken *i.e.*, an edge labeled 1 is then executed, and *(c)* control then flows to the *end* block without history 01 ever occurring.

  The number of times the last branch (before arriving at block 1) using the 01 row of the predictor table is not taken is $p^{01,0}_{start\rightsquigarrow 1} + p^{01,0}_{1\rightsquigarrow 1}$. Note that the other block (block 2) is not considered since block 2 cannot be reached with pattern 01.

  Thus, this case happens $bm^{01,1}_1$ times where

$$bm^{01,1}_1 \leq min(p^{01,1}_{1\rightsquigarrow end},\ p^{01,0}_{start\rightsquigarrow 1} + p^{01,0}_{1\rightsquigarrow 1})$$

– Case 2: The branch at block 1 is not taken, and the last branch using the 01 row of the predictor table is taken. This happens $bm^{01,0}_1$ times where:

$$bm^{01,0}_1 \leq min(p^{01,0}_{1\rightsquigarrow 1} + p^{01,0}_{1\rightsquigarrow end},\ 0) = 0$$

  Note that 0 appears in above formula as in this particular example, no earlier branch using the 01 row of the predictor table with outcome taken can reach block 1.

Since the two cases are mutually exclusive and bm^{01}_1 counts both of the above cases, we have:

$$bm^{01}_1 = bm^{01,1}_1 + bm^{01,0}_1$$

Other misprediction constraints are:

$$\begin{aligned}
bm^{00}_1 \leq\ & min(p^{00,1}_{1\rightsquigarrow end}\ ,\ p^{00,0}_{start\rightsquigarrow 1}) && + && min(p^{00,0}_{1\rightsquigarrow 2}\ ,\ 0)\\
bm^{00}_2 \leq\ & min(p^{00,1}_{2\rightsquigarrow end}\ ,\ p^{00,0}_{1\rightsquigarrow 2}) && + && min(p^{00,0}_{2\rightsquigarrow end}\ ,\ 0)\\
bm^{10}_2 \leq\ & min(p^{10,1}_{2\rightsquigarrow 2} + p^{10,1}_{2\rightsquigarrow end}\ ,\ p^{10,0}_{start\rightsquigarrow 2}) && + && min(p^{10,0}_{2\rightsquigarrow end}\ ,\ p^{10,1}_{2\rightsquigarrow 2})
\end{aligned}$$

They correspond to the constraints on bm^{π}_i shown in the last subsection. Maximizing the objective function w.r.t. all these constraints gives the program's WCET.

The execution counts of basic blocks as well as their misprediction counts computed by the ILP solver are given in Figure 4.

4.3. Modeling various prediction schemes

We now discuss how our modeling can be used to capture the effects of various local and global branch prediction schemes. Our modeling of branch prediction is independent of the definition of the prediction table index, so far called 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) the address of the branch instruction I. In the *GAg* scheme, the index computed depends solely on the history and not the branch instruction address. Other global prediction schemes (*gshare*, *gselect*) use both the history and the 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). This of course affects the possible prediction table indices that can be looked up at a basic block B_i. *No change is made to the linear constraints* (parameterized w.r.t. possible prediction table indices at each basic block) described in the previous subsection. 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* [20, 29]. In *gshare*, 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 XOR-ed with higher-order m bits of $address_n(I)$). The updating of π due to control flow is modeled by the function:

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

where $i \rightarrow j$ is an edge in the control flow graph, $address_n(j)$ is the least significant n bits of the branch instruction in basic block j, and Γ is the function on the 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:

$$\pi = history_m \bullet address_n(j)$$

where m and n are some constants and $\bullet$ denotes concatenation. The updating of π due to control flow is given by function $\Gamma_{gselect}$

$$\Gamma_{gselect}(\pi, i \rightarrow j) = \Gamma(history_m, i \rightarrow j) \bullet address_n(j)$$

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

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 n bits of the address of branch instruction I.

Updating of the index π due to control flow is given by $\Gamma_{local}(\pi, i \rightarrow j) = address_n(j)$. Here, $i \rightarrow j$ is an edge in the control flow graph and $address_n(j)$ is the least significant n bits of the last instruction in basic block j. If block 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. Clearly, since each block j always uses the same index π into the prediction table, index π is used at basic block j if and only if π denotes the least significant n bits of the address of the branch instruction of block j (if any).

5. INTEGRATED MODELING OF CACHE AND BRANCH PREDICTION

In processors with both cache and control speculation mechanism, we need to take their interaction into account. This interaction is in fact unidirectional: speculative execution can alter the behavior of instruction caching. Consider a branch that is mispredicted. The processor will fetch and execute instructions along the wrong path till the branch is resolved. We have modeled processors with both instruction caching and branch prediction elsewhere [14]. We present this combined modeling in this section. First we review the modeling of instruction caching for WCET analysis, as formulated originally by Li et al. [16].

5.1. Pure Instruction Cache Modeling

We recapitulate the earlier instruction cache modeling [16]. A basic block B_i is partitioned into n_i l-blocks[2] denoted as $B_{i.1}, B_{i.2}, \ldots, B_{i.n_i}$. Let $cm_{i.j}$ be the total cache misses for l-block $B_{i.j}$ and cmp be the constant denoting the cache miss penalty. Then, the total execution time is:

$$Time = \sum_{i=1}^{N} (cost_i \times v_i + bmp \times bm_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) \qquad (1)$$

For simplicity of exposition, let us assume a direct mapped cache; the modeling can be easily extended to set-associative caches. For each cache line c, we construct a *Cache Conflict Graph (CCG)* G_c [16]. The nodes of G_c are the l-blocks mapped to c. An edge $B_{i.j} \rightsquigarrow B_{u.v}$ exists in G_c iff there exists a path in the CFG s.t. control flows from $B_{i.j}$ to $B_{u.v}$ without going through any other l-block mapped to c. In other words, there is an edge between l-blocks $B_{i.j}$ to $B_{u.v}$ if $B_{i.j}$ can be present in the cache when control reaches $B_{u.v}$.

