A Contribution to Branch Prediction Modeling in WCET Analysis

Claire Burguière and Christine Rochange
Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier, CNRS
Toulouse, France
{burguier,rochange}@irit.fr

Abstract

The wider and wider use of high-performance processors as part of real-time systems makes it more and more difficult to guarantee that programs will respect their strict deadlines. While the computation of Worst-Case Execution Times relies on static analysis of the code, the challenge is to model with enough safety and accuracy the behaviour of intrisically dynamic components. In this paper, we focus on the dynamic branch predictor. Several models to bound the number of branch mispredictions have been previously published. Some of them exhibit a high complexity while other ones have shown that taking into account semantic information from the source code makes things more tractable. We extend this work to more general nested loop structures. We also give some simulation results that show that the way branch mispredictions are usually taken into account cannot be both safe and accurate in the case of high-performance pipelines. We propose a more realistic approach to be used as part of WCET computation.

1. Introduction

In real-time systems, the knowledge of the Worst-Case Execution Time (or *WCET*) of programs is required to be able to define a scheduling of the different tasks that guarantees the fulfilment of strict deadlines. Dynamic methods based on measurement do not allow a safe evaluation of the WCET because it generally cannot be ensured that all the possible execution paths have been explored. Static analysis methods are then used to compute an upper bound of the WCET [4].

Some components of the processor architecture exhibit a very dynamic behaviour that depends on the past history of the processor state, which makes the evaluation of the WCET by static analysis complex. Among these dynamic components, we focus here on the branch predictor.

The *time predictability* of a processor architecture relies on the availability of techniques that can model this architecture to calculate an upper bound of the execution time. The complexity of these techniques should be taken into account and a processor might be declared *unpredictable* if computation and/or memory requirements for analysing the WCET are prohibitive. The precision of the WCET bound can also moderate this notion of time predictability: if the analysis relies on very pessimistic assumptions about the hardware, the system cannot be considered as predictable. So we say that a processor is *predictable* if it can be analysed with reasonable cost and if the estimated WCET is not too far from the real WCET. By extension, we say that a branch predictor is *predictable* if it does not alter the predictability of the processor.

Taking into account the branch predictor within the WCET analysis involves: (a) estimating the number of branch mispredictions, and (b) integrating the misprediction penalties in the computation of the WCET. As it will be exposed in section 2, several previous works have addressed the question of bounding the number of mispredictions, even for advanced predictors. However, the proposed models are extremely complex and thus it is questionable whether these predictors can be considered as predictable. We will propose some modifications to the branch predictor to improve its predictability. In addition, we feel that the modeling of misprediction penalties in the case of a high-performance pipeline has not correctly been addressed up to now. One contribution of this paper, described in section 3, is the suggestion of a way to integrate the misprediction penalties in the set of constraints used to compute the WCET by the IPET method [5]. In section 4, we extend previous results [2] to fit our requirements to model safely and tightly branch mispredictions, and also to make them applicable to more general algorithmic structures. Finally, we conclude and we present our further work.

2. Modeling branch prediction

2.1. Overview of branch prediction schemes

We focus here on dynamic branch prediction schemes since, by nature, static prediction algorithms are fully predictable. Dynamic schemes make use of the control flow history to predict the outcome of any branch encountered in the fetched instruction flow. The outcome of a branch includes its direction (*taken, not taken*) and, if the branch is predicted as *taken*, its target address. The control flow history is kept in prediction tables: directions are stored in the *Branch History Table (BHT)* while target addresses are stored in the *Branch Target Buffer (BTB)*.

