A Case for Static Branch Prediction in Real-Time Systems

Claire Burguière, Christine Rochange, Pascal Sainrat
Institut de Recherche en Informatique de Toulouse, France
{burguier, rochange, sainrat}@irit.fr

Abstract

Taking dynamic branch prediction into account in WCET determination turns out to be complex, particularly because of the possible interferences between branches. In this paper we argue the case for using static instead of dynamic branch prediction: the aliasing problem is swept away and, in many cases, the estimated worst-case numbers of branch mispredictions are reduced. We propose a method to predict each branch at compile time. Experimental results show how effective this approach can be.

1. Introduction

The design of applications that have to meet strict deadlines makes it essential to estimate their Worst-Case Execution Time (WCET). Academic research promotes techniques based on static code analysis that limit measurements to small parts of code (e.g. basic blocks) against dynamic approaches that measure the execution time of complete paths and may require unacceptable measurement times.

To take branch prediction into account in static WCET analysis, the common approach consists in evaluating an upper bound on the number of branch mispredictions so that the corresponding penalties can be included in the computation. Several models of the behaviour of more or less advanced dynamic branch predictors have been proposed. Some of them focus on the prediction counters [3][4] while other ones also consider the effect of aliasing in the prediction table [6][11] (aliasing effects are more often than not destructive and, in the general case, they increase the possible number of branch mispredictions).

In this paper, our purpose is to show that the *worst-case* numbers of mispredictions would be lower with static per-branch prediction than with a dynamic scheme. We propose heuristics to statically predict an outcome for each branch, on the basis of the program algorithmic structure and of the individual WCETs of the basic blocks. Experimental results show that, considering state-of-the-art estimation techniques, the computed WCET is lower with our static prediction scheme than with a bimodal dynamic scheme.

The paper is organized as follows. Section 2 gives an overview of branch prediction and section 3 surveys previous work on modeling dynamic branch predictors to compute the WCET. Section 4 shows how static branch prediction can help in lowering the estimated WCET. The interest of the proposed approach is confirmed by experimental results given in section 5 and concluding remarks are given in section 6.

2. An overview of branch prediction

Branch prediction includes static techniques that predict a fixed outcome for each branch and dynamic techniques that analyse the history of the control flow to anticipate its future behaviour.

The simplest static schemes predict the same direction (taken or not-taken) for every branch. More sophisticated strategies, implemented in the compiler, exploit information on the overall program structure to predict a specific outcome for each branch [2].

The direction of a conditional branch instruction can also be dynamically predicted (by the hardware) on the basis of the outcomes of past branches. The history of a branch is generally stored as a 2-bit saturating counter, updated every time the branch is executed, as shown in Figure 1. The outcome of a branch is predicted according to the 2-bit counter value: taken whenever the upper bit is 1, and not taken otherwise [17].

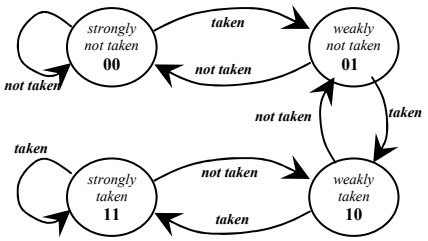


Figure 1. Evolution of a 2-bit saturating counter.

Whenever a branch is encountered in the instruction flow, both its direction and its target address have to be anticipated. Previously computed target addresses are stored in the Branch Target Buffer (BTB), while the 2-bit counters used to predict the direction are maintained in the Branch History Table (BHT).

However the branch prediction tables are indexed, different branches might compete for the same entry. The BTB is tagged (with the branch PC), which allows detecting possible conflicts. But, to save transistors, the BHT is generally not tagged, and several branches can share the same entry. This phenomenon is referred to as *aliasing*. In some cases, aliasing can be constructive (i.e. a branch can take advantage from the fact that its 2-bit counter has been updated by another branch) but it is more often destructive.