Let $r_{i.j \rightsquigarrow u.v}$ be the execution count of the edge between l-blocks $B_{i.j}$ and $B_{u.v}$ in a CCG. Now, the execution count of l-block $B_{i.j}$ equals the execution count of basic block B_i. Also, at each node of the CCG, the inflow equals the outflow and both equal the execution count of the node. Therefore,

$$v_i = \sum_{u.v} r_{i.j \rightsquigarrow u.v} = \sum_{u.v} r_{u.v \rightsquigarrow i.j} \qquad (2)$$

The cache miss count $cm_{i.j}$ equals the inflow from *conflicting* l-blocks in the CCG (whether two l-blocks are conflicting or non-conflicting is statically determined by portions of their instruction addresses, which are used as tags in cache lines). Thus, we have:

$$cm_{i.j} = \sum_{\substack{u.v \\ B_{u.v}\ conflicts\ B_{i.j}}} r_{u.v \rightsquigarrow i.j} \qquad (3)$$

5.2. Effects of Speculative Execution on Caching

WCET analysis as described in the previous section does not take into account the effect of branch misprediction on instruction cache performance. When a branch is predicted, instructions are fetched and

[2] A line-block, or l-block, is a sequence of instructions in a basic block that belongs to the same instruction cache line.

executed from the predicted path. If all the branches are predicted correctly, then the analysis described in previous section will give accurate results. Now, consider a branch that is mispredicted. The processor will fetch and execute instructions along the mispredicted path till the branch is resolved. There can be two scenarios during mispredicted path execution: (1) there is no cache miss, and (2) there is at least one cache miss. In the first scenario, the misprediction has no effect on the instruction cache. However, in the second scenario, the instruction cache content is modified when the processor resumes execution from the correct path. Various studies have concluded that depending on the application, this wrong-path prefetching can have a constructive or a destructive effect on the instruction cache's performance [6, 23]. Our goal here is to model this wrong-path cache effect for WCET analysis.

We make two standard assumptions. First, we assume that the processor allows only *one unresolved branch* at any point of time during execution. Thus, if another branch is encountered during speculative execution, the processor simply waits till the previous branch is resolved. We also assume that the instruction cache is *blocking* (i.e., it can support only one pending cache miss). This is indeed the case in almost all commercial processors.

We introduce some notations for the subsequent parts. We use $[B_{i.j}]$ to denote the cache line to which l-block $B_{i.j}$ maps. The shorthand $B_{i.j} \cong B_{u.v}$ is used to denote that l-blocks $B_{i.j}$ and $B_{u.v}$ map to the same cache line. Thus $B_{i.j} \cong B_{u.v}$ iff $[B_{i.j}] = [B_{u.v}]$.

The effects of speculation on instruction cache performance can be categorized as follows:

1. An l-block $B_{i.j}$ misses during normal execution since it is displaced by another l-block $B_{u.v} \cong B_{i.j}$ during speculative execution (**destructive effect**).

2. An l-block $B_{i.j}$ hits during normal execution, since it is pre-fetched during speculative execution (**constructive effect**).

3. A pending cache miss of $B_{i.j}$ during speculative execution along the wrong path causes the processor to stall when the branch is resolved. How long the stall lasts depends on the portion of cache miss penalty which is masked by the branch misprediction penalty. If the speculative fetching is completely masked by branch penalty, then there is no delay incurred.

The last situation cannot be simply deemed constructive or destructive, although a delay often happens in that case. The cost of the delay may be offset later by a cache hit to the l-block.

5.3. Changes to Cache Conflict Graph

As the interaction between speculative execution and cache is unidirectional (caching does not influence branch prediction), the branch prediction modeling in Section 4.1 stays unchanged. Moreover, both the constructive and destructive effects of speculative execution on cache are modeled by changing the Cache Conflict Graph (CCG).

Additional nodes in Cache Conflict Graph We add all the l-blocks fetched along the mispredicted path to their respective cache conflict graphs. Given a conditional branch b, its actual outcome X (not taken or taken, denoted as 0 and 1, respectively) and misprediction penalty *bmp* (a constant number of clock cycles), we can identify the set of l-blocks accessed along the mispredicted path, called *Spec(b,X)*. Clearly, the cost of executing the blocks in $Spec(b, X)$ cannot exceed *bmp*. If one or more blocks cause cache misses, then not all the l-blocks in $Spec(b, X)$ can execute. Those l-blocks executed along the mispredicted path are called *ml-blocks* and are annotated with the corresponding basic block containing the branch instruction and the actual outcome. For example, if $B_{i.j} \in Spec(b, X)$, then the corresponding ml-block is denoted by $B_{i.j}^{b,X}$. Note that it is possible to have multiple ml-blocks corresponding to an l-block. For an l-block $B_{i.j}$, all its ml-blocks are added to the CCG of the cache line it maps to.

Additional edges in Cache Conflict Graph We now need to add additional edges in the cache conflict graphs. Given a CCG, we add edges between ml-blocks and the normal l-blocks; we also add edges between ml-blocks. For an ml-block $B_{i.j}^{b,X}$, we add edges to/from all the other l-blocks $B_{u.v}$ in the CCG of cache line $[B_{i.j}]$ and their corresponding ml-blocks as follows:

1. $B_{u.v} \rightsquigarrow B_{i.j}^{b,X}$ if there exists a path from $B_{u.v}$ to $B_{i.j}$ through branch b that does not contain any other l-block mapped to $[B_{i.j}]$. This models the flow from the last normal use of the cache line to the ml-block.

