Worst-Case Execution Time Analysis for Dynamic Branch Predictors

Iain Bate and Ralf Reutemann

Department of Computer Science, University of York
York, United Kingdom
e-mail: {ijb,ralf}@cs.york.ac.uk

Abstract

Branch prediction mechanisms are becoming commonplace within modern microprocessors. For developers of real-time control systems, the mechanisms present predictability problems. The reasons are they increase the difficulty in analysing software for its Worst-Case Execution Time without introducing unmanageable pessimism and they increase the variability of the software's execution times. In this paper, we improve upon existing branch prediction analysis by taking into account the semantic context of the branches in the source code in order to classify them as either being easy-to-predict or hard-to-predict. Based on this classification we provide a static analysis approach for bimodal and global-history branch prediction schemes. The analysis is applied to a previously published example with the benefit that a more detailed explanation of its results is obtained.

1. Introduction

In the domain of real-time systems the functional correctness of the computational results produced by a microprocessor is not the only important requirement. In addition the time when actions occur is of significant importance. The main problem from a timing perspective is to verify that a given set of tasks can be assigned to the available system resources such that all tasks are able to meet their respective deadlines. Existing scheduling schemes require an estimation of the *Worst-Case Execution Time* (WCET) of each task as a prerequisite for performing schedulability analysis of real-time programs. However, many commercially available microprocessors, which are usually not designed with real-time systems in mind, exploit micro-architectural features that prove detrimental to the degree of predictability required by schedulability analysis. Features such as instruction pipelining, out-of-order execution, caching, and dynamic branch prediction are key implementation techniques to achieve high microprocessor performance. The increased performance comes at the cost of greater variability in execution times. A number of methods to support static execution time analysis for microprocessors using such features have emerged in recent years. However, the effects of dynamic branch prediction on WCET analysis have received less attention.

Branch prediction is a technique supporting speculative execution, which is the execution of instructions across a branch before the outcome of the branch is known. Instead of stalling execution until the branch has been resolved, the microprocessor predicts the outcome of the branch as either being taken or not-taken and continues to fetch instructions along the predicted path. Consequently, all instructions issued after a branch are not allowed to change the machine state until the branch is resolved.

A simple approach to account for the timing effects of branch prediction would be to assume that all branches are mispredicted. With average branch prediction rates of over 90% [6] this would, obviously, result in unacceptably loose execution time estimates.

In order to limit the overestimation to a more acceptable value, the aim of this work is to provide tight but safe upper bounds on the number of branch mispredictions that can be expected for control statements. The work improves upon existing work by Colin and Puaut [3] on branch prediction. Particular attention is given to branch instructions that appear within loops, and thus are frequently executed, since these provide a promising potential for less pessimistic assumptions about branch misprediction penalties.

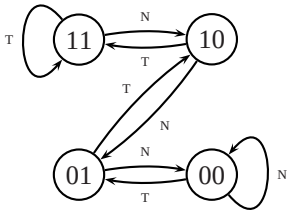
In this paper, Section 2 provides an introduction of dynamic branch prediction techniques. Then, Section 3 presents a review of the existing work on branch prediction analysis. Section 4 describes our analysis model for bimodal branch predictors. Section 5 extends the scope of the analysis to two-level branch predictors. Section 6 evaluates our analysis approach by using simulation results obtained from the SimpleScalar/PISA simulator. Finally, the conclusions are presented in Section 7.

2. Branch prediction techniques

Two basic categories of branch prediction techniques can be distinguished depending on whether or not past execution history is used for making a prediction: *static branch prediction* and *dynamic branch prediction* [11]. A detailed overview of various branch prediction techniques is provided in [6] and omitted here for space reasons.

The simplest dynamic technique is a bimodal branch predictor that stores branch history in a 2^n-entry branch history table (BHT), which is indexed by the n lower bits of the branch instruction address. One-bit predictors simply predict that a branch will always execute the same way next time they are evaluated. A misprediction results each time a branch changes its direction.

Consider a loop that is executed several times. A one-bit predictor causes a misprediction for the last iteration of the loop and another one when the loop is entered again. This is avoided by *two-bit predictors*, which implement a saturating two-bit up/down counter for each entry.



Bits	State	Prediction	Key
00	strongly not-taken	N	SN
01	weakly not-taken	N	WN
10	weakly taken	T	WT
11	strongly taken	T	ST

Figure 1. Two-bit prediction scheme

Figure 1 shows the states and transitions in a two-bit prediction scheme, which provides for some degree of hysteresis and thus is less affected by occasional changes in branch direction. The state of the counter is updated after the branch outcome has been resolved.