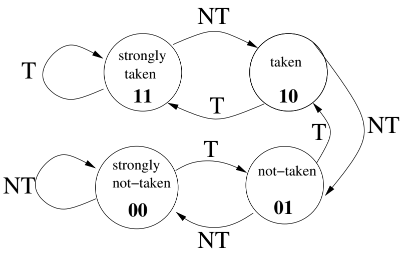

Figure 1. Saturating two-bit predictor counter

Information about the direction history is often represented by a 2-bit saturating counter as illustrated in Figure 1. The BHT is indexed either by the branch PC (*local* scheme), or by a global history vector that keeps the directions of the last predicted branches (*GAg*), or by some function (e.g. XOR) of the PC and the global history (*gshare*) [7]. For economic reasons, the BHT is generally not tagged and one entry might be shared by several branches (which is not detected by the hardware). This phenomemon is referred to as *aliasing*. It has been shown that, on the average case, aliasing is not too much damaging for the overall performance. Nevertheless, when considering the worst case, its possible destructive effects have to be taken into account.

Since it is very unlikely that two branches share the same target address (while the 2-bit counter only has 4 possible values), it is commonly admitted that the BTB has to be tagged. Then conflicts between branches still might occur but they can be detected by the hardware. Whenever the BHT indicates that a branch should be predicted as taken but the tag in the BTB does not match the branch PC, the branch is predicted by default, i.e. not taken.

2.2. Related work

In a recent paper [2], Bate and Reutemann study branches involved in the implementation of simple algorithmic structures (loops and conditional statements). They show how the number of mispredictions for these branches can be bounded while ignoring the aliasing problem. We extend their work to more general loop nests and we show how to compute data that fit our requirements to model the effects of branch prediction on high-performance pipelines.

Colin and Puaut [3] use static simulation, previously proposed to analyse cache memories [9], to compute the abstract state of the BHT at the time a branch has to be predicted (they consider a local scheme where the BHT is indexed by the branch PC). From this state and from an analysis of the program structure (loop nests), they determine for each branch whether it will be correctly predicted. Li, Mitra and Roychoudhury [8] integrate the modeling of branch prediction into the computation of the WCET by the IPET method by adding a number of constraints. Their model allows taking into account several schemes to index the BHT. Both works model the effects of aliasing in the BHT but they ignore the possible conflicts in the BTB. Li *et al.*'s model is appropriate to analyse advanced branch predictors but, due to its complexity, it might not fit the tractability requirements for predictability. We retain that most of this complexity comes from the need of modeling destructive aliasing in the BHT. In the next part, we propose to prevent aliasing to gain in predictability.

2.3. Preventing aliasing to improve predictability

The mixed branch predictor [10] has been proposed to reduce the branch misprediction rate. It allocates each entry of the branch prediction table to a particular branch instruction, and the other branches that would index the same entry are statically predicted. In the context of a high-performance processor, it was proposed to keep in the table for dynamic prediction those branches that are the most frequenlty executed (they can be selected by execution profiling). Here, the purpose is to make the branch predictor more predictable. We suggest that it would be more appropriate to allocate the table entries to the most predictable branches, i.e. the branches with the easiest to estimate misprediction rates. This includes branches with a regular behaviour, like loop control branches, as we will see in section 4. These branches can be automatically identified from a Control Flow Graph representation of the code [1]. A careful selection of the branches to be included in the dynamic predictor should also ensure that no destructive conflict can occur in the Branch Target Buffer.

3. Branch prediction penalty

Previous works consider a simplified view of the impact of a branch misprediction: it is assumed to be constant, at worst for all of the conditional control flow instructions (CI) in the program [3] and at best for every CI [8] (i.e. the penalty differs from one CI to another one but, for a given CI, the misprediction penalty is the same on both possible paths).

We have made some measurements to estimate the validity of this assumption. We have used a cycle-level simulator of a generic 4-way superscalar dynamically-scheduled processor. The benchmarks (*fibcall, jfdctint, matmul, crc, ludcmp, lms, fft1*) were taken from the SNU benchmark suite (http://www.archi.snu.ac.kr/realtime/benchmark/) and were compiled without optimization for the PowerPC 603 target. For each benchmark, we have extracted the control flow graph from the binary code. Conditional branch instructions were identified as the ending instructions of basic blocks (nodes) that have two outgoing edges. For each possible outcome of each CI, we have simulated the two-block sequence (i.e. the block ended by the CI and the target block), first with the branch well predicted and second with the branch mispredicted. The difference between the two simulated times is the misprediction penalty. Figure 2 shows the cumulative distribution of the misprediction penalty over all the benchmarks. It can be observed that the penalty is far from being constant and ranges from 0 to 11 cycles.

