Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis

Iain Bate and Ralf Reutemann

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

Abstract

Advanced microarchitectural features such as caches and branch prediction mechanisms supporting speculative execution are becoming commonplace within modern microprocessors. For developers of real-time systems, these mechanisms present predictability problems. Previous work has demonstrated accurate analysis for instruction caches, data caches, and branch prediction mechanisms is possible. However, the integration of these individual analysis methods is difficult to do without large increases in computational complexity or the introduction of pessimism regarding the Worst-Case Execution Time (WCET) estimate. In this paper, we discuss how a previously published analysis method for branch predictors can be integrated with instruction pipeline analysis.

1. Introduction

Correct scheduling behaviour of tasks is fundamental in the design of a real-time system in order to fulfil its timing constraints and hence behavioural requirements. Existing scheduling schemes require an estimation of the *Worst-Case Execution Time* (WCET) of each task as a prerequisite for performing schedulability analysis of the task set. Modern superscalar microprocessors, however, exhibit micro-architectural features, such as instruction pipelines, caches and speculative execution, that trade a higher average-case processing performance for a lower predictability of execution time behaviour, and thus complicate the analysis of a program's WCET. The challenge in predicting the WCET is to find an estimate that is both *safe* and *tight*, i.e. the estimate is not lower than the actual WCET and it is reasonably accurate. In earlier work [2], a method is presented for obtaining the upper bounds on the number of branch mispredictions for loop constructs and conditional statements assuming bimodal or global-history branch predictors.

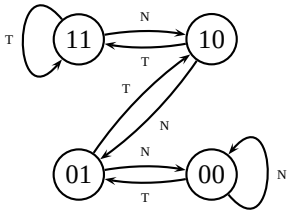
This paper discusses effects to be taken into account when a static analysis method for branch predictors is integrated with pipeline analysis. Ignoring such effects may either lead to unnecessarily pessimistic or even unsafe WCET estimates. Also, dealing with the effects in an inappropriate manner can lead to scalability problems.

The contribution of this paper is to derive an approach for integrating branch prediction analysis into an WCET calculation method based on *Integer Linear Programming* (ILP). Note that in the context of this paper no particular low-level WCET analysis method for instruction pipelines is assumed but instead an interface is defined for integrating it with branch prediction analysis.

2. Branch prediction techniques

Modern microprocessors combine the approach of out-of-order execution with *branch prediction* and *speculative execution* in order to try to alleviate the problem of disrupting the instruction flow into the pipeline due to branches.

A simple dynamic branch prediction technique is a *bimodal branch predictor* [9] 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.



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. N indicates a branch is *not-taken* and T that the branch is *taken*. The scheme provides 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.
Table 1, summarised from [2], details the maximum number of branch mispredictions that can be expected for various branch patterns being repeated m times, depending on the initial state of the predictor. The second column in this table defines for each branch pattern the classification of the associated branch instruction into *taken-biased* (T), *not-taken-biased* (N), and *alternating* (A). The worst-case number of mispredictions, i.e. each instance of a branch is mispredicted, is where the branch alternates between its *taken* and *not-taken* directions. In the table, the two cases corresponding to this worst-case predictor scenario are highlighted in bold. The last two rows represent the

Pattern	Class	Initial Predictor State			
		SN	WN	WT	ST
T^m	T	2	1	0	0
N^m	N	0	0	1	2
$(TN)^m$	A	m	**2m**	m	m
$(NT)^m$	A	m	m	**2m**	m
$(N^{n-1}T)^m,\ n=3$	N	m	m	$1+m$	$3+m$
$(N^{n-1}T)^m,\ n>3$	N	m	m	$1+m$	$2+m$
$(T^{n-1}N)^m,\ n=3$	T	$3+m$	$1+m$	m	m
$(T^{n-1}N)^m,\ n>3$	T	$2+m$	$1+m$	m	m

Table 1. Number of branch mispredictions

branch behaviour of a loop typically generated by a compiler for $n = 3$ and $n > 3$ iterations, respectively.