On the whole, dynamic branch prediction performs better than static branch prediction. However, in some particular cases, static branch prediction can be as good as dynamic branch prediction. First, some conditional branches are highly biased (e.g. error tests) and a static prediction for these branches is almost always correct. Second, the static prediction of a conditional branch implementing a loop is as least as efficient as a bimodal predictor (it is incorrect only at the loop exit). It can even more be better in the case of destructive aliasing. On the other hand, some advanced dynamic branch predictors can predict correctly the conditional branch of loops with small numbers of iterations and might also capture dynamic correlation between branches. However, very few processors implement such advanced schemes. Finally, dynamic prediction might be more efficient for branches related to conditional statements that depend on the input data (the condition might be constant during the execution of the program but unknown at compile time).

Most of today processors feature hardware static branch prediction. In the simplest case like in the ARM1020e [1], all branches are predicted according to the BTFN algorithm (Backward Taken, Forward Not taken) while some processors, like the PowerPC [8], allow the user to invert this algorithm through a bit in the branch instruction codes. Support for compiler-directed static prediction is provided in several processors, like the PowerPC 750, by the means of branch hints (a bit in the opcode specifies the predicted outcome). High-performance processors also include a dynamic branch predictor and, most of the time, a bit in a control register makes it possible to select from static or dynamic branch prediction. The Intel Itanium 2 [9] goes further by making it possible to choose the prediction mode (static/dynamic) on a per-branch basis.

3. Modeling dynamic branch prediction

3.1 Impact of mispredictions on the WCET

The execution time of a two-basic-block sequence is generally shorter than the sum of their two individual execution times: the gain is due to the overlap of the two blocks in the pipeline. If the first block is ended by a conditional branch and if this branch is mispredicted, the gain is reduced by the *misprediction penalty* that stands for the time spent in fetching instructions along the wrong path until the branch is resolved, flushing them out of the pipeline and redirecting the instruction fetch to the correct path. The penalty is not the same for all the branches in the program, and also varies for a given branch from one direction to the other one [4].

Here, we assume that the WCET is computed using the IPET approach [10]. To take the impact of branch prediction into account, we have suggested modifying the CFG as illustrated in Figure 1 [4]. Each edge is weighted by the gain due to the overlap of the two blocks in the pipeline, and, for the "mispredicted-branch" edges, this gain is reduced by the misprediction penalty.

In the case where no bound is specified for the number of executions of these edges, it is commonly assumed that maximizing the overall execution time comes to considering that the branch is always mispredicted because the corresponding edges are longer (even if timing anomalies discussed in [13] might, in rare cases, invalidate this statement). To calculate a tighter WCET, it is desirable to be able to provide upper bounds on the numbers of mispredictions. In the rest of this section, we will show how the bounds can be derived.

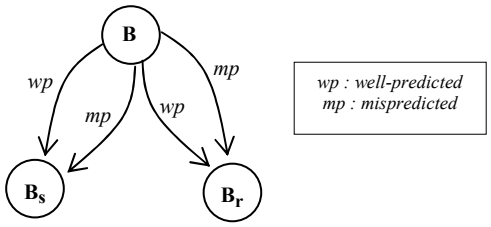


Figure 1. Modified Control Flow Graph

The question of evaluating the number of branch mispredictions has been addressed in several recent papers. Some researchers propose to decouple this evaluation from the WCET determination and to statically analyse the behaviour of branches linked to algorithmic constructs [3][4][5][6]. The behaviour of the branch predictor can also be expressed within the overall model used to compute the WCET [11].

3.2 Modeled dynamic prediction schemes

Modelling the behaviour of the branch predictor involves: (i) expressing the evolution of the counter used to make the prediction, (ii) taking into account possible conflicts (aliasing) in the prediction table and their possible destructive effects.

Bate and Reutemann [3] assume that a careful allocation of the code in memory prevents aliasing. Patil and Emer [15] select some non-conflicting branches to be handled by the dynamic predictor and make a compiler-directed static prediction for the remaining ones. As said above, choosing between static and dynamic prediction on a per-branch basis is now possible with some processors like the Itanium 2 [9]. Once the aliasing problem is removed, the behaviour of a 2-bit counter can be simply analysed according to the algorithmic construct it relates to and the number of mispredictions can be bounded. Previously published results will be exposed below.

