Analysing Branch Mispredictions Related to Algorithmic Constructs

RAPPORT INTERNE IRIT-2005-12-R

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

April 2005

Résumé : De nos jours, les pipelines haute performance sont de plus en plus utilisés dans les sytèmes temps réel et il devient de plus en plus difficile de garantir que les programmes respecteront leurs échéances. Les éléments de l'architecture des processeurs récents ont des comportements dynamiques complexes à modéliser pour le calcul du pire temps d'exécution (WCET) qui est basé sur une étude statique du code. Dans ce rapport, nous étudions la prédiction de branchement dynamique. Quelques modèles permettant de borner le nombre de mauvaises prédiction ont été publiés. Alors que certains de ces modèles présentent une importante complexité, d'autres proposent d'utiliser des informations sémantiques provenant du code source pour simplifier ce calcul. Nous étendons ces travaux en affinant les bornes du nombre de mauvaises prédictions pour les branchements conditionnels de type "si-alors-sinon". Nous proposons aussi un calcul plus précis de ces bornes pour les boucles répétées ayant un nombre d'itérations qui évolue de façon croissante (ou décroissante).

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 intrinsically dynamic components. In this report, 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 these works, as regards conditional statements, taking into account more accurate misprediction penalties. We also propose a more precise upper-bound on the number of mispredictions for repeated loops with an increasing (or decreasing) number of iterations.

Chapter 1

Introduction

The design of applications that have to meet strict deadlines makes it essential to be able to estimate their Worst-Case Execution Time (WCET). These last twenty years, much research work has been carried out to develop frameworks for WCET analysis. Academic researchers promote techniques based on static program 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 be confronted to path coverage problems as well as unacceptable measurement times. Part of the published work [20, 18, 13, 1] aims to find the possible paths in the code (or rather the infeasible ones). Other studies focus on modelling the impact of some components of the processor architecture on the overall execution time: mechanisms that exploit the execution history (e.g. the cache memory, the branch predictor, etc.) cannot be analysed locally within each basic block and they require the use of algorithms that consider all the possible paths together. Usual techniques are based on static simulation [3, 8], abstract interpretation [2], integer linear programming [16].

As far as branch prediction is concerned, the aim is to evaluate an upper bound on the number of branch mispredictions so that the corresponding penalty can be taken into account while computing the WCET. As exposed in chapter 3, several approaches have been proposed these last five years to model the behaviour of more or less advanced dynamic branch prediction schemes. Some of these approaches limit the analysis to the behaviour of the prediction counters [5, 7] while others also model the effect of aliasing in the prediction table [8, 14] (aliasing effects are more often than not destructive and, in the general case, they increase the possible number of branch mispredictions).

In this report the aim is to refine the computation of upper-bounds on the number of mispredictions as regards two specific algorithmic constructs. First, we focus on repeated conditional statements ("if-then-else") and we refine previous work on the condition which permits to guarantee that the longest execution time is taken into account in the WCET. We also compute the upper-bound on the number of mispredictions for repeated loops with an increasing (or decreasing) variable number of iterations.

The report is organized as follows. Chapter 2 gives an overview of branch prediction schemes. Chapter 3 details how to model branch prediction mechanisms in WCET analysis. In chapter 4, we focus on the computation of upper-bounds on the number of mispredictions for repeated conditional statements and loops with an increasing (or decreasing) number of iterations. Concluding remarks are presented in chapter 5.

Chapter 2

An overview of branch prediction strategies

Branch prediction mechanisms include 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. In this section, we will survey these two categories of techniques and we will discuss their respective strong points. Then, we will review the branch prediction schemes available in off the shelves processors.

2.1 Static branch prediction

Various strategies have been proposed to predict a fixed outcome for each branch [17]. The simplest mechanisms predict the same direction (taken or not-taken) for every branch, or exploit basic information extracted from the branch instruction code (e.g. the BTFN scheme predicts backward branches as taken and forward branches as not-taken). These simple algorithms can be implemented either at the hardware-level or at the compiler-level. More sophisticated schemes, implemented in the compiler, exploit information on the overall program structure to predict a specific outcome for each branch [4]. Profile-based approaches analyse execution traces produced by the instrumented program code and predict each branch according to its favourite direction [12].