3. Related work

The original timing schema presented by Shaw [10] considers only high-level language constructs and does not address the effects of instruction pipelines and other micro-architectural features on execution time. This approach was appropriate at that time because most of the micro-architectural enhancements present in microprocessors today were not commonly used or even known about. The introduction of these features meant there was now a new need for low-level analysis.

The aim of low-level analysis is to model the different micro-architectural features being exploited in contemporary microprocessor architectures such that accurate bounds on WCET of basic blocks or individual instructions can be obtained.

Significant work has been performed on other parts of the low-level analysis, e.g. caches [9], but branch prediction has received relatively little attention.

The first work on execution time analysis for microprocessors using dynamic branch prediction techniques is presented by Colin and Puaut [3]. Their method is based on static program analysis and models the branch target buffer (BTB) of a Intel Pentium microprocessor in order to statically bound the number of timing penalties caused by branch mispredictions. They apply the idea of the classification scheme used in the static cache simulator [9] to worst-case analysis of branch prediction such that the worst-case impact on program execution time can be estimated. The purpose of this categorisation is to determine for each branch instruction whether or not it is present in the BTB at a particular execution instance. For conditional constructs, predictions based on the history in the BTB are always correct if the execution time difference between the two paths exceeds the branch prediction penalty. Otherwise, it is assumed that the branch associated with the conditional construct is always mispredicted.

More recent work by Mitra et al. [8] present a framework for modelling the effects of global-history branch prediction schemes on WCET analysis. Their approach uses integer linear programming (ILP) to bound the number of branch mispredictions by solving a set of linear constraints derived from the program's control-flow graph. Currently, they have only addressed history tables with one-bit predictor state entries in their analysis approach.

4. Static analysis of bimodal predictors

The purpose of this section is to develop a method for static timing analysis of bimodal branch predictors.

4.1. Overview of the approach

The method presented in this paper transforms an approach proposed previously by Lundqvist and Stenström [7] for data cache analysis to the area of WCET for dynamic branch predictors. There are a number of similarities between caches and dynamic branch predictions techniques. For example, the main problem of data cache analysis is to make data references predictable such that it can be decided whether a particular reference is present in the cache or not. While this is not possible for all data references, a relatively large number of references turn out to be predictable. Lundqvist and Stenström show that more than 84% of the data accesses are in fact predictable based on sample programs from the SPEC95 benchmark suite. In their approach, analysis is performed for predictable data references only and unpredictable references are excluded from the cache so they do not make the contents of the cache unpredictable. Effects of being unpredictable are not only the accesses have to be assumed to be not in the cache but also that potentially valuable data is displaced. By taking Lundqvist's approach, only the penalty for missing the cache is suffered.

Similarly to data caches, the outcome of conditional branches typically depends on input data that is not always known at compile-time. However, not only does the data reference need to be predictable in order to be able to determine the outcome of a conditional branch, but also knowledge of the data itself is necessary.

Similarly, our analysis method uses the semantic context of a branch as a criterion for the classification of branch instructions. Based on this classification, branches are either *easy-to-predict* or *hard-to-predict* in terms of static timing analysis. A branch instruction is classified easy-to-predict if there exists a static branch execution pattern and this pattern can be determined from the semantic context of the branch by means of static analysis. Otherwise, the branch instruction is classified as hard-to-predict.

4.2. Basic terminology and assumptions

In the following section we define the notion of WCET for the context of basic blocks. The timing schema originally proposed by Shaw [10] is used for low-level timing analysis of basic blocks and is modified to include the timing effects caused by branch mispredictions. Without loss of generality, we measure the WCET of a basic block in *clock cycles* rather than in a continuous time measure. The WCET in clock cycles of a basic block b_i is denoted by $T(b_i)$. A basic block b_i of a program is the longest continuous sequence of low-level instructions in memory such that the basic block is always entered at its first instruction and always exited at its last instruction.

Branch mispredictions interrupt the instruction fetch mechanism of the microprocessor and therefore cause a gap of several clock cycles between the basic block with the mispredicted branch instruction and the branch target block. Therefore, we associate the timing effects of mispredictions with the WCET of two basic blocks executed in sequence:

$$T(\{b_1, b_2\}) = T(b_1) + T(b_2) + \delta, \tag{1}$$

where $\delta \geq 0$ is the branch misprediction penalty, i.e. the difference between the WCET of the mispredicted and correctly predicted execution of the sequence.