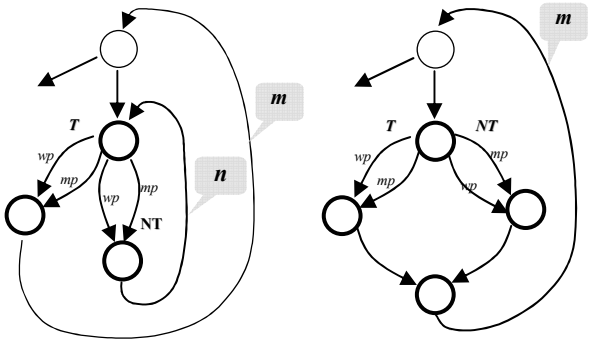
Other studies count the effects of aliasing. Colin and Puaut [6] use static simulation to determine whether the prediction counter associated to a branch might be corrupted by another branch. Then they combine these results with an analysis of the behaviour of the 2-bit counters based on the algorithmic structures to calculate the misprediction numbers. Mitra and Roychoudhury [11] add constraints that model the possible interferences in the BHT as well as the evolution of the counters to the set of structural and flow constraints used to compute the WCET by IPET. The number of new constraints is substantial.

3.3 Bounding branch mispredictions

3.3.1. Loops. Let us consider the loop construct illustrated in Figure 2a. It iterates n times, and is repeated m times. The conditional branch that controls the loop is taken to exit the loop.

This kind of construct has been studied in previous work and several cases have been tackled. First, Bate and Reutemann [3] focused on loops with a fixed number of iterations (n). We extended their work to address the case where the number of iterations n varies from one execution of the loop to the other one. We considered two cases: (a) the evolution of n does not match any particular pattern (most general case) [4], and (b) the loop has a constantly increasing (or decreasing) number of iterations: it is executed m_0 times with 0 iteration, then m_1 times with 1 iteration, then m_2 times with 2 iterations and then m_3 times with more than 3 iterations (in this order, or in the reverse order), with $m_0 + m_1 + m_2 + m_3 = m$. We have found this last case in several benchmarks. For example, the `four1` benchmark includes the code given in Figure 3, where the internal loop iterates once, then twice, then 4 times, 8 times, etc. We found that the upper bound of the number of mispredictions could be refined from the general case for this particular kind of loops [5].

The worst-case number of mispredictions for all these different cases, split between the two possible paths (*T* when the branch is *taken* and *NT* when it is *not taken*), are given in Table 1 (DYNAMIC columns). The proofs can be found in [3][4][5].



(a) repeated loop **(b) repeated cond. statement**

Figure 2. CFGs associated to statements

number of iterations (*n*)	*number of executions* (*m*)	*worst-case number of mispredictions*			
		NT (branch not taken) path		*T (branch taken) path*	
		DYNAMIC	***STATIC***	***DYNAMIC***	***STATIC***
fixed	**= 1**	$\min(n, 2)$	0	1	1
	> 1	1 if n=0 m if n=1 3 if n=2 2 if n≥3	0	m	m
variable with "regular" pattern	**> 0**	$\min(2,m_3) + \min(3, 2m_2) + m_1$	0	$m_3 + m_2 + m_1 + \min(2, m_0)$	m = $m_0+m_1+m_2+m_3$
variable (general case)		$m + 1$	0	m	m

Table 1. Comparison of static or dynamic prediction for a loop branch.

comparison of path lengths with dynamic prediction	*statically predicted path*	*worst-case number of mispredictions*			
		NT (branch not taken) path		*T (branch taken) path*	
		DYNAMIC	*STATIC*	*DYNAMIC*	*STATIC*
¬longer(NT) && ¬longer(T)	*NT*	$\frac{m}{2}+1$	*0*	$\frac{m}{2}+1$	*m*
	T	$\frac{m}{2}+1$	*m*	$\frac{m}{2}+1$	0
longer(NT)	*NT*	2	0	0	0
longer(T)	*T*	0	0	2	0

Table 2. Comparison of static and dynamic prediction for a conditional statement.

```
while (n > mmax) {
    istep = mmax << 1;
    for (m=1 ; m<mmax ; m+=2) {
        …
    }
    mmax = istep;
}
```

Figure 3. Example of a repeated loop with a variable-but-regular number of iterations.