2.2 Dynamic branch prediction schemes

The direction of a conditional branch instruction can 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 2.1. 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 [21]. 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). The BTB is generally indexed by the branch address (Program Counter, or PC). The BHT can also be indexed by the PC, as in the bimodal scheme [21]. Some predictors maintain History Registers, i.e. shift registers that store the outcomes of the n last branches. History Registers (HRs) can be local (each branch updates its own history) or global (all the branches share the same history) to exploit possible correlations between branches: the outcome of a given branch may depend on the directions of other branches previously executed [22]. The HRs can be used to index the BHT: this way, a given branch can be predicted from different 2-bit counters according to the recent history.

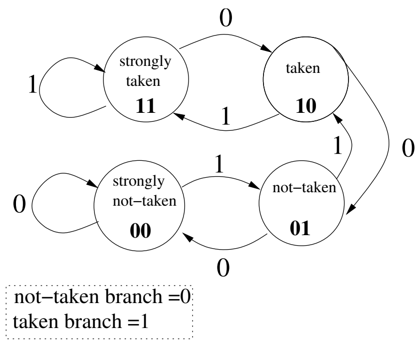

Figure 2.1: Evolution of the 2-bit saturating counter

However 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 is particularly true when the BHT is indexed by the Global History Register. This phenomenon is referred to as aliasing. Aliasing can be limited by using complex indexing schemes (e.g. XOR-ing some bits of the branch PC with the global history) but it cannot be completely avoided. 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.

2.3 Static vs. dynamic branch prediction

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 at 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, dynamic branch prediction with local history registers can predict correctly the conditional branch of loops with small numbers of iterations. Furthermore, dynamic branch prediction can capture dynamic correlation between branches. However, we will see in the next section that very few processors implement such advanced dynamic 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).

2.4 Branch prediction in real processors

Most of today processors feature hardware static branch prediction. In the simplest case like in the ARM1020e [9], all branches are predicted according to the BTFN algorithm (Backward Taken, Forward Not taken) while some processors, like the PowerPC [10], 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 (e.g. a bit in the opcode to specify the predicted outcome). High-performance processors usually 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 [11] goes further by making it possible to choose the prediction mode (static or dynamic) on a *per-branch* basis, thanks to instruction hints.

Chapter 3

Modelling dynamic branch prediction to estimate the WCET

3.1 Impact of branch mispredictions on the execution time

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. As shown in previous work, 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 [7]. In this report, we assume that the WCET is computed using the IPET approach [15] which seems to be the most widely used. The Control Flow Graph (CFG) of the program is described by a set of constraints that express the relationships between the respective numbers of executions of the different basic blocks along an execution path. Some other constraints express the results of the preliminary flow analysis (loop bounds, infeasible paths, etc.). The global execution time is the sum of the execution times of the basic blocks weighted by their respective numbers of executions, and the WCET is estimated by maximizing this function under the structural and flow constraints. To take the impact of branch prediction into account in the evaluated WCET, we suggest modifying the CFG as follows. Each outing edge of a block ended by a conditional branch is broken down into two sub-edges that account for the case the branch is well-predicted or mispredicted, as illustrated in Figure 3.1. Each edge is weighted by the gain due to the overlap of the two basic 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, maximizing the overall execution time comes to considering that the branch is always mispredicted because the corresponding edges are longer. If we want to calculate a tighter WCET, it is desirable to be able to provide upper bounds on the numbers of mispredictions. In the next chapter, we will show how the bounds have been derived in previous work and how we propose to compute more accurate bounds.