With respect to other areas of WCET analysis we make the following assumptions:

1. A list of possible execution paths and upper bounds on the number of loop iterations are available from program flow analysis. Such information can either be provided by the programmer as manual program annotations [2] or is determined automatically [5].
2. Information about the timing of basic blocks and single instructions is available from pipeline analysis.
3. Different branch instructions do not interfere with each other, for example, by relocating branch instructions in the code such that no two branches share the same entry in the BHT.

Assumption (3) might be too restrictive in practice. Alternatively, a method for taking into account the effects of interference among different branch instructions on WCET analysis could be integrated into our analysis approach. However, this is considered outside the scope of this paper.

4.3. Analysis of simple loop constructs

In our first example we analyse the branch predictor behaviour of a simple loop construct that executes for n iterations. The static branch execution pattern for the branch that represents the loop condition is straightforward and given by $\langle T^{n-1}, N \rangle$. For $n = 1$ the branch execution pattern is simply $\langle N \rangle$.

We apply this branch pattern to the two-bit saturating counter defined in Figure 1 in order to determine an upper bound on the number of mispredictions that can be expected for different initial states of the predictor (see Table 1). The last column of this table shows the worst-case initial state for a given number of loop iterations. The behaviour of the predictor is the same for all $n \geq 4$. This is because the number of consecutive taken branches for the branch execution pattern $\langle T^{n-1}, N \rangle$ is at least three and after reaching the strongly taken state any subsequent taken branch does not alter the predictor state. Finally, the last not-taken branch changes the state to weakly taken.

n	Entry State				wc
	SN	WN	WT	ST	
1	0	0	1	1	ST
2	1	2	1	1	WN
3	3	2	1	1	SN
4	3	2	1	1	SN

Table 1. Number of branch mispredictions

We can observe from the results provided in Table 1 that the maximum number of mispredictions is given by:

$$mp_{loop}(n) = min(n, 3) \tag{2}$$

where n is the number of loop iterations.

Let us now assume that the loop construct itself is repeated m times and the number of loop iterations remains fixed for each repetition. The maximum number of mispredictions is now given by:

$$mp_{loop}(m, n) = m \cdot min(n, 3) \tag{3}$$

where m is the number of times the loop statement is repeated itself and n is the number of loop iterations.

This figure is pessimistic as it considers each loop repetition in isolation and assumes a worst-case initial predictor state for each repetition of the loop. In the following we will show how the associated pessimism can be reduced.

n	Entry State			
	SN	WN	WT	ST
1	SN	SN	WN	WT
2	SN	WN	WT	WT
3	WN	WT	WT	WT
4	WT	WT	WT	WT

Table 2. Exit states

In order to provide a less pessimistic upper bound on the number of mispredictions we now consider sequences of loop repetitions and use the predictor exit state of a repetition as the initial predictor state of the subsequent repetition. If the initial predictor state is unknown, e.g. because the loop is executed for the first time, we assume the worst-case state according to Table 1. Table 2 defines the exit states S_{exit} of a bimodal predictor depending on the initial state S_{in} and the number of loop iterations n.

We determine the upper bound on the number of mispredictions for m repetitions of the loop construct itself by concatenating the entry/exit states of each individual loop repetition, as illustrated in Table 3. In this table, the arrows represent a transition from the entry state to the exit state after n iterations of the loop. The sequence is interrupted once the entry/exit states no longer change. For each transition the number of branch mispredictions according to Table 1 is given and the last column of the table provides the maximum number of mispredictions for m repetitions of the loop construct.

n	Transitions	mp
1	ST $\xrightarrow{1}$ WT $\xrightarrow{1}$ WN $\xrightarrow{0}$ SN $\xrightarrow{0}$ SN	2
2	WN $\xrightarrow{2}$ WN $\xrightarrow{2}$ WN	$2m$
3	SN $\xrightarrow{3}$ WN $\xrightarrow{2}$ WT $\xrightarrow{1}$ WT	$3+m$
≥ 4	SN $\xrightarrow{3}$ WT $\xrightarrow{1}$ WT	$2+m$

Table 3. Repeated execution of the loop

The following summarises our observations and states a bound on the number of mispredictions depending on the number of iterations executed for each loop repetition.