2. $B_{i.j}^{b,X} \rightsquigarrow B_{u.v}^{b,X}$ if $B_{u.v}$ is the next use of the cache line $[B_{i.j}]$ in $Spec(b, X)$ after $B_{i.j}$. This models the flow from the ml-block to the next possible use of the cache line along the mispredicted path.

3. $B_{i.j}^{b,X} \rightsquigarrow B_{u.v}$ if there exists a path from branch b with outcome X to $B_{u.v}$ that does not contain any other l-block mapped to $[B_{i.j}]$.

4. In addition, in case 3, if the path to $B_{u.v}$ goes through branch b' and $B_{u.v} \in Spec(b', Y)$ (b' can be the same as or different from b),

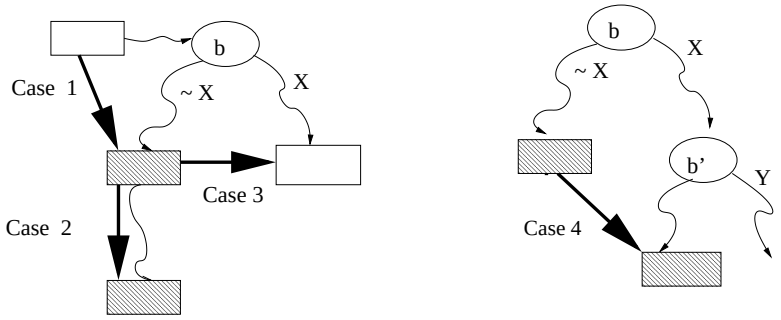

Figure 5. Additional edges in the Cache Conflict Graph due to Speculative Execution. The l-blocks are shown as rectangular boxes, and the ml-blocks among them are shaded.

then we also add $B_{i.j}^{b,X} \rightsquigarrow B_{u.v}^{b',Y}$. The edges in cases 3 and 4 model the flow from the ml-block to the next possible use of the cache line after the branch is resolved.

Figure 5 illustrates these cases. The shaded rectangles are the ml-blocks and the unshaded ones are the normal l-blocks. The third and fourth type of edges require some explanation. If there are multiple l-blocks along the speculative path that map to a particular cache block, then we conservatively add outgoing edges from all of them to the first use of the cache block in the correct path (or another speculative path). This is because any one of these l-blocks can be in the cache when the branch is resolved; exactly which one will be in the cache when the branch is resolved depends on the exact values of *bmp*, *cmp* and the execution time of the individual basic blocks.

Figure 6 illustrates the modifications to the CCG with an example. The control flow graph is shown in Figure 6(a). Let us assume that l-blocks $B_{0.1}$, $B_{1.2}$ and $B_{3.1}$ belong to the same cache block. Then, the original CCG for that cache block is shown in Figure 6(b). A dummy start node and an end node are added to each CCG to make the initial and terminal flow equations correct.

The modifications to the CCG due to wrong-path prefetching is shown in Figure 6(c). We add two ml-block $B_{3.1}^{2,1}$ and $B_{1.2}^{3,0}$ corresponding to the mispredictions at node B_2 and node B_3, respectively. Note that we do not add any node corresponding to a 0 outcome at branch B_2 and a 1 outcome at branch B_3. This is because with a 0 outcome at branch B_2, the mispredicted path fetches basic block B_2 which does not contain any l-block that maps to the cache line, and similarly for B_3 with outcome 1. Among the additional edges, $B_{1.2} \rightsquigarrow B_{3.1}^{2,1}$ and

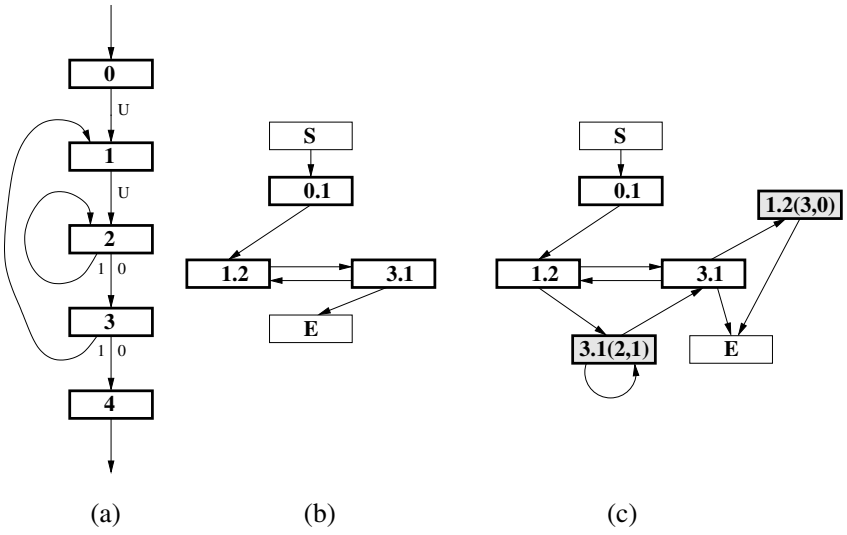

Figure 6. Changes to Cache Conflict Graph (Shaded nodes are ml-blocks)

$B_{3.1} \rightsquigarrow B_{1.2}^{3,0}$ belong to the first type. The edges $B_{3.1}^{2,1} \rightsquigarrow B_{3.1}$ and $B_{3.1}^{2,1} \rightsquigarrow B_{3.1}^{2,1}$ belong to the third and fourth type respectively.