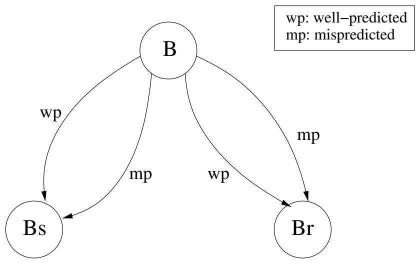

Figure 3.1: Modelling the impact of branch prediction

3.1.1 Introducing the impact of mispredictions within WCET computation

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 calculation of the WCET and to statically analyse the behaviour of branches linked to algorithmic constructs [8, 5, 7]. Another approach consists in modelling the behaviour of the branch predictor within the overall model used to compute the WCET [14].

3.2 Modelled 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 [5] assume that a careful allocation of the code in memory prevents aliasing. Patil and Emer [19] propose to select some non-conflicting branches to be handled by the dynamic hardware predictor and to make a compiler-directed static prediction for the remaining ones. As said in section 2.4, choosing between static and dynamic prediction on a *per-branch* basis is now possible with some processors like the Itanium 2 [11]. Once the aliasing problem is removed, the behaviour of a 2-bit counter can be simply analysed according to the algorithmic construct it implements (a loop or a conditional statement, executed one or several times) and the number of mispredictions can be bounded. Details on the computation of these bounds will be given in chapter 4. Other studies take into account the effects of aliasing. Colin and Puaut [8] use the static simulation technique to determine whether the prediction counter associated to a branch might be corrupted by another branch (they consider that the prediction table is indexed by the branch PC). Then they combine these results with an analysis of the behaviour of the 2-bit counters based on the algorithmic structure of the code to calculate the misprediction numbers. Mitra and Roychoudhury [14] 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. They consider a predictor where the BHT is indexed by a global history but their model can be adapted to other schemes (table indexed by a local history or by the branch PC). The number of new constraints is substantial, even though they have limited their study to 1-bit counters. Extending their model to 2-bit counters seems to dramatically increase the overall complexity.

Chapter 4

Bounding the number of branch mispredictions

4.1 Conditional Statements ("if-then-else")

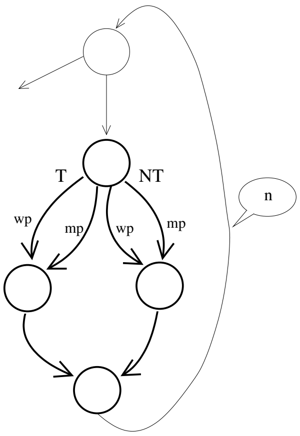

Figure 4.1: Control flow graph associated to a repeated conditional statement ("if-then-else")

In this chapter we focus on repeated "if-then-else" conditional statements (figure 4.1). Bate and Reutemann [5], as Colin and Puaut [8], identify two possible worst-cases for this algorithmic structure:

- one possible worst case occurs when the branch is mispredicted each time it is executed. This happens when both directions (*then* path and *else* path) are executed in alternation.
- the second possible worst-case is encountered when one path (*then* path or *else* path) is

longer than the other one. The execution time is greater if the longest path is always executed. The branch is then mispredicted at most twice.

The decision of which case is to be considered as the worst-case is based on a formula that we will refer to as the *worst-case condition*. In previous work [5, 8], they consider that misprediction penalty is the same for both directions of the branch. However, as shown in [7], this penalty varies, for a given branch, from one direction to the other one. Taking into account *per-direction* penalties increases the precision of the estimated WCET. But the *worst-case condition* has to be extended.

Let:

- t_1 and t_2 be the execution times of both paths when the branch is correctly predicted
- m_1 and m_2 be the execution times of both paths when the branch is mispredicted
- $p_1 = m_1 - t_1$ and $p_2 = m_2 - t_2$ are the misprediction penalties of both branches
- $d = t_1 - t_2$ is the difference between execution times of the two paths.

In order to derive general results, we label the two path with 1 and 2 instead of NT (for Not-Taken) and T (for Taken).We assume that path 1 has the longest execution time when the branch is mispredicted.

$$m_1 \geq m_2$$

This assumption does not imply any relation between the misprediction penalties and p_1 can be lower than p_2.