Theorem 1 (Repeated loop)
Let n be the number of loop iterations and m be the number of times the loop is repeated. Then, the upper bound on the number of mispredictions for the repeated execution of a loop statement is defined by:

$$mp_{loop}(n,m) = \begin{cases} 2, & \text{if } n = 1 \\ 2m, & \text{if } n = 2 \\ 3+m, & \text{if } n = 3 \\ 2+m, & \text{if } n \geq 4 \end{cases}$$

Proof: For $n = 1$ the loop exists after its first iteration and therefore the loop control branch is always not-taken. The loop control branch is mispredicted only for the first two loop repetitions; any subsequent repetition results in a correct prediction. Hence, the upper bound on the number of mispredictions for $n = 1$ is two. The remaining three cases follow immediately from Table 3. □

It should be noted that for $n \geq 3$ the mispredictions stated in Theorem 1 are not equally distributed over the individual iterations of a loop. There are up to three additional mispredictions caused by the initial "warm-up" of the predictor state during the initial two repetitions of the loop. The additional mispredictions mean the first loop repetition always has the highest execution time among all repetitions. The actual number of mispredictions for any loop instance varies between one and three. If we have to guarantee a bound on the number of mispredictions that is valid for all iterations of a loop statement the jitter due to the initial mispredictions requires us to include a significant amount of pessimism in the WCET analysis. In this case, we have to assume that at most three mispredictions occur for each repetition of the loop construct.

4.4. Analysis of conditional constructs

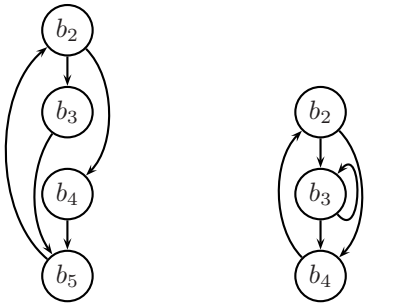
In our next example we derive an upper bound on the number of mispredictions for a conditional construct. The corresponding control-flow graph for this example is depicted in Figure 2(a).

The condition of the `if-then-else` statement depends completely on the provided input data, which is usually unknown at compile-time. Thus, it is not possible to determine the outcome of the branch instruction in b_2 statically. Before we tackle this problem, let us consider the two execution paths we have to include in the analysis.

Without taking into account the timing effects of branch mispredictions, the WCETs for the two possible execution paths $p_{\text{then}} = \{b_2, b_3, b_5\}$ and $p_{\text{else}} = \{b_2, b_4, b_5\}$ of the conditional statement in the `while`-loop are given by the following two equations:

$$T(p_{\text{then}}) = T(b_2) + T(b_3) + T(b_5) \quad (4)$$
$$T(p_{\text{else}}) = T(b_2) + T(b_4) + T(b_5), \quad (5)$$

where $T(b_i)$ is the WCET of the basic block b_i.



(a) Loop with an embedded conditional construct (b) Two nested loops

Figure 2. Control-flow graphs

Let us first consider the approach used by traditional WCET analysis methods that do not take into account branch prediction or caches. Such methods include only the path with the highest execution time for estimating the WCET of a conditional construct, i.e. for our example:

$$T(S_{\text{cond}}) = \max\Big(T(p_{\text{then}}), T(p_{\text{else}})\Big) \quad (6)$$

Let us now assume the case where the WCET of path p_{then} is greater than or equal to the WCET of path p_{else}:

$$T(p_{\text{then}}) = T(p_{\text{else}}) + \lambda, \quad \text{with } \lambda \geq 0$$
$$\Leftrightarrow \quad T(b_3) = T(b_4) + \lambda \quad (7)$$

According to Equation (6) we have only to include path p_{then} in the calculation of the overall WCET. Therefore, when we repeat the conditional construct for n iterations of the loop we obtain for its WCET:

$$T(S_{\text{loop}}) = nT(p_{\text{then}}) = n\Big(T(p_{\text{else}}) + \lambda\Big) \quad (8)$$

Bounding the timing effects that occur due to branch mispredictions for Equation (8) is straightforward. After

two iterations of the conditional construct the branch predictor state associated with the branch instruction in basic block b_2 is biased toward the not-taken direction. The initial mispredictions that occur in the first two iterations of the loop represent jitter caused by the warm-up of the predictor state. All subsequent instances of that branch are predicted correctly. Thus, the maximum number of mispredictions is two and we can now rewrite Equation (8):

$$T_1(S_{\text{loop}}) = n\Big(T(b_2) + T(b_3) + T(b_5)\Big) + 2\delta \qquad (9)$$