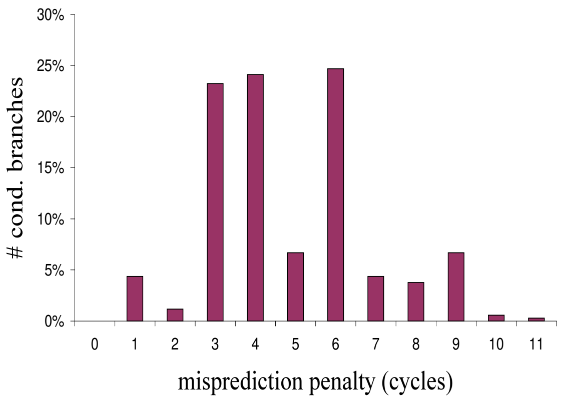

Figure 2. Distribution of branch prediction penalties

Then we compared, for every CI, the difference between the penalties observed on the two possible paths. Figure 3 shows the cumulative distribution of these differences over all the benchmarks.

We can remark that the difference can be large (up to 7 cycles, which is significant compared to the execution time of short basic blocks). Thus, we will obtain a more reliable and tighter WCET if we distinguish the two branch

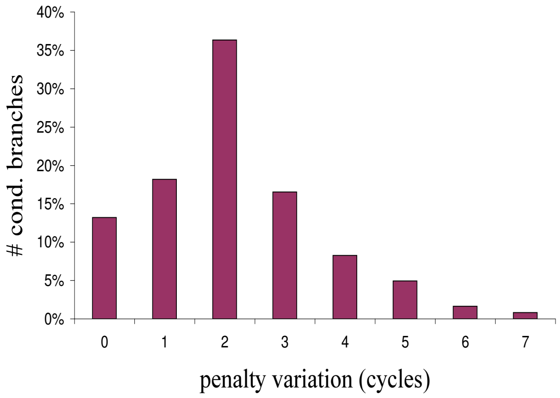

Figure 3. Difference on the branch prediction penalty between both possible paths

directions and if we bound the number of mispredictions for both of them. Figure 4 shows how to modify the control flow graph to model the effects of branch prediction: block *I* ends with a conditional branch instruction and is followed by block *J* if the branch is taken and by block *K* otherwise. Each possible outcome is represented by two edges to model that the branch can be well-predicted (*wp*) or mispredicted (*mp*).

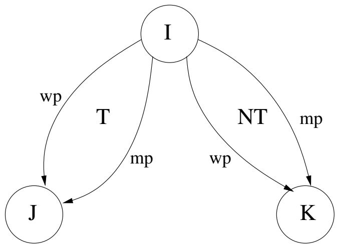

Figure 4. Representing branch prediction in the control flow graph

The proposed CFG makes it possible to integrate the analysis of the worst-case branch predictor behaviour in the computation of the WCET by the IPET method [5]. Each element of the CFG has an execution time (for an edge, the execution time represents the gain due to the pipelined execution of two successive basic blocks): in the case of a block ending with a CI, each outgoing edge is doubled to model two possible values of the gain, depending on whether the CI is correctly predicted or not. In the case of a misprediction, the gain will generally be lower due to the recovery penalty (however, note that in some very particular situations, a misprediction could generate a shorter execution time, due to some execution anomalies as the ones described in [6]). Let:

- T be the overall execution time
- b_i be the number of times block i is executed
- t_i be the execution time of block i
- s_{ij}^{mp} be the number of times sequence $i \rightarrow j$ is executed with the branch mispredicted
- s_{ij}^{wp} be the number of times sequence $i \rightarrow j$ is executed with the branch well-predicted
- t_{ij}^{mp} be the execution time of sequence $i \rightarrow j$ when the branch is mispredicted
- t_{ij}^{wp} be the execution time of sequence $i \rightarrow j$ when the branch is well-predicted

The expression of the overall execution time is:

$$\boxed{\begin{array}{c} T = b_i \times t_i + b_j \times t_j + b_k \times t_k + s_{ij}^{mp} \times t_{ij}^{mp} + \\ s_{ij}^{wp} \times t_{ij}^{wp} + s_{ik}^{mp} \times t_{ik}^{mp} + s_{ik}^{wp} \times t_{ik}^{wp} \end{array}}$$

Computing the WCET using the IPET method comes to maximizing T while respecting structural constraints extracted from the CFG (e.g. $b_i = s_{ij}^{mp} + s_{ij}^{wp} + s_{ik}^{mp} + s_{ik}^{wp}$). In the absence of further information about the possible values of s_{ij}^{mp} and t_{ij}^{mp}, the result of this optimization problem would probably be that all branches are mispredicted because the execution time of a sequence is generally higher in the case of a misprediction. To be able to obtain a more accurate estimation of the WCET, it is desirable to derive as-tight-as-possible upper bounds on the numbers of branch mispredictions on both path. In the next section, we extend recent work by Bate and Reutemann [2] for this purpose.

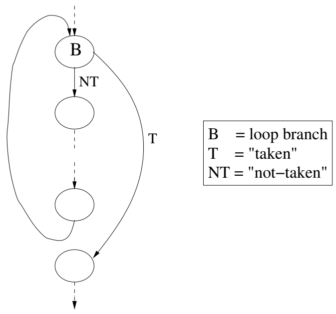

Figure 5. Loop structure

4. Evaluation of an upper bound on the number of mispredictions

In this section, we will show how to compute an upper bound on the number of mispredictions for *predictable* branches. Among predictable branches, we focus on loop branches and we consider the loop compilation scheme represented in Figure 5: path "T" (taken) leaves the loop and path "NT" enters it for a new iteration.

Bate and Reutemann [2] have already analysed the number of mispredictions for this kind of loop. We propose to extend their study by differenciating the mispredictions on both paths of the branch.

Considering the simple loop of Figure 5, Table 1 gives the number of mispredictions for both possible paths as a function of the number n of loop iterations and of the initial counter state. Note that, in each case, the total number of mispredictions is the same as the one computed in [2].

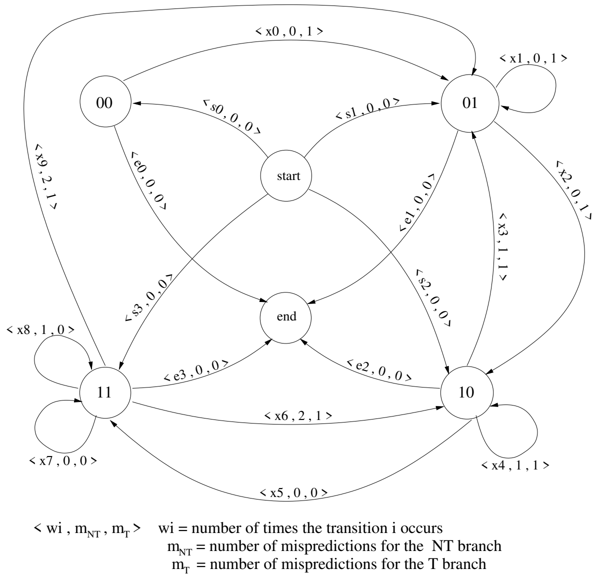

Figure 6. Evolution of the 2-bit counter for a nested loop

Bate and Reutemann also model nested loops (the loop under analysis is nested in another one and then is executed several times). They assume that the internal loop has a fixed number of iterations. We consider a more general case where the internal loop can have a *variable* number of iterations. If the external loop is executed M times, we have to compute the evolution of the two-bit counter for the M executions of the internal loop. This evolution can be represented by the automaton given in Figure 6: each state (except for start and end) represents one possible value of the 2-bit counter, each edge represents one execution of the internal loop and the starting and ending states of an edge are the values of the 2-bit counter before and after executing the loop. Each edge is annotated by a triplet $< w_i, m_{NT}, m_T >$: w_i is the number of times the transition is executed, m_{NT} and