The overall execution time when only the longest path is executed (and mispredicted twice) is given by:

$$T_l = 2 * m_1 + (n - 2) * t_1$$

The overall execution time when the two paths are executed in alternation (and the branch is always mispredicted) is given by:

$$T_a = (\lfloor n/2 \rfloor + 1) * m_1 + (\lceil n/2 \rceil - 1) * m_2$$

where n is the number of executions of the statement. The *worst-case condition*, that defines when the second case is the worst case, is: $T_l > T_a$.

If n is even $\lceil n/2 \rceil = \lfloor n/2 \rfloor = n/2$ and the equation can be simplified:

$$T_a = (n/2+1) * m_1 + (n/2-1) * m_2$$

.

Then the *worst-case condition* is:

$$2 * m_1 + (n-2)t_1 > (n/2+1) * m_1 + (n/2-1) * m_2$$

By definition: $t_1 = d + t_2$, $m_1 = p_1 + t_1 = p_1 + d + t_2$ and $m_2 = p_2 + t_2$. We replace t_1, m_1 and m_2 by their value:

$$2 * (p_1 + d + t_2) + (n-2) * (d + t_2) > (n/2+1) * (p_1 + d + t_2) + (n/2-1) * (p_2 + t_2)$$

Then:

$$n * t_2 + 2 * p_1 + n * d > (n/2+1) * p_1 + (n/2+1) * d + (n/2-1) * p_2 + n * t_2$$

$$\iff 2 * p_1 + n * d > (n/2+1) * p_1 + (n/2+1) * d + (n/2-1) * p_2$$

$$\iff (n - (n/2+1)) * d > ((n/2+1) - 2) * p_1 + (n/2-1) * p_2$$

$$\iff (n/2-1) * d > (n/2-1) * p_1 + (n/2-1) * p_2$$

$$\iff d > p_1 + p_2$$

Let us now consider the case where n is odd. We can write:

$$(\lfloor n/2 \rfloor = \frac{n-1}{2} \;\; and \;\; (\lceil n/2 \rceil = \frac{n+1}{2}$$

Then the *worst-case condition* is:

$$2 * m_1 + (n-2)t_1 > (\frac{n-1}{2} + 1) * m_1 + (\frac{n+1}{2} - 1) * m_2$$

We replace t_1, m_1 and m_2 by their value:

$$2 * (p_1 + d + t_2) + (n - 2) * (d + t_2) > (\frac{n+1}{2}) * (p_1 + d + t_2) + (\frac{n-1}{2}) * (p_2 + t_2)$$

$$\Longleftrightarrow n * t_2 + 2 * p_1 + n * d > (\frac{n+1}{2}) * p_1 + (\frac{n+1}{2}) * d + (\frac{n-1}{2}) * p_2 + n * t_2$$

$$\Longleftrightarrow 2 * p_1 + n * d > (\frac{n+1}{2}) * p_1 + (\frac{n+1}{2}) * d + (\frac{n-1}{2}) * p_2$$

$$\Longleftrightarrow (n - (\frac{n+1}{2})) * d > ((\frac{n+1}{2}) - 2) * p_1 + (\frac{n-1}{2}) * p_2$$

$$\Longleftrightarrow (\frac{n-1}{2}) * d > (\frac{n-3}{2}) * p_1 + (\frac{n-1}{2}) * p_2$$

$$d > (\frac{n-3}{n-1}) * p_1 + p_2$$

We name as *longer(m_1)* the overall condition:

$$longer(m_1) = \begin{cases} d > p_1 + p_2 & \text{if } n \text{iseven} \\ d > (\frac{n-3}{n-1}) * p_1 + p_2 & \text{if } n \text{ isodd} \end{cases}$$

When the *worst-case condition* is true, T_l has to be taken into account in the WCET. When the paths have been measured, the labels (1, 2) can be instanciated into (T, NT) or (NT, T).