Figure 6 shows the modeling of the constructive effect of wrong path prefetching. In the original CCG, there is an edge $B_{1.2} \rightsquigarrow B_{3.1}$ and that is the only path between the two nodes. Therefore, every time control reaches from $B_{1.2}$ to $B_{3.1}$, it is a cache miss. In the modified CCG in Figure 6(c), there is another path from $B_{1.2}$ to $B_{3.1}$ via the ml-block $B_{3.1}^{2,1}$. First, there is no cache miss along $B_{3.1}^{2,1} \rightsquigarrow B_{3.1}$ as they are physically the same l-block. Second, the cache miss along $B_{1.2} \rightsquigarrow B_{3.1}^{2,1}$ is partially masked by the branch misprediction delay. Thus, this kind of *prefetching* is constructive to the execution.

Additional constraints on ml-blocks The execution count of a normal l-block is equal to the execution count of the basic block it belongs to. However, for an ml-block $B_{i.j}^{b,X}$, this count is dependent on the number of mispredictions at branch b where the actual outcome is X (X is 0 or 1). To derive this execution count, note that the number of ml-blocks missed due to a single misprediction is $\left\lceil \frac{bmp}{cmp} \right\rceil$ where bmp (cmp) denotes branch misprediction penalty (cache miss penalty). In accordance with most modern processors, we assume $bmp < cmp$ and therefore $\left\lceil \frac{bmp}{cmp} \right\rceil = 1$. This assumption is, however, not required, and our modeling can be easily extended. Given $bmp < cmp$, a single misprediction can result in at most one cache miss along the mispredicted

path. Let $Spec(b, X) = \langle B_{u_1.v_1}, \ldots, B_{u_k.v_k} \rangle$. Therefore, the execution count of the ml-block $B^{b,X}_{u_i.v_i}$ is:

$$bm_b(X) - \sum_{l=1}^{i-1} cm^{b,X}_{u_l.v_l}$$

where $bm_b(X)$ is the number of mispredictions at branch b with outcome X (obtained from the modeling of branch prediction) and $cm^{b,X}_{u_l.v_l}$ is the number of cache misses for the ml-block $B^{b,X}_{u_l.v_l}$. Constraints on $cm^{b,X}_{u_l.v_l}$ are obtained from the CCG as shown in Equation 3 (refer to page 21). Constraints on $bm_b(X)$ are obtained from our modeling of branch prediction described in Section 4.1.

Objective function The objective function is:

$$\begin{aligned} Time \;=\; & \sum_{i=1}^{N} (cost_i \times v_i + bmp \times bm_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) \\ & + \sum_{\substack{Cond.\ branch\ b \\ X \in 0,1}} mp\_delay(b, X) \end{aligned} \tag{4}$$

The three subterms of the first term are the ideal execution time, the branch penalty and the cache penalty, respectively. The last term, $mp\_delay(b, X)$ is the delay that the processor has waited for pending cache misses (arising during mispredictions) after mispredictions have been resolved. As the assumption $bmp < cmp$ holds, the criteria for such a delay to happen are: (a) a cache miss happens during a misprediction, and (b) this cache miss is not completely masked by the misprediction (still pending when the branch is resolved). Recall that $Spec(b, X) = \langle B_{u_1.v_1}, \ldots, B_{u_k.v_k} \rangle$. We define:

$$mp\_delay(b, X) = \sum_{i=1}^{k} (cm^{b,X}_{u_i.v_i} \times delay^{b,X}_{u_i.v_i})$$

$$delay^{b,X}_{u_i.v_i} = cmp - (bmp - \sum_{l=1}^{i-1} cost_{u_l.v_l})$$

where $cost_{u_l.v_l}$ is the ideal execution time of the *l-block* $B_{u_l.v_l}$. Also, $delay^{b,X}_{u_i.v_i}$ is the delay introduced due to the cache miss of $B_{u_i.v_i}$ along the mispredicted path of branch b (where the actual outcome is X). This delay is not a constant, as part of the cache miss penalty cmp can be masked, depending on the location of the cache miss in the mispredicted path.

Table I. Description of Benchmark Programs.

Program	Description
`matsum`	Summation of two 100×100 matrices
`matmul`	Multiplication of two 10×10 matrices
`isort`	Insertion sort of 100-element array
`bsearch`	Binary search of 100 element array
`fdct`	Fast Discrete Cosine Transform
`fft`	1024-point Fast Fourier Transform
`dhry`	Dhrystone benchmark
`des`	Data Encryption Standard
`whet`	Whetstone benchmark
`fir`	FIR filter with Gaussian number generation

6. IMPLEMENTATION AND EXPERIMENTS

We select 10 different benchmarks for our experiments (refer to Table I). They have been used by other research groups for WCET analysis [9, 16]. The benchmarks can be divided into two groups according to size: the first four programs are smaller compared to the remaining six benchmarks. We can also categorize the 10 benchmarks according to their branch instructions. Thus, `matsum`, `matmult`, `fft` and `fdct` are loop intensive programs; `isort`, `bsearch`, `dhry`, `des`, `whet` and `fir` contain a number of conditional branches arising from if-then-else statements within nested loops.

6.1. Methodology

Since we want to examine the effects of instruction caching and branch prediction, we exclude the impact of other factors, such as data caching and data dependence among instructions. In our experiments, we assume a perfect processor pipeline with no stalls due to data dependencies. This allows each instruction to take a fixed number of clock cycles to execute. The only timing overhead is introduced by instruction cache misses and branch mispredictions of conditional branches.