The branch instruction in basic block b_2 does not exhibit the worst-case behaviour of a bimodal branch predictor. The worst-case scenario in terms of branch prediction occurs when the execution of the loop body alternates between the p_{then} and p_{else} paths and the bimodal predictor alternates between the weakly taken and the weakly not-taken states. In this case, each instance of the branch instruction in basic block b_2 results in a branch misprediction. Let n be the total number of loop iterations. Without loss of generality, we assume that n is even. Then, the overall execution time of the loop construct S_{loop} for n iterations is given by the following equation:

$$\begin{aligned} T_2(S_{\text{loop}}) &= \underbrace{T(p_{\text{then}})}_{\text{1st iteration}} + \underbrace{T(p_{\text{else}})}_{\text{2nd iteration}} + \ldots \\ &= \frac{n}{2}T(p_{\text{then}}) + \frac{n}{2}T(p_{\text{else}}) \end{aligned}$$

$$= n\Big(T(b_2) + T(b_5) + \frac{1}{2}T(b_3) + \frac{1}{2}T(b_4)\Big) + n\delta \qquad (10)$$

Equation (10) implies that 100% of the branch instructions are predicted incorrectly (second term), which is overly pessimistic in most cases [6]. Repeating the process for *n* being odd also implies 100% of branches are mispredicted. On the other hand, Equation (9) limits the maximum number of branch mispredictions to only two, independent of the number of loop iterations. Consequently, the question arises as to when it is safe to use $T_1(S_{\text{loop}})$ rather than $T_2(S_{\text{loop}})$. In order to answer this question we try to find a condition for which the following inequality holds true:

$$T_1(S_{\text{loop}}) \geq T_2(S_{\text{loop}}) \qquad (11)$$

Substitution of $T_1(S_{\text{loop}})$ and $T_2(S_{\text{loop}})$ in Equation (11) finally yields:

$$T(b_3) \geq 2\delta(1 - \frac{2}{n}) + T(b_4) \qquad (12)$$

We use our assumption stated in Equation (7) and substitute $T(b_3)$ to obtain:

$$\lambda \geq 2\delta(1 - \frac{2}{n}), \quad \text{with } n > 1 \qquad (13)$$

Similar considerations apply for the case where $T(p_{\text{else}}) > T(p_{\text{then}})$. Thus, we can set $\lambda = |T(p_{\text{then}}) - T(p_{\text{else}})|$. Note that the condition in Equation (13) tends to 2δ for $n \to \infty$. If this condition is met we have to use $T_1(S_{\text{loop}})$ instead of $T_2(S_{\text{loop}})$ for calculating the overall WCET of the conditional statement nested in the loop. In this case, it is safe to consider only the path with the highest WCET in the calculation.

Theorem 2 (Repeated conditional statement)
Let n be the number of times the conditional statement is repeated within the loop body and δ be the misprediction penalty. Then, the upper bound on the number of mispredictions for the repeated execution of a conditional statement is defined by:

$$mp_{loop}(n, \lambda, \delta) = \begin{cases} 2, & \text{if } \lambda \geq 2\delta \\ n, & \text{if } \lambda < 2\delta \end{cases}$$

A similar approach is presented by Colin and Puaut [3] but they use $\lambda > \delta$ as condition instead. Furthermore, they do not account for jitter that is caused by initial mispredictions until the predictor has reached its steady state. The following example shows that their condition is not only less stringent but can in fact produce unsafe WCET estimates, i.e. although the condition is met and $T_1(S_{\text{loop}})$ is therefore used for the WCET calculation, $T_1(S_{\text{loop}}) > T_2(S_{\text{loop}})$ does not always hold true. In order to illustrate this let us assume that $T(p_{\text{then}}) = 16$, $T(p_{\text{else}}) = 10$, and $\delta = 5$ clock cycles. Their condition is clearly met since $T(p_{\text{then}}) - T(p_{\text{else}}) = 6 > 5$. However for $n > 5$ loop iterations, the WCET of the alternating sequence of the two paths is greater than the WCET that is taking only into account path p_{then}, since

$$T_2(S_{\text{loop}}) > T_1(S_{\text{loop}}) \Leftrightarrow 18n > 16n + 10 \Leftrightarrow n > 5.$$

5. Global-history predictors

This section continues the static timing analysis of dynamic branch predictors, but the scope of the analysis is now extended to more complex predictor configurations. Such predictors include two-level adaptive predictors that use global branch history for predicting the direction of branches. Global branch history records the outcome of all branch instructions recently executed in a single branch pattern. In contrast, local branch history uses the branch address to associate the branch pattern with a particular branch instruction or set of branch instructions.

We assume a two-level adaptive predictor using global branch history. The *branch history register* (BHR) of this GAg [12] predictor stores a pattern representing the outcome of recent branch instructions. The branch history register has a size of k bits, these bits are used to select one of the 2^k two-bit counters stored in the *pattern history table* (PHT). For a GAg predictor, both the branch history register and the pattern history table are the same for all static branch instructions. Thus, there is no direct mapping between the address of branch instructions and the predictor state, unlike the case for the bimodal predictor analysed in Section 4. This complicates the static analysis of branch predictor performance because the set of possible branch patterns has to be derived from the combined

static branch execution patterns of all branches executed recently. A static branch execution pattern exists if we can determine the behaviour of a branch from its semantic context, i.e. the branch is easy-to-predict.

