A Framework to Model Branch Prediction for WCET Analysis

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

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

In this paper, we present a framework to model branch prediction for Worst Case Execution Time (WCET) analysis. Our micro-architectural modeling is completely generic, and parameterizable w.r.t. the currently used branch prediction schemes. It automatically derives linear constraints on the total misprediction count from the control flow graph of the program. These constraints can be solved by any integer linear programming (ILP) solver to estimate the WCET.

Current generation processors perform control flow speculation through branch prediction, which predicts the outcome of branch instructions. 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 between 3-19 clock cycles. If branch prediction is not modeled, all the branches in the program must be conservatively assumed to be mispredicted for finding the WCET. This pessimism results in as much as $60 - 70\%$ overestimation for some of the benchmarks in this paper, even assuming a 3 clock cycle branch misprediction penalty.

A classification of branch prediction schemes appears in Figure 1. Branch prediction can be *static* or *dynamic*. Static schemes associate a fixed prediction to each branch instruction via compile time analysis. Almost all modern processors, however, predict the branch outcome dynamically based on past execution history. Dynamic schemes are more accurate than static schemes, and in this work we study only dynamic branch prediction. The first dynamic technique proposed is called *local branch prediction* [4], where each branch is predicted based on its last few outcomes. This scheme uses a 2^n-entry *branch prediction table* to store the past branch outcomes, which is indexed by the n lower order bits of the branch address. In the simplest case, each prediction table entry is 1-bit and stores the last outcome of the branch mapped to that entry. When a branch is encountered, the corresponding table entry is looked up and used as the prediction. When a branch is resolved, the corresponding table entry is updated with the outcome. A more accurate version of local scheme uses k-bit counter per table entry.

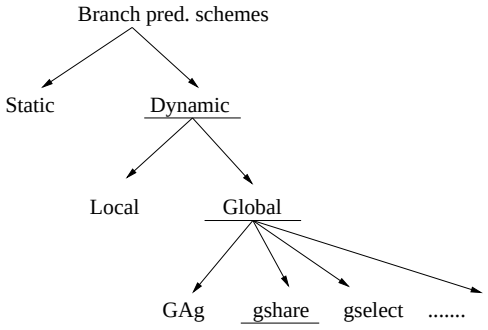

Figure 1: Classification of Branch Prediction Schemes. At each level, the more widely used category is underlined.

Most modern processors however use *global* branch prediction schemes [4] (also called correlation based schemes), which are more accurate. Examples of processors using global branch prediction include Intel Pentium Pro, AMD, Alpha as well as embedded processors IBM PowerPC 440GP and SB-1 MIPS 64. 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. Global schemes uses a single shift register, called *branch history register (BHR)* to record the outcomes of n most recent branches. As in local schemes, there is a global *branch prediction table* in which the predictions are stored. The various global schemes differ from each other (and from local schemes) in the way the prediction table is looked up when a branch is encountered.

Little work has been done to study the effects of branch prediction on WCET. Effects of static branch prediction have been investigated in [1, 3]. However, most current day processors (Intel Pentium, AMD, Alpha, SUN SPARC) implement dynamic branch prediction schemes, which are more difficult to model. To the best of our knowledge, [2] is the only other work on timing estimation under dynamic branch prediction. However, their technique only models the effects of local prediction schemes.

The starting point of our analysis is the control flow graph (CFG) of the program. Let v_i denote the number of times block i is executed, and let $e_{i,j}$ denote the number of times control flows through the edge $i \rightarrow j$. As inflow equals outflow, $v_i = \sum_{j \rightarrow i} e_{j,i} = \sum_{i \rightarrow j} e_{i,j}$. We provide bounds on the maximum number of iterations for loops and maximum depth of recursive invocations for recursive procedures. These bounds can be user provided, or can be computed off-line for certain programs.

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 $\sum_i (cost_i * v_i + penalty * m_i)$ 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. By maximizing this objective function we can get WCET.

Modeling Prediction Schemes To determine the prediction of a block i, we first compute the index into the prediction table. We define v_i^π and m_i^π: the execution count and the misprediction count of block i when branch in i is executed with index $= \pi$. By definition:

$$m_i^\pi \leq v_i^\pi \qquad m_i = \textstyle\sum_\pi m_i^\pi \qquad v_i = \textstyle\sum_\pi v_i^\pi$$

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

Our modeling is independent of the definition of the prediction table index π. Hence it can apply to any branch prediction scheme that uses a single prediction table. To model the effect of different branch prediction schemes, we only alter the meaning of π, and show how π is updated with the control flow.