We need to know the *actual WCET* of the benchmarks so as to evaluate the accuracy of our *estimated WCET*. To do this, we use the *SimpleScalar* architectural simulation toolset [3]. SimpleScalar instruction set architecture (ISA) is a superset of MIPS ISA – a popular embedded processor. Given a benchmark program, we attempt to iden-

tify the program input that will generate the WCET. Once this input is found, the worst case profile can be computed via SimpleScalar simulation. Among the benchmarks, `matsum`, `matmult`, `fft`, `fdct`, `dhry` and `whet` have only one possible input. Since the other programs have many possible inputs, determining their worst-case inputs can be tedious. In these programs, we use human guidance to select a set of inputs (which are suspected to increase execution time via cache misses and mispredictions). We call the maximum execution time we obtain from non-exhaustive simulation the *observed WCET*. Thus *observed WCET* $\leq$ *actual WCET* $\leq$ *estimated WCET*. Since the *actual WCET* of a benchmark is in general not available, we measure the accuracy of our estimation technique by comparing *estimated WCET* with *observed WCET*.

We write a prototype analyzer that accepts assembly language code annotated with loop bounds and recursion depths. Our analyzer is parameterized w.r.t. the cache configuration, the cache miss penalty, the predictor table size, the choice of prediction schemes and the misprediction penalty. The analyzer first disassembles the code, identifies the procedures/basic blocks and constructs the procedural calling graph/ control flow graph (CFG) of each procedure. The executions of each basic block are differentiated w.r.t. the calling context. From the graphs, our analyzer automatically generates the objective function and the linear constraints. These constraints, together with the functional constraints provided by the user or data flow analysis, are then submitted to an Integer Linear Programming (ILP) solver. In our experiments, we use CPLEX [7], a commercial ILP solver.

Parameters used in Experiments The default parameters in our experiments are as follows: (1) branch prediction scheme is *gshare*; (2) the two-bit branch history is XOR-ed with the 4 least significant bits of the branch address; (3) the branch misprediction penalty and cache miss penalty are five and 10 clock cycles, respectively; (4) with regard to the direct-mapped instruction cache, we have two settings for the two groups partitioned by their sizes. An eight-line cache with 16 bytes per line is used for the benchmarks with smaller sizes; for the other benchmarks, we use a 16-line cache with 32 bytes per line. Experiments on the impact of changing the parameters are reported later in the section.

6.2. Results for Branch Prediction Modeling

To measure the importance of modeling branch prediction, we first check its impact on execution time. If branch prediction is not modeled,

Table II. Modeling gshare Branch Prediction Scheme for WCET Analysis.

Program	**Obs.**	**Est.**	**Est./Obs.**
`matsum`	101821	101821	1.00
`matmul`	15084	15184	1.00
`isort`	47120	47251	1.00
`bsearch`	133	144	1.08
`fdct`	2513	2513	1.00
`fft`	219192	229406	1.04
`dhry`	128420	131024	1.02
`des`	53047	58022	1.09
`whet`	537125	571615	1.06
`fir`	29412	33145	1.12

all program branches have to be pessimistically estimated to be mispredicted for purposes of WCET analysis. This pessimism results in a 60 to 70% overestimation for some of the benchmarks, even assuming a meager three cycle branch misprediction penalty. In real-life processors, the branch misprediction penalty varies from three to 19 clock cycles. In fact, recent processors with deeper pipelines have substantial misprediction penalties. Thus, the actual impact of not modeling branch prediction will be higher on such platforms.

The results of branch prediction modeling are reported in Table II and Table III. Table II shows the observed WCET (column **Obs.**) obtained from SimpleScalar and the estimated WCET (column **Est.**) obtained from our ILP based technique. We use the popular gshare prediction scheme in these experiments. We also evaluate the accuracy of our estimation technique by presenting the ratio **Est./Obs.** Table III gives the detailed results for the three branch prediction schemes: *gshare*, *GAg* and *local*. Note the WCETs are in clock cycles while mispredictions are in counts. Our estimates of WCET and mispredictions are tight for all benchmarks as summarized by the **Est./Obs.** column in Table II.

Difficulty in Exploiting Temporal Path Information One reason for the overestimation of misprediction counts is the aggregate nature of the ILP approach. The ILP approach only allows us to provide linear constraints on basic block execution counts. However, path information (even if provided by the user) *cannot* be exploited by the ILP solver.

Table III. Observed and estimated WCET and misprediction counts of gshare, GAg and local schemes.

Pgm.	WCET					
	gshare		GAg		local	
	Obs.	Est.	Obs.	Est.	Obs.	Est.
matsum	101821	101821	101826	101826	101806	101806
matmul	15084	15184	15179	15179	15064	15064
isort	47120	47251	46185	47741	47135	47246
bsearch	133	144	123	148	113	133
fdct	2513	2513	2508	2508	2493	2493
fft	219192	229406	225932	249747	229552	229665
dhry	128420	131024	127425	129385	126405	127035
des	53047	58022	52942	58006	54207	57671
whet	537125	571615	571580	571610	571570	571580
fir	29412	33145	31177	34617	29622	33337

Pgm.	Mispredictions					
	gshare		GAg		local	
	Obs.	Est.	Obs.	Est.	Obs.	Est.
matsum	203	203	204	204	200	200
matmul	204	224	223	223	200	200
isort	391	400	204	596	394	399
bsearch	8	10	6	11	4	8
fdct	8	8	7	7	4	4
fft	3094	5140	4442	9205	5166	5193
dhry	2603	3170	2404	2800	2200	2406
des	574	1519	553	1509	806	1438
whet	3752	10650	10643	10649	10641	10643
fir	183	770	536	1074	225	820