3. Related work

Several static analysis techniques for obtaining WCET estimates of real-time programs have emerged over the last two decades. The first work on execution time analysis for microprocessors using dynamic branch prediction techniques to model the branch target buffer (BTB) of a Intel Pentium microprocessor is presented by Colin and Puaut [3]. Disadvantages of their approach include it is based on source-code rather than object-code level which ignores additional branches introduced by the compiler, and it does not integrate the results with other parts of WCET analysis.

In our previous work [2], a scheme for classifying branch instructions as either being *easy-to-predict* or *hard-to-predict*, taking into account the semantic context of the branches in the source code is introduced. Based on this scheme, a static WCET analysis approach for dynamic branch prediction schemes using either local or global history is provided.

Mitra et al. [8] present a framework for modelling the effects of global-history branch prediction schemes on WCET analysis. Their approach uses ILP to bound the number of branch mispredictions by solving a set of linear constraints derived from the control-flow graph. As with all ILP-based calculation methods, the scalability has to be carefully managed, especially when dealing with more complex branch predictor configurations. While their focus is on modeling global-history predictors, they claim that their approach can also be adopted to local-history predictors. However, this has not yet been demonstrated. A key issue is the fact that their approach only deals with one-bit predictors and not bimodal predictors.

4. Integration of branch prediction analysis

In this section, we present an approach for combining a static WCET analysis method for branch predictors (e.g. as described in [2] or [3]) with instruction pipeline analysis in order to estimate the WCET using an ILP-based calculation method. The results of branch prediction analysis are used to define additional ILP constraints on the execution counts of transitions between basic blocks. The application of this approach is demonstrated through an example.

We analyse the WCET of a program at the level of *basic blocks* within the corresponding *control-flow graph*. Without loss of generality, we measure the WCET of a basic block in *clock cycles* rather than as a continuous time measure. A *branch block* is a basic block that contains a branch instruction. We denote the set of all branch blocks in a control-flow graph G(V,E) as the subset $V_c \subseteq V$. The transition $(b_i \rightarrow b_j) \in E$ represents the *taken* path and the transition $(b_i \rightarrow b_{i+1}) \in E$ the *not-taken* path of the branch instruction in a basic block $b_i \in V_c$.

For the program flow analysis, an approach is used based on *Implicit Path Enumeration Technique* (IPET), which models the control-flow of a program as a sequence of constraints on the execution count variables x_i (i.e. the number of times b_i is being executed) for each basic block in the control-flow graph [7]. The WCET of a program, in the following denoted by $T(G)$, can then be calculated by finding the maximum value of the following equation:

$$T(G) = \sum_{\forall i \bullet b_i \in V} x_i \cdot t_i \qquad (1)$$

where t_i is the maximum number of clock cycles required to execute b_i.

A major limitation of Equation (1) is that it does not take into account the pipeline overlap between instructions located across the boundary of basic blocks. Therefore, using this formula results in a significant overestimation of the WCET for microprocessors using instruction pipelines.

Li et. al. [6] address this problem by defining the execution time x_i of a basic block as the interval between the commit of the last instruction of the block preceding it and the commit of its last instruction. Unfortunately, they state that branch prediction modeling has not yet been included into their WCET analysis tool. Engblom et al. [4] propose to use a trace-driven microprocessor simulator instead of pipeline modelling to determine the execution time effect of pipeline overlap between successive instructions and basic blocks.

In the following, we modify the latter approach by defining the execution time of a basic block depending on its preceding blocks and possible branch misprediction events that might occur due to the transition between the blocks. For this purpose, we will use the execution count of edges instead of basic blocks for the final calculation of the WCET and show how bounds on the maximum number of branch mispredictions can be integrated into the overall WCET analysis.