3.3.2. Conditional statements (if-then-else). Bate and Reutemann also analyse the behaviour of a branch that implements a repeated conditional statement [3] as the one shown in Figure 2b. As Colin and Puaut [6], they identify two possible worst cases:

- when the two paths (*then* path and *else* path) have comparable execution times, the worst-case is when they are executed alternately because the conditional branch is then always mispredicted.
- if one path is longer than the other one (by a certain margin that they express as twice the branch misprediction penalty, which they consider to be constant), the worst-case is when this path is always taken. In this case, the branch is mispredicted at most twice (bimodal scheme).

As said before, the misprediction penalties vary from one branch to the other one, and from one path to the other one. In the condition that defines which worst-case has to be considered, Bate and Reutemann assumed a fixed penalty: in [5], we extended their work to to take into account different penalties. Table 2 gives the number of mispredictions on each path, with *longer (x)* expressing that path *x* is the longest one with respect to the above condition (DYNAMIC columns). These formulas were proved in [3][6].

4. Reducing the estimated WCET through static prediction

Our claim is that, most of the time, compiler-directed static branch prediction can reduce the worst-case number of branch mispredictions.

In the case of a loop, the branch should be statically predicted as *not taken*. Then, if the loop iterates *n* times, the branch is well-predicted *n* times (each time it enters the loop) and mispredicted once (when it exits the loop). If the loop is repeated *m* times, the branch is well-predicted each time it enters the loops (i.e. $n \times m$ times) and mispredicted each time it exits the loop (i.e. *m* times). As far as conditional structures are concerned, the predicted outcome should be to the path that has the longest execution time when the branch is mispredicted.

Tables 1 and 2 give, side by side, the maximum numbers of mispredictions on each path for static and dynamic (bimodal) branch prediction. In the case of loops, the statically predicted outcome is *not taken* and the *NT* path is always well-predicted. Static prediction can then avoid the control hazards that must be taken into account when the branch prediction is done by the hardware. The worst-case misprediction number on the *T* path is almost the same for both prediction schemes, except for the repeated loop with a constantly increasing (or decreasing) number of iterations: in that case, compiler-directed prediction engenders a few more mispredictions. For conditional statements, static branch prediction might increase the worst-case number of mispredictions. However, it should be noted that a higher number of mispredictions does not always mean a longer execution time. To illustrate this point, let us assume that the execution time of the *NT* path is higher than the execution time of the *T* path, whether the branch is mispredicted or not: the worst-case execution path is the *NT* path in any case. Then, if the statically predicted outcome is *not taken*, the *m* mispredictions that would occur on the *T* path do not have any impact on the overall WCET.

The first stage in static branch prediction consists in identifying algorithmic constructs from the CFG. This can be done by structural analysis [14][16], a method derived from interval analysis which converts the CFG into a depth-first spanning tree and searches for known structural patterns like the ones presented in Figure 4.

As constructs are identified, a control tree is built to express their hierarchical relationships.

The second stage is to make the predictions, which is straightforward. For loop branches, the predicted outcome is *not taken* (we assume that loops are compiled in such a way that the branch is taken to exit the loop). Branches that implement conditional statements are predicted to the path that has the longest execution time when the branch is mispredicted.

This leads to a bottom-up approach where the control-tree produced by the structural analysis of the control flow graph is analysed from its leaves to its root. The static prediction is made on the way: when a predicted outcome has been fixed for a branch, the WCETs of both paths are estimated. These times can later be used to make a decision concerning another conditional branch at a higher level in the control-tree.