Let us consider a simple nested `for`-loop construct in order to illustrate how the space of possible global history patterns can be explored. An inner loop is iterated a fixed number of times, n. We assume that the inner loop only contains sequential statements. Then, an outer loop continually repeats the inner loop. Figure 2(b) shows the corresponding control-flow graph for this example. In addition to the two loop-control branches for the inner and outer loop in b_3 and b_4, respectively, the conditional branch in b_2 examines the condition of the inner loop prior to its first iteration. The pattern shifted through the branch history register for each iteration of the outer loop is $\langle\, NT^{n-1}NT \,\rangle$. The length of this pattern is $n+2$.

$n=4$			$n=6$		
#	Pattern	*dir*	#	Pattern	*dir*
1	xxxx	N	1	xxxx	N
	xxxN	T		xxxN	T
	xxNT	T		xxNT	T
	xNTT	T		xNTT	T
	NTTT	N		NTTT	T
	TTTN	T		TTTT	T
2	TTNT	N		TTTT	N
	TNTN	T		TTTN	T
	NTNT	T	2	TTNT	N
	TNTT	T		TNTN	T
	NTTT	N		NTNT	T
	TTTN	T		TNTT	T
	...			NTTT	T
				TTTT	T
				TTTT	N
				TTTN	T
				...	

Table 4. History patterns for nested loop

Table 4 shows the resulting branch history patterns of length $k=4$ for $n=4$ and $n=6$ iterations of the inner loop. The first sub-column, #, indicates the current iteration of the outer loop. The second sub-column, *Pattern*, of each column in the table shows the current predictor history pattern prior to the execution of the branch instruction. The third sub-column, *dir*, represents the resolved outcome of the branch and the history pattern in the following line is updated accordingly. Like in Section 4, our aim is now to find conditions where we have biased predictor states, i.e. the outcome of branches is easy-to-predict at compile-time.

If we assume that the global history pattern is initially unknown, then the first k – in this case four – iterations of the inner loop produce history patterns where at least one bit is unknown (indicated by an `x`). These first k branches are all mispredicted because in the worst-case the two-bit counter of each pattern is initially in the strongly not-taken state and therefore at least two consecutive taken branches are required to predict the outcome of the taken branches correctly. For a not-taken branch, however, we assume that the predictor is in the strongly taken state. We refer to this type of history patterns as *unknown history patterns*.

After the unknown history patterns, only patterns with at most one consecutive zero occur in the global branch history register and the patterns repeat themselves after a number of iterations. In order to determine how many different history patterns exist we have to distinguish between two cases:

1. ($n \leq k$): In this case, there exist $n+2$ different branch patterns of length k, which we call *repeating history patterns*. There is a one-to-one correspondence between the history patterns and the branch instances. Therefore, branches are predicted correctly after the corresponding patterns occur twice and the maximum number of branch mispredictions is $2(n+2)$.
2. ($n > k$): The number of consecutive taken branches exceeds the length of the history pattern and the not-taken branches disappear from the history pattern for the last $n-k$ branch instances of the inner loop. There are now two types of history patterns: the *repeating history patterns* always record at least one not-taken branch and the *biased history patterns* record only taken branches. The latter pattern type $\langle\, T^k \,\rangle$ occurs for $n-k$ branch instances, which then share the same entry in the pattern history table of the two-level predictor. For this pattern type the behaviour of the predictor is equal to a bimodal predictor. The number of repeating history patterns is given by the difference between the total number of patterns and the number of biased history patterns, i.e. $n+2-(n-k)=k+2$, and the maximum number of mispredictions for this pattern type is $2(k+2)$.

Note that in the first case the global history predictor is also able to predict the not-taken branches correctly because a unique branch pattern exists for these branches.

Pattern	Frequency	Mispredictions
Unknown	k	k
Repeating	$\min(n,k)+2$	$2\cdot\min(n,k)+4$
Biased	$\max(0,n-k)$	$0, \quad n-k\leq 0$ $2, \quad n-k=1$ $2m, \quad n-k=2$ $3+m, \quad n-k=3$ $2+m, \quad n-k\geq 4$

Table 5. Pattern types for the nested loop

Table 5 summarises our findings about the three different types of history patterns that can occur during the execution of the nested loop discussed in this section. These results are applicable to nested loop constructs without any additional control structure embedded. For reasons of space, a proof to confirm the findings is not presented here. In order to derive an upper bound on the number of

mispredictions for the loop construct in our example we have to distinguish between several cases, which are broken down in the last column of Table 5. The upper bound on the number of mispredictions for the biased history pattern can be obtained by substituting n by the expression $n-k$ in Theorem 1 because this pattern uses a single entry in the PHT. The number of iteration of the outer loop is denoted by m.