Each edge $(b_i \rightarrow b_j) \in E$ in the control-flow graph is assigned with an variable d_{ij}, which represents the number of times this edge is being followed. Furthermore, for each basic block b_i that contains a branch instruction we can distinguish between the execution counts for mispredicted (denoted by *mp*) and correctly predicted (denoted by *cp*) outgoing transitions:

$$d_{ij} = d_{ij}^{mp} + d_{ij}^{cp}, \qquad (2)$$

with $\forall i, j \bullet (b_i \rightarrow b_j) \in E \wedge b_i \in V_c$,

Overlap between basic blocks. In general, the amount of overlap, δ_{ij}, across two adjacent basic blocks b_i and b_j is calculated as follows:

$$\delta_{ij} = t_{ij} - (t_i + t_j), \tag{3}$$

with $\forall i, j \bullet (b_i \rightarrow b_j) \in E$,

where t_{ij} is the number of clock cycles required to execute b_i and b_j in succession; t_i and t_j are the number of clock cycles required for executing b_i and b_j, respectively, in isolation.

We introduce a special vertex π in order to define a predecessor for the first executed basic block and set $t_{\pi 1} = t_1$ and $t_\pi = 0$. This will simplify the formal expression of the WCET calculation formula later in this section.

A misprediction of the branch instruction in basic block b_i reduces the amount of pipeline overlap between basic block b_i and the block b_j on its correct branch target. This may even result in a positive value for δ_{ij}, which would occur when there is no pipeline overlap but instead a pipeline stall. Also, it may have an impact on the overlap among instructions within basic block b_j and therefore change the execution time of b_j. Although Equation (3) implicitly takes into account a misprediction of the branch instruction in basic block b_i we distinguish between the mispredicted and correctly predicted case in order to provide a more accurate WCET estimate. Otherwise, potentially significant pessimism could be introduced. Thus, we calculate the pipeline overlap, δ_{ij}^{mp}, in the case of a branch misprediction according to the following equation:

$$\delta_{ij}^{mp} = t_{ij}^{mp} - (t_i + t_j^{mp}), \tag{4}$$

with $\forall i, j \bullet (b_i \rightarrow b_j) \in E \wedge b_i \in V_c$,

where t_{ij}^{mp} is the maximum number of clock cycles required to execute b_i and b_j in succession, and t_j^{mp} is the maximum number of clock cycles required for executing b_j taking into account the effects of a misprediction of the branch instruction contained in b_i.

Analysis of loop statements. The pattern $T^{n-1}N$ represents the behaviour of a branch instruction associated with a loop condition, assuming that the loop iterates n times. According to Table 1, an initial state of *strongly not-taken* (SN) is assumed as this equates to the worst-case initial state for a bimodal predictor.

Table 1 defines the maximum number of branch mispredictions, denoted by mp_i, that can be expected for the branch in b_i associated with the test of the loop condition. We can state the following linear constraint on mp_i:

$$mp_i = d_{ij}^{mp} + d_{i,i+1}^{mp} \tag{5}$$

In general, the *not-taken* path of the branch instruction in b_i is always mispredicted upon each loop exit, thus:

$$d_{i,i+1}^{mp} = m \tag{6}$$

Analysis of conditional statements. The behaviour of a branch instruction associated with a conditional statement cannot be determined statically if it depends solely on input data available only during run-time. In this case, we have to assume that all instances of the branch instruction are mispredicted, which according to Table 1 corresponds to an alternating behaviour of the branch. Thus, we can define the following linear constraints:

$$d_{ij}^{mp} = \left\lfloor \frac{n}{2} \right\rfloor \quad \wedge \quad d_{i,i+1}^{mp} = n - d_{ij}^{mp}$$

We assume here that the WCET of the *not-taken*-path exceeds that of the other path. Otherwise, d_{ij}^{mp} and $d_{i,i+1}^{mp}$ have to be interchanged with each other.