4.2 Repeated Loops

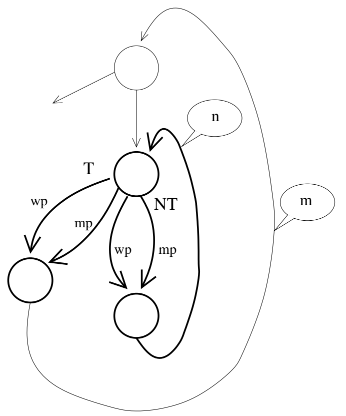

Figure 4.2: Control flow graph associated to a repeated loop

Let us consider a nested loop that iterates n times and is repeated m times (figure 4.2). This kind of construct has been studied in previous work and several cases have been tackled. First, Bate and Reutemann [5] 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 [7]. In this work, we assumed that the variation of the number of iterations was irregular. We have found that we could refine this worst-case number of mispredictions when the number of iterations was constantly increasing (or decreasing): the loop is executed m_0 times with 0 iteration, then m_1 times with 1 iteration, then m_2 times with 2 iterations and finally m_3 times 3 iterations or more (in this order or in the reverese order). It means that the loop is repeated $m = m_0 + m_1 + m_2 + m_3$ times. We have found this pattern in several benchmarks like the *four1* benchmark that includes the code given below, where the internal loop iterates once, then twice, then 4 times, 8 times, etc.

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

We want to compute a more accurate upper-bound on the number of mispredictions for this kind of loop. For this aim we use the table 4.1 which describes the evolution of the 2-bit counter [6]. Based on these results we derive a table for each number of iterations, as shown in Table 4.2: the loop is executed i_j times with j iterations.

Taking into account the worst-case number of mispredictions for each direction and each table, we can derive an expression of the overall worst-case number of mispredictions for a variable increasing (or decreasing) number of iterations:

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	$00 \rightarrow 00 \rightsquigarrow 01$	0/1	$00 \rightarrow 00 \rightarrow 00 \rightsquigarrow 01$	0/1	$00 \cdots \rightsquigarrow 01$	0/1
01	$01 \rightsquigarrow 10$	0/1	$01 \rightarrow 00 \rightsquigarrow 01$	0/1	$01 \rightarrow 00 \rightarrow 00 \rightsquigarrow 01$	0/1	$01 \rightarrow 00 \cdots \rightsquigarrow 01$	0/1
10	$10 \rightarrow 11$	0/0	$10 \rightsquigarrow 01 \rightsquigarrow 10$	1/1	$10 \rightsquigarrow 01 \rightarrow 00 \rightsquigarrow 01$	1/1	$10 \rightsquigarrow 01 \rightarrow 00 \cdots \rightsquigarrow 01$	1/1
11	$11 \rightarrow 11$	0/0	$11 \rightsquigarrow 10 \rightarrow 11$	1/0	$11 \rightsquigarrow 10 \rightsquigarrow 01 \rightsquigarrow 10$	2/1	$11 \rightsquigarrow 10 \rightsquigarrow 01 \cdots \rightsquigarrow 01$	2/1

Table 4.1: Number of mispredictions for both directions of a branch for a simple loop which iterates n times ($\rightsquigarrow$ represents a transition with misprediction)

$$\boxed{\begin{array}{c} m_T \leq MIN(2, i_3) + MIN(3, i_2) + i_1 \\ m_{NT} \leq i_3 + i_2 + i_1 + MIN(2, i_0) \end{array}}$$

Init State	$i_0 = 1$			$i_0 = 2$			$i_0 \geq 3$		
	Final State	m_{NT}	m_T	Final State	m_{NT}	m_T	Final State	m_{NT}	m_T
00	01	0	1	10	0	2	11	0	2
01	10	0	1	11	0	1	11	0	1
10	11	0	0	11	0	0	11	0	0
11	11	0	0	11	0	0	11	0	0
worst-case	–	0	1	–	0	2	–	0	2