We can then add the misprediction bounds for each of the three pattern types provided in Table 5 to state the following theorem:

Theorem 3 (Global history loop)
Let n be the number of loop iterations, k be the length of the global branch history table, and m be the number of times the loop is repeated. Then, the upper bound on the number of branch mispredictions for the repeated execution of a simple nested loop construct is given by:

$$mp_{loop}(n,k,m) = \begin{cases} 2n+k+4, & \textit{if } n-k \leq 0 \\ 3k+6, & \textit{if } n-k = 1 \\ 3k+2m+4, & \textit{if } n-k = 2 \\ 3k+m+7, & \textit{if } n-k = 3 \\ 3k+m+6, & \textit{if } n-k \geq 4 \end{cases}$$

Note from Theorem 3 that there is a significant increase in the upper bound on the number of mispredictions from $(n-k)=1$ to $(n-k)=2$. Furthermore, for $(n-k) \geq 2$ the misprediction bound is proportional to the number of iterations of the outer loop and is completely independent from the number of times the inner loop iterates.

6. Evaluation of the approach

Figures 3 and 4 show the simulation results obtained from the SimpleScalar `sim-outorder` simulator [1] for the nested loop construct in Figure 2(b). The original simulator has been extended to generate an execution trace of all branch instructions in order to be able to analyse branch predictor behaviour on a per-instruction basis. The execution time results have been extracted from the pipeline trace output. We have configured the simulator to ignore the effects of data and instruction cache access latencies so that the effects of branch prediction can be isolated. The simulation has been executed for three different dynamic branch predictor configurations: a bimodal predictor using a 4096-entry BHT, a two-level adaptive predictor using a single 12-bit BHR to index a global 4096-entry PHT, and a predictor based on the AMD Athlon implementation. The Athlon uses a global history two-level predictor that is combined with a 2048-entry BTB. The global predictor uses eight bits of branch history and four bits of the branch instruction address to index the PHT.

According to our analysis results presented in Section 4, the maximum number of branch mispredictions for the bimodal predictor occurs for an inner loop with two iterations. For more than two loop iterations the number of mispredictions remains constant. This is also indicated by the simulation results provided in Figure 4, which shows the number of mispredictions for the three predictor configurations. Accordingly, Figure 3 shows a significant increase in the number of clock cycles for two loop iterations and then a steady increase of the execution time for more than two iterations. The simulator has not been configured to resemble an existing microprocessor architecture, but the results presented here for the Athlon have been confirmed on a real Athlon platform using its performance counters.

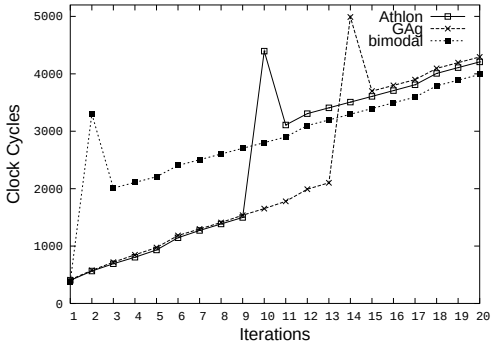


Figure 3. Execution time in clock cycles

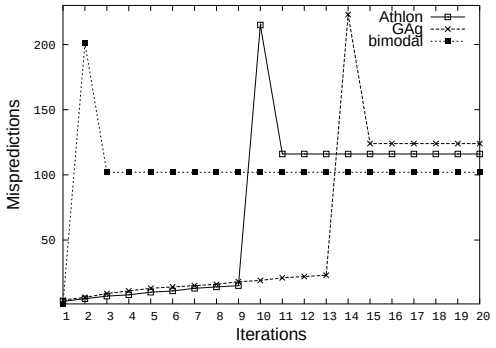


Figure 4. Number of branch mispredictions

The observation, above, that for $(n-k) \geq 2$ the misprediction bound is proportional to the number of iterations of the outer loop and is completely independent from the number of times the inner loop iterates supports the experimental results for the Intel Pentium III, AMD Athlon, UltraSparc III, and other microprocessors reported by Engblom [4]. He evaluates the execution times of a nested loop construct for different numbers of iterations of the inner loop. The number of iterations of the outer loop remains fixed. The results for the above mentioned microprocessors indicate that the average execution time per loop iteration decreases when the number of executed iterations increases. For the Athlon microprocessor, the total execution time rises significantly between the ninth and tenth iteration of the inner loop. Similar results are reported for the Pentium III and the UltraSparc III microprocessors where the rise appears after the fourth and thirteenth iteration, respectively.