However, according to the condition stated in our previous paper [2], we need to consider the path with the highest WCET in the calculation (instead of the alternating paths case), when the execution time difference between the two paths of the conditional statement is at least twice the misprediction penalty. Then, the corresponding pattern of the branch is either N^n, i.e. always *not-taken*, or T^n, i.e. always *taken*, and according to Table 1, the maximum number of branch mispredictions is two:

$$d_{ij}^{mp} = 0 \quad \wedge \quad d_{i,i+1}^{mp} = 2$$

We will now introduce the general condition based on the theoretical model defined in [2]:

Definition 1 (Conditional Statement)
Let $n > 1$ be the number of times the conditional statement is repeated within the loop body and δ be the branch misprediction penalty. Then, the upper bound mp on the number of branch mispredictions is defined by:

$$mp(n, \lambda, \delta) = \begin{cases} 2, & \text{if } \lambda \geq 2\delta(1 - \frac{2}{n}) \\ n, & otherwise \end{cases},$$

with $\lambda = |T(p_{then}) - T(p_{else})|$.

$T(p_{\text{then}})$ and $T(p_{\text{else}})$ represent the WCET for the `then` and `else`-path, respectively. Note that the condition in the definition above tends to 2δ for $n \rightarrow \infty$.

The condition in the definition above needs to be evaluated for each conditional statement prior to establishing the set of ILP constraints for the WCET calculation.

WCET calculation. We can restate the ILP cost function provided in Equation (1) such that the effects of pipelining and branch prediction are taken into account:

$$T(G) = \sum_{\forall\ (i,j) \in E} \left(d_{ij}^{cp} \cdot (t_j + \delta_{ij}) + d_{ij}^{mp} \cdot (t_j^{mp} + \delta_{ij}^{mp}) \right) \tag{7}$$

This is a general calculation formula for estimating a WCET figure in the presence of an instruction pipeline and branch prediction, independent of how the individual analyses are actually performed. The scalability of the ILP

problem to account for more complex control-flow graphs is always of concern. For our approach, the increase in the number of ILP constraints is proportional to the number of branch blocks in the control-flow graph. This relationship is achieved by pre-determining the worst-case number of branch mispredictions. In other approaches, e.g. Mitra et al. [8], each branch and each possible execution pattern have to be modelled as a separate ILP constraint which leads to a much higher number of constraints. Although we acknowledge the principal reason for their approach is that they model a complex global-history predictor, the additional complexity associated with their analysis is not necessary when modeling a bimodal predictor.

In a recent work, Li et al. [5] describe a different approach for including the effects of mispredictions into the WCET calculation by extending Equation (1):

$$T(G) = \sum_{\forall i \bullet b_i \in V} (x_i \cdot t_i + mp_i \cdot mpp) \qquad (8)$$

where mp_i is the number of branch mispredictions for basic block b_i and mpp is the constant branch misprediction penalty for a single misprediction.

The branch misprediction penalty mpp in Equation (8) represents the number of stall cycles required by the instruction pipeline to recover from a mispredicted path. It should be noted, however, that the *actual* misprediction penalty, i.e. the execution time difference between a mispredicted and correctly predicted branch, not only includes the stall cycles but also the additional cycles required by the missing overlap in the case of a misprediction. Also, since the pipeline overlap usually varies for different combinations of basic blocks, the actual misprediction penalty is not constant and it may be too pessimistic to simply use its maximum value in Equation (8).

More importantly, only considering the number of stall cycles in Equation (8) underestimates the execution time overhead caused by mispredicted branches. The advantage of our approach is that it does in fact take into account the actual misprediction penalty for each basic block individually. Instead, it is included implicitly in Equation (1) without adding any unnecessary pessimism.

5. Branch Interference

In practice, the number of entries in the BHT of a bimodal predictor is of limited size, so different branch instructions may have to share the same predictor entry. This effect is called *branch interference* or *aliasing*.