(a) Final state of the 2-bit counter and number of mispredictions when the loop executes i_0 times with 0 iteration

Init State	$i_1 > 0$		
	Final State	m_{NT}	m_T
00	01	0	i_1
01	01	0	i_1
10	10	i_1	i_1
11	10	i_1	0
worst-case	–	i_1	i_1

(b) Final state of the 2-bit counter and number of mispredictions when the loop iterates once for i_1 successive executions

Init State	$i_3 > 0$		
	Final State	m_{NT}	m_T
00	01	0	i_3
01	01	0	i_3
10	01	1	i_3
11	01	2	i_3
worst-case	–	2	i_3

(c) Final state of the 2-bit counter and number of mispredictions when the number of iterations is greater than 3 for i_3 successive executions

Init State	$i_2 = 1$			$i_2 > 1$		
	Final State	m_{NT}	m_T	Final State	m_{NT}	m_T
00	01	0	1	01	0	i_2
01	01	0	1	01	0	i_2
10	01	1	1	01	1	i_2
11	10	2	1	01	3	i_2
worst-case	–	2	1	–	3	i_2

(d) Final state of the 2-bit counter and number of mispredictions when the number of iterations equals 2 for i_2 successive executions

Table 4.2: Number of mispredictions for both directions of a branch for a simple loop which is executed i_n times with n iterations

Chapter 5

Conclusion

In the case of high-performance pipeline, the impact of mispredictions is more variable. In this report we have proposed to refine previous work by taking into account a *per-direction* branch misprediction penalty. Conditional statements can lead to two different worst-case paths depending on their respective execution time. We have computed the *worst-case condition* which permits to identify which of these two paths has the worst case execution time. We have also detailed the calculation of the upper-bound on the number of mispredictions for repeated loops with an increasing (or decreasing) number of iterations.

Bibliography

[1] J. Gustafsson A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. In *Journal of Parrallel and Distributed Computing Practices*, volume 1, pages 61–74. 1998.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behaviour prediction by abstract interpretation. In *Static Analysis Symposium*, pages 51–66, september 1996.

[3] R. Arnold, F.Mueller, D.Whalley, and M.Harmon. Bounding worst-case instruction cache performance. In *15th IEEE Real-Time Systems Symposium (RTSS'94)*, pages 172–181, 1994.

[4] 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.

[5] 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.

[6] C. Burguière and C. Rochange. Analyse statique des erreurs de prédiction de branchement. Technical Report 2004-27-R, IRIT, september 2004.

[7] C. Burguière and C. Rochange. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Design, Automation and Test in Europe (DATE'05)*, march 2005.

[8] 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.

[9] ARM Corp. *ARM1020T Technical References Manual*, 2000.

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

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

[12] J. Fischer and S. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5^{th} International Conference on Architectural Support for Programming Langages and Operating Systems*, pages 85–95, october 1992.

[13] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. Van Engelen. Supporting timing analysis by automatic bounding of loop iterations. In *Real-Time Systems*, volume 18, pages 129–156, 2000.

[14] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for real-time analysis. In *Real-Time Systems*, volume 29, pages 27–58, january 2005.

[15] 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.

[16] Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *17^{th} IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263, 1996.

[17] D. Lindsay. *Statics Methods in Branch Prediction*. PhD thesis, Univ. Colorado, 1998.

[18] C. Park. Predicting program execution times by analysing static and dynamic program paths. In *Real-Time Systems*, volume 5, pages 31–62, 1993.

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

[20] P. Pushner and C. Koza. Calculating the maximum execution time of real-time programs. In *Real-Time Systems*, volume 1, pages 159–176, 1989.

[21] J. Smith. A study of branch prediction strategies. In *8^{th} annual ACM/IEEE international symposium on Computer Architecture*, 1997.

[22] T.-Y. Yeh and Y. Patt. Alternative implementation of the two-level adaptative branch prediction. In *19^{th} international symposium on Computer Architecture*, 1992.