Init	number of iterations (2-bit counter evolution / number of mispredictions (NT/T))							
	$n = 0$		$n = 1$		$n = 2$		$n \geq 3$	
00	$00 \rightsquigarrow 01$	0/1	$00 \to 00 \rightsquigarrow 01$	0/1	$00 \to 00 \to 00 \rightsquigarrow 01$	0/1	$00 \cdots \rightsquigarrow 01$	0/1
01	$01 \rightsquigarrow 10$	0/1	$01 \to 00 \rightsquigarrow 01$	0/1	$01 \to 00 \to 00 \rightsquigarrow 01$	0/1	$01 \to 00 \cdots \rightsquigarrow 01$	0/1
10	$10 \to 11$	0/0	$10 \rightsquigarrow 01 \rightsquigarrow 10$	1/1	$10 \rightsquigarrow 01 \to 00 \rightsquigarrow 01$	1/1	$10 \rightsquigarrow 01 \to 00 \cdots \rightsquigarrow 01$	1/1
11	$11 \to 11$	0/0	$11 \rightsquigarrow 10 \to 11$	1/0	$11 \rightsquigarrow 10 \rightsquigarrow 01 \rightsquigarrow 10$	2/1	$11 \rightsquigarrow 10 \rightsquigarrow 01 \cdots \rightsquigarrow 01$	2/1

Table 1. Number of mispredictions for both directions of a branch for a simple loop which iterates n times ($\rightsquigarrow$ represents a transition with misprediction)

m_T are the numbers of mispredictions of both branch directions. The transitions and their annotations are derived from Table 1. Each transition that starts from a given state stands for one (or several) particular number(s) of iterations of the loop.

This automaton constitutes a support for computing upper bounds. Considering M iterations of the external loop, we can write:

$$\sum_{i=0}^{9} x_i = M \quad (1)$$

$$\forall i,\ 0 \leq x_i \leq M \quad (2)$$

$$\sum_{i=0}^{3} e_i = 1 \text{ and } \sum_{i=0}^{3} s_i = 1 \quad (3)$$

First, we calculate the number of mispredictions for the NT direction. It is given by:

$$m_{NT} = x_0 + x_3 + x_4 + 2x_6 + x_8 + 2x_9$$

Thanks to equation (1) we can write:

$$m_{NT} = M - x_1 - x_2 - x_5 + x_6 - x_7 + x_9 \quad (4)$$

Since the "11" state is neither final nor initial, we have:

$$x_9 + x_6 + e_3 = s_3 + x_5$$

which is equivalent to:

$$x_9 + x_6 - x_5 = s_3 - e_3 \quad (5)$$

Equations (3) give:

$$0 \leq s_3 \leq 1 \text{ and } 0 \leq e_3 \leq 1$$

Thus, we obtain:

$$-1 \leq s_3 - e_3 \leq 1 \quad (6)$$

Equations (2) and (6) drive to:

$$-x_1 - x_2 - x_5 + x_6 - x_7 + x_9 \leq 1 \quad (7)$$

From equations (4) and (7), we can derive an upper bound of m_{NT}:

$$m_{NT} \leq M + 1$$

Note that this upper bound can be reached (e.g. sequence s_3, x_8, x_6, e_2).

We can bound the number of mispredictions on the T path in the same manner. From Figure 6, we write:

$$m_T = x_0 + x_1 + x_2 + x_3 + x_4 + x_6 + x_9$$

and

$$m_T = M - x_5 - x_7 - x_8 \quad (8)$$

From equation (2), we can write:

$$-x_5 - x_7 - x_8 \leq 0$$

And thus :

$$m_T \leq M$$

Again this bound can be reached (e.g. sequence s_3, x_6, e_2).