For example, let us study a program segment of the `whet` benchmark given in Figure 7. Figure 7(a) is a loop body with loop iteration counts annotated. There are three `if-then-else` constructs embedded in the loop body. By taking a closer look, we can figure out that the outcomes of these branches are not dependent on the input data. The paths the loop body can take in each iteration is given in Figure 7(b). We can see there are only two paths and they alternate during the iterations. However, this temporal information cannot be fed into the ILP solver. Instead, the ILP solver uses the constraints in Figure 7(c) to implicitly consider any path satisfying these constraints. All such paths are

```
L0:  j = 1;
L1:  for (i = 1; i <= n4; i += 1) { /* 3450 */
L2:      if (j == 1)
L3:          j = 2;
L4:      else
L5:          j = 3;

L6:      if (j > 2)
L7:          j = 0;
L8:      else
L9:          j = 1;

L10:     if (j < 1 )
L11:         j = 1;
L12:     else
L13:         j = 0;
L14: }
```

(a) Source Code Segment

Itr.	Paths
(1)	L2 L3 L6 L9 L10 L13
(2)	L2 L5 L6 L7 L10 L11
(3)	L2 L3 L6 L9 L10 L13
(4)	L2 L5 L6 L7 L10 L11
.	.
.	.
.	.
(2n-1)	L2 L3 L6 L9 L10 L13
(2n)	L2 L5 L6 L7 L10 L11

(b) Paths in Loop

L3 = 1725	(1a)
L5 = 1725	(1b)
L7 = 1725	(2a)
L9 = 1725	(2b)
L11 = 1725	(3a)
L13 = 1725	(3b)

(c) Linear Constraints

Figure 7. A Fragment of the Whetstone Benchmark

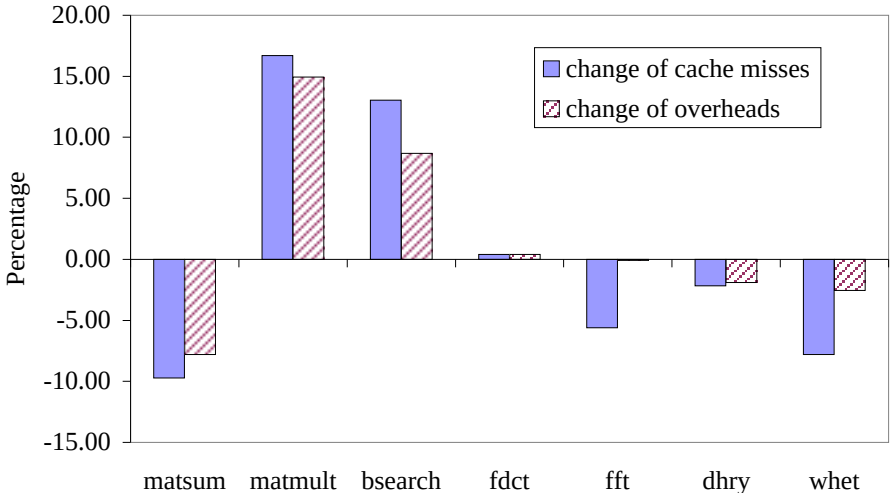

Figure 8. Change (in Percentage) of Cache Misses and Overall Penalties in Combined Modeling to Those in Individual Modelings

considered in the ILP solver's quest to maximize branch predictions (leading to overestimation).

6.3. Results for Integrated Modeling of Cache and Speculation

So far, we have presented the experimental results for our modeling of branch prediction. We now discuss the integrated modeling of instruction caching and branch prediction. First, we illustrate the importance of combined modeling of cache and speculation for WCET analysis by comparing it against a naive technique which models both caching and speculation but ignores the cache-speculation interaction. Figure 8 shows this comparison with benchmarks for which we can find the actual WCET (and the corresponding cache miss and branch misprediction overheads) through exhaustive simulation. This means that the number of feasible execution paths in these programs is not very large.

The first group of bars indicate the percentage increase/decrease in cache misses due to the effect of branch prediction on cache behavior. For the benchmarks `matmult`, `bsearch` and `fdct`, there are more cache misses in combined modeling than in naive modeling, indicating that the destructive effects of speculation are more significant than the constructive effects. For other programs, the constructive effects outperform the destructive effects, thereby decreasing the number of cache misses. The second group of bars shows the percentage change in total timing overhead of cache misses and branch mispredictions due to cache-speculation interaction. The timing overhead shows similar behavior as cache misses. The results show that if naive modeling is used (i.e., the effect of branch prediction on caching is not modeled), the WCET can either be overestimated (as the downward bars indicate), or, more seriously, be underestimated (as the upward bars indicate).

The comprehensive results for the combined modeling are presented in Table IV. Note that the numbers for the `WCET` columns are in processor cycles while the `Mispred.` and `Cache miss` columns denote misprediction and cache miss counts. As we can see from the `ratio` column, most benchmarks have very tight estimated bounds. Some benchmarks have lower estimated misprediction counts or cache misses than their observed counterparts, such as `dhry` and `fir`. This is because the ILP solver may trade fewer mispredictions for more cache misses (or vice versa) to maximize the overall WCET.

Modern processors have deeper pipelines and an increasing gap between processor speed and memory latency. Deeper pipelining leads to a larger misprediction penalty (in terms of clock cycles). The increasing processor-memory speed gap results in a longer cache miss penalty. Due to this trend of hardware advancement, we examine the accuracy of our WCET analysis with more aggressive parameters by doubling the *bmp* (from the default five clock cycles to 10 clock cycles) and the *cmp* (from

Table IV. Combined Modeling of Caching and Speculation: Observed and Estimated WCET, Misprediction Count and Cache Misses

Pgm.	WCET			Mispred		Cache miss	
	Obs.	Est.	Ratio	Obs.	Est.	Obs.	Est.
matsum	105504	105917	1.00	203	203	307	409
matmul	25155	25679	1.02	204	215	945	975
isort	48685	48836	1.00	391	400	107	109
bsearch	506	546	1.07	8	10	33	35
fdct	8798	8803	1.00	8	8	626	626
fft	219428	229651	1.04	3094	5139	21	25
dhry	218684	232523	1.06	2603	2514	8125	9639
des	87436	96437	1.10	574	1460	3255	3497
whet	545544	581557	1.06	3752	10580	765	986
fir	65223	67370	1.03	183	370	3506	3451