As far as static WCET analysis is concerned, we need to address the effects of interference among branches because this may invalidate any assumption regarding the behaviour of the involved branches when analysed separately. Branch interference complicates static analysis because we have to widen the scope of the analysis such that the execution behaviour of multiple branch instructions can be taken into account.

In the context of WCET analysis, the problem of interference among branch instructions can be tackled from two different sides:

- Address the effect of branch interference in the static WCET analysis approach. This is the most desirable option in terms of completeness of the analysis model but it also complicates the analysis.

- Avoid branch interference by changing the instruction address of the affected branches, for example, by introducing `nop` instructions. Zhao et al. [10] discuss the repositioning of complete basic blocks in order to reduce the WCET. Although relocation of branches may not be feasible for all cases where branches interfere with each other, it may be beneficial (also from a performance point of view) to apply it if the interference may lead to worst-case predictor behaviour.

We propose to use the latter approach as it does not further complicate the complexity of the branch predictor analysis model itself, the first approach would be difficult to analyse without significant pessimism and it can also help to reduce the WCET in cases of destructive interference. In order to keep the number of branch instructions that have to relocated as small as possible we extend our branch classification model to identify branches being subject to destructive interference:

- *Hard-to-predict branches* are not relocated because these branches are already assumed to exhibit worst-case behaviour.

- *Easy-to-predict branches*, in contrast, are further classified into those being biased toward the taken direction and those being biased toward the not-taken direction. Then, all branches that both share the same entry in the BHT and are biased to different directions are relocated.

The number of branch instruction to be relocated can be further reduced, if necessary, by not considering interference among branches that are on mutually exclusive execution paths. The detailed algorithm for selecting and relocating branch instructions that are subject to interference is omitted here for space reasons.

6. Case Study and Evaluation

Figure 2 depicts the control-flow graph G(V,E) for a conditional statement that is embedded within a loop. We have omitted the source code because of space limitations. The set of branch blocks is defined by $V_c = \{b_2, b_3, b_5\}$. Basic blocks b_2 and b_5 contain conditional branches linked with the conditional statement and the loop condition, respectively, and b_3 contains an unconditional branch. The following set of linear constraints imposed by the program structure for the execution count of each basic block can

be observed from the control-flow graph:

$$\begin{aligned} x_1 &= d_{12} \\ x_2 &= d_{12} + d_{52} = d_{24} + d_{23} \\ x_3 &= d_{23} = d_{35} \\ x_4 &= d_{24} = d_{45} \\ x_5 &= d_{35} + d_{45} = d_{52} + d_{56} \\ x_6 &= d_{56} \end{aligned}$$

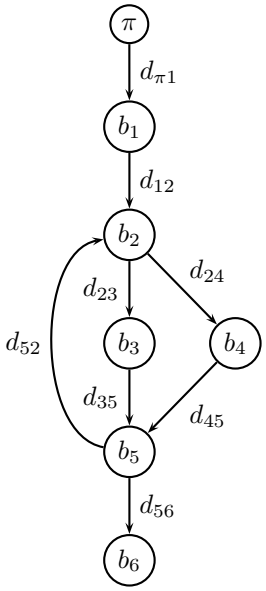


Figure 2. Annotated control-flow graph

Table 2 provides the execution counts d_{ij}^{cp} and d_{ij}^{mp} derived using the analysis results in Table 1, the execution time t_j of the basic block b_j, and the amount of overlap δ_{ij} between b_i and b_j for each transition $(b_i \rightarrow b_j) \in E$ in the control-flow graph. Although the execution time values in this table were obtained from the SimpleScalar architectural simulator [1], any other source of basic block execution time estimates could be used as well. For this example, we have configured the simulator to use an in-order instruction pipeline without caches.