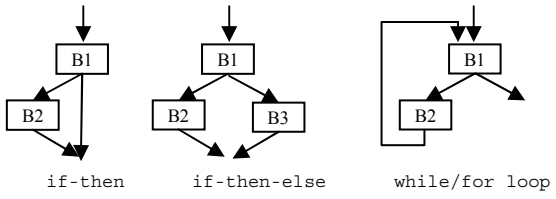


Figure 4. Patterns for structural analysis [14]

5. Evaluation of the approach

To estimate the impact of our compiler-directed static branch prediction scheme, we use a framework that makes it possible to analyse the WCET by the IPET method. It includes a cycle-level simulator that models a generic 4-way superscalar processor, with out-of-order execution of the instructions (including memory operations). Using a Perl script, we generate the set of structural constraints that describe the CFG and are required to compute the WCET by IPET. We add the flow constraints as well as the constraints on branch mispredictions (upper bounds) by hand. Finally, we use the `lp_solve` tool to maximize the overall execution time. We used four benchmarks from the SNU suite. They were compiled for the PowerPC 603 target, which is supported by our simulator.

We have computed the WCET of four benchmarks considering three different models of branch prediction. The *basic* model assumes that every conditional branch is mispredicted. The *dynamic* model represents a dynamic predictor based on 2-bit counters indexed by the branch address, with no aliasing in the prediction table [4]. The *static* model stands for our compiler-directed static prediction scheme. Estimated WCETs for these three models are given in Table 3 and the improvement on the WCET provided by our approach is given in Table 4.

Program	WCET according to the branch prediction model		
	basic	***dynamic***	***static***
`four1`	57 871	44 426	42 741
`matmul`	42 674	35 414	35 354
`bsort`	473 440	344 986	344 938
`isort`	149 853	91 455	90 831

Table 3. Estimated WCETs.

Program	Improvement due to *static* branch prediction	
	over *basic*	over *dynamic*
`four1`	32.7 %	3.8 %
`matmul`	17.1 %	0.2 %
`bsort`	27.1 %	0.01 %
`isort`	39.4 %	0.7 %

Table 4. Improvement due to static branch prediction over the Basic and the Dynamic models.

For every benchmark, the estimated WCET is lower with our static branch prediction than with the *basic* and *dynamic* models. The gain over the *basic* scheme is large, as it could be expected. The improvement over the *dynamic* scheme is slighter but always positive. The `matmul` program only contains nested loops with constant numbers of iterations and the *dynamic* scheme is very efficient for this kind of constructs. The `bsort` benchmark includes loops with fixed numbers of iterations as well as conditional statements with same-length paths. As shown above, the WCET estimated for static prediction is almost the same as the one computed for dynamic prediction. Static prediction also yields a WCET close to the one computed of dynamic prediction in the case of repeated loops with regularly variable iteration numbers as in the `isort` application. Finally, `four1` records the highest improvement due to a nested loop with an irregular number of iterations.

Our aim in this work was to *not* improve the average case performance of branch prediction schemes but rather to determine which scheme is the most appropriate when the WCET of programs is to be estimated. Our experimental results make it possible to compare the different techniques with respect to the computed WCET, but average-case performance results are out of the scope of the paper.

For every loop, the estimated WCET is lower with static branch prediction than with a dynamic scheme, because the path that enters the loop is never mispredicted. As far as repeated conditional statements are concerned, the difference between static and dynamic prediction depends on the relative lengths of the two possible paths. In the case where they have comparable lengths, the total number of mispredictions is *m (*number of executions of the statement) with both

schemes. Again, it is difficult to draw conclusions on the resulting WCET when the number of mispredictions is reduced on one path and increased on the other one because it depends on their respective execution times and mispredictions penalties.