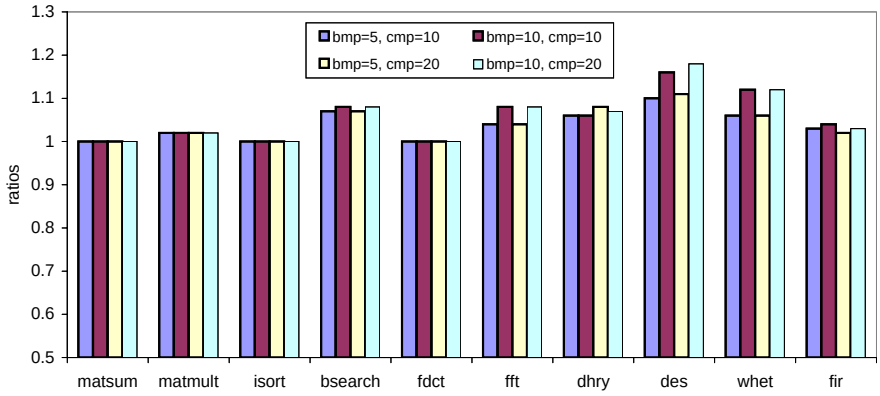

Figure 9. Est./Obs. WCET ratios under different **misprediction penalties** and **cache miss penalties**

10 clock cycles to 20 clock cycles). From the chart in Figure 9 we can see that for each benchmark, the Est/Obs WCET ratios change little under different penalty settings. That is, the accuracy of our analysis does not suffer from increasing penalties.

6.4. Scalability of our ILP based approach

The complexities of the programs and their solving times are given in Table V. The complexity of a program is presented by its number of basic blocks as well as its conditional branches. This is only an ap-

Table V. Program Complexity and Processing Time

Program	**Complexity**		**Time (seconds)**	
	Blocks	Branches	Formulation	Solving
`matsum`	5	2	0.02	0.01
`matmul`	7	3	0.03	0.01
`isort`	7	4	0.04	0.03
`bsearch`	9	3	0.03	0.01
`fdct`	5	2	0.05	0.01
`fft`	16	11	0.15	0.09
`dhry`	98	33	4.06	0.67
`des`	57	21	1.79	1.48
`whet`	36	18	0.62	0.37
`fir`	69	13	2.14	0.57

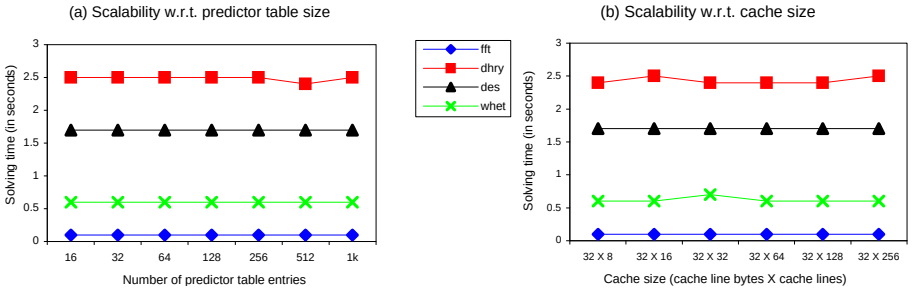

Figure 10. Scalability with Increasing Branch Prediction Table Size and Cache Size

proximate measure of the complexity. The column `Formulation` gives the times for automatically generating the ILP formulation; and the `Solving` column gives the ILP solving times by CPLEX. Here, the gshare scheme is used with the default parameters given in Section 6.1. As we can see, the ILP formulation times and ILP solving times of all benchmarks are within seconds.

We now consider the variation of ILP solution time for some benchmarks with larger predictor table sizes (the gshare scheme) and cache sizes. In Figure 10(a), the branch prediction table sizes vary from 16 to 1024 entries. Recall that in *gshare*, the branch instruction address is XOR-ed with the global branch history bits. In practice, the *gshare* scheme uses a smaller number of history bits than address bits, and XORs the history bits with the selected portion of the address [20]. The number of history bits is normally not very large as the correla-

tion among remote branches is very weak in most cases. So, we use a maximum of four history bits. Figure 10(a) shows that the ILP solving times do not change substantially. The reason is that, with an increasing number of history bits (from two to four bits), the number of possible patterns per branch increases. But with a fixed history size (four bits) and an increased prediction table size, the number of cases where two or more branches have the same pattern starts to decrease. Since the constraints for each individual pattern are independent of the other patterns, the complexity of the ILP problem largely depends on how many branches can execute with the same pattern. Thus, ILP solution time does not increase significantly with the increase in size of the branch prediction table.

Figure 10(b) shows the solving times when the instruction cache size is varied. Again, we observe that the solving time does not change substantially. One of the reasons is that the constraints for each cache line is independent of the other cache lines. Thus, increasing the number of cache lines does not change the structure of the ILP problem.

7. DISCUSSION

In this paper, we have presented a framework of WCET analysis which models the effects of advanced speculative execution and its interaction with caching. The integer linear programming technique is uniformly applied to program path analysis, branch prediction modeling and cache modeling. This makes it convenient to integrate them. Our experimental results indicate that our modeling is accurate. The destructive/constructive effects of branch prediction on cache behavior are demonstrated, showing the need to capture their interaction. This technique also scales up with regard to the increased size of the two hardware features: branch prediction table with larger size or larger instruction cache.