Thus, for a nested loop executed M times with a variable number of iterations, the upper bounds on the number of mispredictions for both possible paths are :

$$\boxed{\begin{array}{c} m_T \leq M \\ m_{NT} \leq M + 1 \end{array}}$$

and then:

$$m_T + m_{NT} \leq 2M + 1$$

This bound on the total number of mispredictions can also be reached (e.g. s_3, x_9, s_1).

Considering a fixed number of loop iterations, Bate and Reutemann [2] have found that the total number of mispredictions was $2M$. This bound is reached whenever the loop iterates once each times is executed. For a number of iterations greater than 3, the maximum number of mispredictions is $M + 2$. Our results show that using their model by fixing the (variable) number of iterations to a worst-case value can lead to underestimating the number of mispredictions. For example, let us consider a loop executed three times that iterates twice, then once and finally more than three times. Assuming that the initial counter state is "11", the total number of mispredictions is $3 + 2 + 2 = 7$ (these figures are taken from Table 1). The upper bound given by our model is $2M + 1 = 7$, which is safe. Using Bate and Reutemann's model and fixing the number of loop iterations to its maximal value (greater than three) would give an underestimated bound of $M + 2 = 5$. Even fixing the number of iterations to their worst-case value (a single iteration) would give an unsafe bound ($2M = 6$). This shows that the previous model cannot directly be used to evaluate the general case (variable iteration number) and the extension we propose is useful to this case.

5. Conclusion

Dynamic branch predictors make the computation of the Worst-Case Execution Time of programs based on static analysis difficult. Several models have been proposed in the past to bound the number of mispredictions. However, it appears that modeling the aliasing effect, that might have a destructive impact on the branch prediction tables, requires a tremendous effort. Then the resolution of the model might not be tractable. We argue in favour of allowing the use of simpler models by eliminating possible conflicts. Previous work done for high-performance applications could be adapted to a real-time context.

We also have made the point that previous studies use a simplistic view of the impact of a branch misprediction on the execution time. We have given some measurement results that show that this penalty can vary significantly from one branch to the other one, but also, for a given branch, from one possible path to the other one. This is due to the use of advanced algorithms to schedule the instructions in superscalar out-of-order pipelines. This point makes it necessary to bound separately the number of mispredictions on each possible path. We have shown how this could be done as part of the computation of the WCET with the IPET method.

Finally, we have extended previous work by Bate and Reutemann [2] to differentiate the *taken* path from the *not-taken* one. We also have considered more general loop nests, where the internal loop can have variable iteration counts.

Future work includes developping an algorithm to select *predictable* branches to allocate in the branch prediction table. We also intend to extend the bounding of misprediction numbers to more complex algorithmic structures.

References

[1] T. Ball and J. Larus. Branch prediction for free. In *ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, volume 28, pages 300–313, june 1993.

[2] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *16th Euromicro Conference on Real Time Systems*, pages 215–222, june 2004.

[3] A. Colin and I. Puaut. Worst case execution time analysis for a processors with branch prediction. In *Real-Time Systems*, pages 249–274, may 2000.

[4] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry-strength worst-case execution time analysis. Technical report, ASTEC 99/02, April 1999.

[5] Y.-T. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, pages 88–98, 1995.

[6] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, page 12, december 1999.

[7] S. McFarling. Combining branch predictors. Technical Report TN-36, Western Research Laboratory, june 1993.

[8] T. Mitra and A. Roychoudhury. A framework to model branch prediction for wcet analysis. In *WCET Analysis workshop held in conjunction with Euromicro Conference on Real-Time Systems*, june 2002.

[9] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, july 1994.

[10] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *6th International Symposium on High-Performance Computer Architecture*, page 251, january 2000.