$b_i \rightarrow b_j$	t_j	δ_{ij}	d_{ij}^{cp}	t_j^{mp}	δ_{ij}^{mp}	d_{ij}^{mp}	d_{ij}
$\pi \rightarrow b_1$	10	0	1	–	–	0	1
$b_1 \rightarrow b_2$	14	-9	1	–	–	0	1
$b_2 \rightarrow b_3$	29	-8	18	25	1	2	20
$b_2 \rightarrow b_4$	8	-7	0	5	1	0	0
$b_3 \rightarrow b_5$	11	-9	19	7	-1	1	20
$b_4 \rightarrow b_5$	10	-5	0	–	–	0	0
$b_5 \rightarrow b_2$	10	-5	17	8	-1	2	19
$b_5 \rightarrow b_6$	–	–	0	17	1	1	1

Table 2. Execution times and counts

Providing a loop bound is essential in order to make the ILP problem decidable. The loop in the control-flow graph can be identified by the fact that basic block b_2 dominates b_5 (i.e. all paths from the start vertex to basic block b_5 pass through b_2) and there is also a back edge $(b_5 \rightarrow b_2) \in E$ between these two blocks. Therefore, basic block b_2 represents the header of the loop, which we assume iterates $n = 20$ times. Furthermore, we assume that the function itself is executed only once, thus:

$$x_1 = 1 \quad \wedge \quad x_2 = 20 \cdot x_1 \quad \wedge \quad x_5 = x_2$$

The branch instruction in b_5, which is associated with the loop condition, has the pattern $T^{n-1}N$ and we assume in the following that $n > 3$. According to Equations (5) and (6), we can define two additional linear constraints:

$$d_{52}^{mp} = 2 \quad \wedge \quad d_{56}^{mp} = 1$$

We assume that the behaviour of the conditional statement is not known at compile-time and, therefore, we are not able to define exact values for the execution count variables x_3 and x_4 representing the two paths of the conditional statement. However, we can make reasonable worst-case assumptions about the behaviour of the branch by taking into account the condition stated in Definition 1.

If this condition is met we need to consider the path with the highest WCET in the calculation (instead of the alternating paths case), which in our example is the *then*-path represented by basic block b_3. In order to verify the condition we have to determine how much the misprediction penalty decreases the pipeline overlap between the blocks involved in the two paths. For this purpose, we also estimate the WCET *difference* between the mispredicted and non-mispredicted case for each of the two possible execution paths, which for our example is five clock cycles for each path:

$$\begin{aligned} T^{mp}(p_{\text{then}}) - T(p_{\text{then}}) &= (t_3^{mp} + \delta_{23}^{mp}) - (t_3 + \delta_{23}) \\ &= (25 + 1) - (29 - 8) = 5 \\ T^{mp}(p_{\text{else}}) - T(p_{\text{else}}) &= (t_4^{mp} + \delta_{24}^{mp}) - (t_4 + \delta_{24}) \\ &= (5 + 1) - (8 - 7) = 5 \end{aligned}$$

This execution time difference represents the *actual* penalty on the pipeline overlap caused by a branch misprediction in basic block b_2. It should be noted that for our example the actual penalties for the *then* and *else*-paths are the same but this is not necessarily the case in general. The actual penalty is slightly higher than the three cycle misprediction penalty purely associated with the branch misprediction. This is due to the fact the branch prediction also causes the pipeline to stall. The analysis shows that the condition stated in Definition 1 is fulfilled and therefore the upper bound on the number of mispredictions can be taken as two.

In this case, the corresponding pattern of the branch instruction in b_2 is N^n, i.e. always *not-taken*, and according to Table 1, the maximum number of mispredictions for this branch is two. Furthermore, we assume that the target address of the unconditional branch instruction in basic block

b_3 is mispredicted on its first execution. We can define the following additional ILP constraints for this case:

$$x_4 = 0 \quad \wedge \quad d_{35}^{mp} = 1 \quad \wedge \quad d_{23}^{mp} = 2$$

Considering branch interference. Based on the classification defined in Table 1 the branch instruction in b_5 is taken-biased while the branch in b_2 is not-taken-biased. Whilst the case study presented here is probably too small to exhibit extensive branch interference in practice, let us assume that the branches in basic blocks b_2 and b_5 are mapped to the same BHT entry and thus their predictor behaviour interferes with each other. In this case, the actual branch behaviour seen by the predictor alternates between the *taken* (due to b_5) and the *not-taken* (due to b_2) directions, which, in the worst-case, causes both branches to be always mispredicted. As there is a significant impact on performance due to this interference we would certainly want to remove it rather than model it.