Acknowledgements

Preliminary versions of parts of this paper have been published elsewhere [14, 21]. We would like to thank the anonymous referees of DAC 2003 and ISSS 2002 for their comments. This work has been partially supported by National University of Singapore research projects R252-000-088-112 and R252-000-171-112.

References

1. Arnold, R., F. Mueller, D. Whalley, and M. Harmon: 1994, 'Bounding worst-case instruction cache performance'. In: *IEEE Real-Time Systems Symposium (RTSS)*.
2. Bate, I. and R. Reutemann: 2004, 'Worst-Case Timing Analysis for Dynamic Branch Predictors'. In: *30th EuroMicro Conference*.
3. Burger, D., T. Austin, and S. Bennett: 1996, 'Evaluating Future Microprocessors: The SimpleScalar Toolset'. Technical Report CS-TR96-1308, University of Wisconsin - Madison.
4. Chen, K., S. Malik, and D. August: 2001, 'Retargatable Static Software Timing Analysis'. In: *IEEE/ACM International Symp. on System Synthesis (ISSS)*.
5. Colin, A. and I. Puaut: 2000, 'Worst case execution time analysis for a processor with branch prediction'. *Journal of Real time Systems* **18**(2/3), 249–274.
6. Combs, J., C. Combs, and J. Shen: 1999, 'Mispredicted path cache effects'. In: *Euro-Par Conference*.
7. CPLEX: 2002, 'The ILOG CPLEX Optimizer v7.5'. Commercial software, http://www.ilog.com.
8. Engblom, J.: 2003, 'Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction'. In: *IEEE Real-time/Embedded Technology and Applications Symposium (RTAS)*.
9. Ferdinand, C., F. Martin, and R. Wilhelm: 1997, 'Applying compiler techniques to cache behavior prediction'. In: *ACM International Workshop on Languages, Compilers and Tools for Real-Time Systems*.
10. Healy, C., R. Arnold, F. Mueller, D. Whalley, and M. Harmon: 1999, 'Bounding pipeline and instruction cache performance'. *IEEE Transactions on Computers* **48**(1), 53–70.
11. Healy, C., M. Sjodin, V. Rustagi, D. Whalley, and R. Engelen: 2000, 'Supporting Timing Analysis by Automatic Bounding of Loop Iterations'. *Journal of Real-Time Systems* **18**(2/3), 129–156.
12. Hennessy, J. and D. Patterson: 1996, *Computer Architecture- A Quantitative Approach*. Morgan Kaufmann.
13. Lee, M., S. L. Min, and C. S. Kim: 1994, 'A Worst Case Timing Analysis Technique for Instruction Prefetch Buffers'. *Microprocessing and Microprogramming* pp. 681–684.
14. Li, X., T. Mitra, and A. Roychoudhury: 2003, 'Accurate Timing Analysis by Modeling Caches, Speculation and their Interaction'. In: *ACM Design Automation Conf. (DAC)*.
15. Li, Y.-T. S. and S. Malik: 1995, 'Performance Analysis of Embedded Software Using Implicit Path Enumeration'. In: *Workshop on Languages, Compilers and Tools for Real-Time Systems*.
16. Li, Y.-T. S., S. Malik, and A. Wolfe: 1999, 'Performance Estimation of Embedded Software with Instruction Cache Modeling'. *ACM Transactions on Design Automation of Electronic Systems* **4**(3), 257–279.
17. Lim, S.-S., Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim: 1995, 'An accurate worst-case timing analysis technique for RISC processors'. *IEEE Transactions on Software Engineering* **21**(7), 593–604.
18. Lundqvist, T. and P. Stenstrom: 1998, 'Integrating path and timing analysis using instruction-level simulation techniques'. In: *International Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*.

19. Lundqvist, T. and P. Stenstrom: 1999, 'An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution'. *Journal of Real-Time Systems* **17**(2-3), 183–207.
20. McFarling, S.: 1993, 'Combining Branch Predictors'. Technical report, DEC Western Research Laboratory.
21. Mitra, T., A. Roychoudhury, and X. Li: 2002, 'Timing Analysis of Embedded Software for Speculative Processors'. In: *ACM SIGDA International Symposium on System Synthesis (ISSS)*.
22. Park, C. and A. Shaw: 1991, 'Experiments with a program timing tool based on source-level timing schema'. *IEEE Transactions on Computers* **24**(5), 48–57.
23. Pierce, J. and T. Mudge: 1996, 'Wrong-path instruction prefetching'. In: *ACM International Symp. on Microarchitectures(MICRO)*.
24. Puschner, P. and C. Koza: 1989, 'Calculating the maximum execution time of real-time programs'. *Journal of Real-time Systems* **1**(2), 159–176.
25. Schneider, J. and C. Ferdinand: 1999, 'Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation'. In: *ACM International Workshop on Languages, Compilers and Tools for Embedded System (LCTES)*.
26. Shaw, A.: 1989, 'Reasoning about time in higher level language software'. *IEEE Transactions on Software Engineering* **15**(7), 875–889.
27. Theiling, H., C. Ferdinand, and R. Wilhelm: 2000, 'Fast and precise WCET prediction by separated cache and path analysis'. *Journal of Real Time Systems* **18**(2/3), 157–179.
28. Wolf, F., J. Staschulat, and R. Ernst: 2002, 'Associative caches in formal software timing analysis'. In: *ACM Design Automation Conference (DAC)*.
29. Yeh, T. and Y. Patt: 1992, 'Alternative Implementations of two-level adaptive branch prediction'. In: *ACM International Symp. on Computer Architecture (ISCA)*.
30. Zhang, N., A. Burns, and M. Nicholson: 1993, 'Pipelined Processors and Worst Case Execution Times'. *Journal of Real-Time Systems* **5**(4), 319–343.