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). To model the change in history due to control flow, we use the left shift operator ; thus $left(\pi, 0)$ shifts pattern π to the left by one position and puts 0 as the rightmost bit. We define:

Definition 1 *Let $i \rightarrow j$ be an edge in the control flow graph and let π be the BHR content at basic block i. The change in history pattern on executing $i \rightarrow j$ is given by $\Gamma(\pi, i \rightarrow j) = \pi$ if $i \rightarrow j$ is an unconditional jump. If $i \rightarrow j$ is a taken (non-taken) branch then $\Gamma(\pi, i \rightarrow j)$ is $left(\pi, 0)$ ($left(\pi, 1)$).*

In the popular *gshare* [4] scheme, the BHR is XOR-ed with last n bits of the branch address to look up the prediction table. Usually, *gshare* results in

Pgm.	gshare		GAg		local	
	Mispred		Mispred		Mispred	
	Obs.	Est.	Obs.	Est.	Obs.	Est.
check	3	3	3	3	198	198
matsum	204	204	204	204	200	200
matmul	223	223	223	223	200	200
fdct	7	7	7	7	4	4
fft	3678	6165	3398	5175	4129	5154
isort	9687	9952	587	598	399	399
bsearch	9	9	9	10	6	7
eqntott	203	205	202	206	203	204

Table 1: Observed and estimated misprediction count with gshare, GAg, and local schemes.

a more uniform distribution of table indices compared to *GAg*. We define the index π as $\pi = history_m \oplus address_n(I)$ where m, n are constants, $n \geq m$, $\oplus$ is XOR, $address_n(I)$ denotes the lower order n bits of I's address, and $history_m$ denotes the most recent m branch outcomes (which are XOR-ed with higher-order m bits of $address_n(I)$). And,

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

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

Bounding Mispredictions Given the definition of π and Γ, we derive inflow and outflow constraints on the flow of π through the control flow graph to derive upper bounds on v_i^π. To bound m_i^π, we note the following. Suppose there is a misprediction of the branch in block i with history π. This means that certain blocks (maybe i itself) were executed with history π, the outcome of these branches appear in the πth row of the prediction table, and the outcome of these branches *must have created* a prediction different from the current outcome of block i. To model mispredictions, we therefore capture repeated occurrence of a history π during program execution with differing outcomes; we provide constraints to bound such occurrences. Details of our modeling appear in [5] and are ommitted here for space considerations.

Experimental Results We selected eight different benchmarks for our experiments. We assumed zero cache misses and a perfect processor pipeline with no stalls except for penalty due to misprediction of conditional branches. These assumptions, although simplistic, allow us to separate out and measure the accuracy of our estimation technique. We assumed that the branch misprediction penalty is 3 clock cycles (as in the Intel Pentium processor). We used the SimpleScalar architectural simulation platform in the experiments. By changing SimpleScalar parameters, we could change the branch prediction scheme for the experiments.

To evaluate the accuracy of our branch prediction modeling, we present the experiments for three different branch prediction schemes: *gshare*, *GAg* and *local*. Since finding the worst case input of a benchmark (which produces the actual WCET) is a human guided and tedious process, we only measured the actual WCET assuming a 4-entry prediction table. The results appear in Table 1. In this table, we have shown only the observed and estimated misprediction counts to enable clear understanding of the accuracy of our technique (which models the effect of branch prediction). Even though not shown here due to space shortage, the estimation accuracy was independent of the prediction table size. Our esti-

mation technique obtains a very tight bound on the WCET and misprediction count in all benchmarks except `fft`. The reason is that the number of iterations of the innermost loop of `fft` depends on the loop iterator variable value of the outer loops. This problem can be solved by providing expressions on the loop iteration counts instead of constants, as shown in [2].

Using CPLEX, a commercial ILP solver distributed by ILOG, on a Pentium IV 1.3 GHz processor with 1 GByte of main memory, our timing estimation technique requires less than 0.11 second for all the benchmarks with prediction table size varying 4–1024 entries.

One major concern with any ILP formulation of WCET is the scalability of the resulting solution. To check the scalability of our solution, we formulated the WCET problem for the popular *gshare* scheme with branch prediction table size varying from 4–1024 entries. Recall that in *gshare*, the branch instruction address is XOR-ed with the global branch history bits. In practice, *gshare* scheme uses smaller number of history bits than address bits, and XORs the history bits with the higher order address bits [4, 6]. The choice of the number of history bits in a processor depends on the expected workload. In our experiments, we used a maximum of 4 history bits as it produces the best overall branch prediction performance across all our benchmarks. As Figure 2 shows, the number of variables generated for the ILP problem initially increases and then decreases. With increasing number of history bits, number of possible patterns per branch increases. But with fixed history size and increasing prediction table size, the number of cases where two or more branches have the same pattern starts to decrease. This significantly reduces the number of ILP variables.

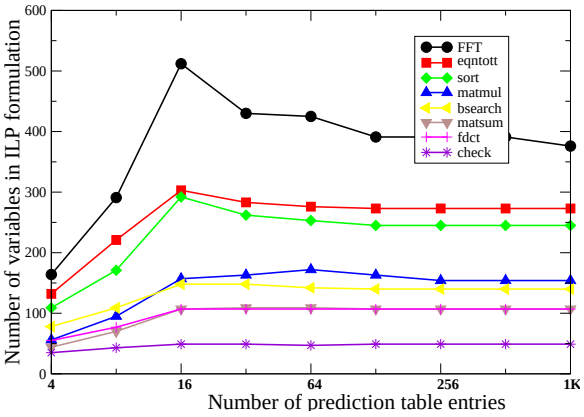

Figure 2: Change in ILP problem size with increase in number of entries in the branch prediction table for *gshare* scheme

References

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

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

[3] 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.*

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

[5] T. Mitra and A. Roychoudhury. Effects of branch prediction on worst case execution time of programs. Technical Report 11-01, School of Computing, National University of Singapore, 2001.

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