Engblom's results [4] can be explained using Theorem 3. The maximum number of mispredictions can be expected for $(n-k)=2$, which would suggest a history

length of $k = n-2 = 10-2 = 8$, which in fact represents the predictor configuration of the Athlon microprocessor. Similar considerations apply to the Pentium III and UltraSparc III microprocessors.

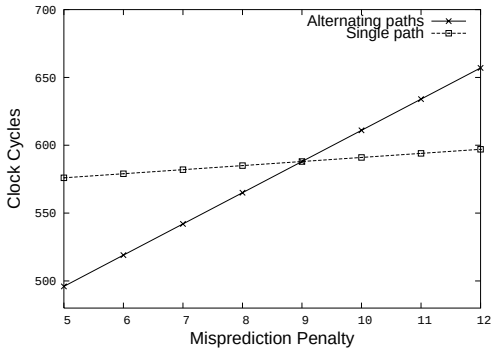


Figure 5. Conditional construct example

Figure 5 illustrates the simulation results for our second evaluation example – a loop construct containing a conditional statement as discussed in Section 4.4. The two graphs show the number of clock cycles required for executing only the `then`-path and both paths in alternation, respectively, depending on the *actual* misprediction penalty, i.e. the execution time difference between the correctly predicted and mispredicted paths.

Assuming $n = 20$ iterations of the loop and a constant execution time difference between the two paths of 17 cycles, it is necessary to consider the path with the highest WCET instead of the alternating paths if the condition stated in Equation (13) is fulfilled, i.e.:

$$2\delta(1 - \frac{2}{20}) \leq 17 \quad \Leftrightarrow \quad \delta \leq 9.4$$

7. Conclusions

In this paper, we have established the foundation of our static analysis approach for bimodal branch predictors. Instead of using the dynamic execution behaviour of branches to classify them the classification approach has been based on the semantic context of a branch.

Using this classification approach, an upper bound on the number of branch mispredictions has been derived for loop constructs and conditional statements. For the latter, the condition originally stated by Colin and Puaut [3] has been corrected.

The extension of our approach toward global history predictors has been addressed in the second part of this paper. The benefit of this extension is twofold. First, the analysis approach developed in Section 5 has been applied to an existing example presented by Engblom [4] and the results used to provide further explanation of his findings. Second, using our approach it is possible to reduce the complexity of the ILP problem of the method presented by Mitra et al. [8] by replacing a high number of constraints with the results of our analysis.

In general, we conclude that although static WCET analysis of dynamic branch predictors is feasible, not all branch predictor configurations are equally well suited. As far as static WCET analysis is concerned, developers should use static or bimodal branch predictors instead of more complex two-level predictor schemes. This is because the slight performance gain usually achieved by two-level predictors does not justify the additional complexity introduced to WCET analysis that is required to accurately model such predictors. It may also be possible that the performance gain is even outweighed by the additional pessimism caused by the wider scope of analysis.

Future work will extend our analysis to other control statements and integrate the approach into an overall WCET analysis framework including instruction pipeline and cache analysis.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] R. Chapman, A. Wellings, and A. Burns. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. In *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 135–148, 1994.

[3] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems Journal*, 18(2/3):249–274, 2000.

[4] J. Engblom. Effects of Branch Predictors on Execution Time. Technical Report 2002-013, Department of Computer Science, Uppsala University, Sweden, 2002.

[5] C. A. Healy, M. Sjödin, V. Rustagi, and D. B. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 12–21, Denver, Colorado, USA, 1998.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 3rd edition, 2002.

[7] T. Lundqvist and P. Stenström. Empirical Bounds on Data Caching in High-Performance Real-Time Systems. Technical Report 99-4, Chalmers University of Technology, Sweden, 1999.

[8] T. Mitra, A. Roychoudhury, and X. Li. Timing Analysis of Embedded Software for Speculative Processors. In *Proceedings of the 15th International Symposium on System Synthesis*, Kyoto, Japan, 2002.

[9] F. Müller, D. B. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, Orlando, Florida, USA, 1994.

[10] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[11] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, Minnesota, USA, 1981.

[12] T.-Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 257–266, San Diego, California, USA, 1993.