WCET calculation. Using Equation (7) we can now calculate the total WCET of our example function from the values provided in Table 2 by using an ILP problem solving program (e.g. `lp_solve`). This gives us an estimated WCET of 606 clock cycles assuming a branch misprediction penalty of three clock cycles and $n = 20$ loop iterations. The pessimism of this WCET estimate compared with the measured execution time of 576 cycles is 5.2%.

The main reason for this overestimation is the use of maximum values for t_j, δ_{ij}, t_j^{mp}, and, δ_{ij}^{mp} for any transition $(b_i \rightarrow b_j) \in E$. Taking into account that these values may actually vary, depending on *some* basic block preceding b_i, would reduce the overestimation but increase the complexity of the WCET analysis significantly because it would require pipeline effects to be modelled along block sequences of arbitrary length. The amount of pessimism in this case is independent of the number of loop iterations due to a constant overestimation of the loop body.

If we assumed that the branch instruction in basic block b_2 alternates between its *taken* and *not-taken* directions and is always mispredicted (see Table 1) the estimated WCET would have been 526 clock cycles. In comparison with 576 clock cycles (see above), this figure does clearly not represent the actual WCET. Hence, simply assuming the worst-case number of branch mispredictions and an alternating behaviour of the branch instruction provides an unsafe WCET estimate in this case.

A conservative and simplistic approach would be to take into account both the maximum number of branch mispredictions and the longest execution path of the conditional statement. The ILP problem solving program automatically assumes this scenario if no constraints are provided on d_{23}^{mp} and d_{24}^{mp} (and hence on x_3 and x_4) and, as a result, increases the pessimism of the WCET estimate to 20.8%. In fact, this scenario is not even possible for a bimodal branch predictor because a branch instruction cannot always have the same outcome and be mispredicted at the same time.

7. Conclusions

The integration of individual WCET analysis methods is not straightforward due to the interaction between the analyses. This often leads to large increases in computational complexity, the introduction of unnecessary pessimism, or even to unsafe WCET estimates.

We have shown how a previously published approach for WCET analysis of dynamic branch predictors can be integrated with instruction pipeline analysis and the WCET be estimated using an ILP-based calculation method. This is achieved by first calculating the number of branch mispredictions and then representing these as constraints. Taking this approach results in significantly fewer constraints than for other approaches estimating misprediction numbers as part of the ILP problem. Hence the approach presented is more scalable.

References

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

[2] I. Bate and R. Reutemann. Worst-Case Execution Time Analysis for Dynamic Branch Predictors. In *Proceedings of the 16th Euromicro Conference on Real Time Systems*, pages 215–222, Sicily, Italy, 2004.

[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 and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Hong Kong, China, 1999.

[5] X. Li, T. Mitra, and A. Roychoudhury. Accurate Timing Analysis by Modeling Caches, Speculation and their Interaction. In *Proceedings of the 40th Conference on Design Automation*, pages 466–471, Anaheim, California, USA, 2003.

[6] X. Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for Software Timing Analysis. In *Proceedings of the 25rd Real-Time Systems Symposium (RTSS)*, pages 92–103, Lisbon, Portugal, 2004.

[7] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[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] 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.

[10] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET Code Positioning. In *Proceedings of the 25rd Real-Time Systems Symposium (RTSS)*, pages 81–91, Lisbon, Portugal, 2004.