Experimental results show that the WCET estimated for the *basic* model (i.e. considering that every branch is mispredicted) can be noticeably improved by enabling static branch prediction. As we show it, making the predictions at compile-time is straightforward when the goal is to lower the worst-case (and not the average-case) execution time.

On the other hand, the gain over the dynamic scheme is most often limited. However, it must be emphasized that the dynamic scheme considered here is alias-free. As we mentioned it, taking aliasing into account introduces pessimism in the estimated worst-case numbers of mispredictions. Then we can expect that static prediction would yield a more appreciable improvement if a real dynamic scheme was analysed. Moreover, modelling dynamic branch prediction has proved to be complex, particularly when aliasing is taken into account. On the contrary, the impact of static branch prediction on the WCET is straightforward.

6. Conclusion

Modern processors feature advanced mechanisms which are more and more difficult to model for the evaluation of the Worst-Case Execution Time of real-time codes. One of these mechanisms is dynamic branch prediction, which has been recently studied by several research groups. Modelling dynamic branch prediction means analysing the evolution of the 2-bit counters that maintain the branch histories and taking into account interferences in the branch prediction tables. This second part is the most difficult to handle and accounts for most of the model complexity.

In this paper, we argued in favour of static branch prediction as a means of getting rid of the aliasing problem. We showed that, in most of the cases, the estimated WCET is lower with static than with dynamic prediction (bimodal scheme). We proposed heuristics to predict branch outcomes at compile-time with the aim of reducing the estimated WCET.

Experimental results presented in this paper show that the proposed approach is relevant. Larger improvement would probably be observed if static prediction was compared to dynamic prediction *with* aliasing. Moreover, the important thing is that, even if the gain on the estimated WCET is limited, using compiler-directed branch prediction makes the WCET analysis much easier.

7. References

[1] ARM Corp., *ARM1020T Technical Reference Manual, ref. ARM DDI 0135A*.

[2] T. Ball, J. Larus, "Branch Prediction for Free", ACM Conference on Programming Language Design and Implementation, 1993.

[3] I. Bate, R. Reutemann, "Worst-Case Execution Time analysis for Dynamic Branch Predictors", *16th Euromicro Conf. on Real-Time systems*, June 2004.

[4] C. Burguière, C. Rochange, "A Contribution to Branch Prediction Modeling in WCET Analysis", *Conf. on Design, Automation and Test in Europe (DATE)*, March 2005.

[5] C. Burguière, C. Rochange, "Analysing branch mispredictions related to algorithmic constructs", Technical Report IRIT-2005-12-R, March 2005.

[6] A. Colin, I. Puaut, "Worst-Case Execution Time Analysis for Processors with Branch Prediction", *Real-Time Systems*, vol. 18, n° 2-3, May 2000.

[7] J. Engblom, "Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction", *9th IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2003.

[8] IBM Corp., *PowerPC 750 RISC Microprocessor Technical Summary*, august 1997.

[9] Intel Corp., *Intel Itanium 2 Reference Manual for Software Development and Optimization*, June 2002.

[10] Y.-T. Li, S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", *ACM SIGPLAN Notices*, vol. 30, 1995.

[11] X. Li, T. Mitra, A. Roychoudhury, "Modeling Control Speculation for Timing Analysis", *Real-Time Systems*, vol. 29, n°1, January 2005.

[12] Y.-T. Li, S. Malik, A. Wolfe, "Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Cache", *17th IEEE Real-Time Systems Symposium*, 1996.

[13] T. Lundqvist, P. Stenström, "Timing Anomalies in Dynamically-Scheduled Microprocessors", *20th IEEE Real-Time Systems Symposium*, 1999.

[14] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[15] H. Patil, J. Emer, "Combining Static and Dynamic Branch Prediction to Reduce Destructive Aliasing", *Symp. on High-Performance Computer Architecture*, January 2000.

[16] M. Sharir, "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers", *Computer Languages*, vol. 5, n°3/4, 1980.

[17] J. Smith, "A Study of Branch Prediction Strategies", *8th Int. Symposium on Computer Architecture*, 1981.