Technical report IRIT/RR–2011-12–FR
–
A framework for the timing analysis of dynamic branch predictors

Claire Maïza
INP Grenoble, Verimag
Grenoble, France
claire.maiza@imag.fr

Christine Rochange
IRIT - CNRS
Université de Toulouse, France
rochange@irit.fr

Abstract

Real-time systems require predictable processor behavior such that tight upper bounds on the worst-case execution times (WCETs) of their critical tasks can be derived. Methods for estimating these upper bounds received much attention in the last fifteen years. However, the use of high-performance processors to meet ever growing performance requirements demands the development of more and more complex models. This is also the case for dynamic branch prediction schemes that are implemented in processors used in embedded systems. In this paper we focus on the static timing analysis of dynamic branch prediction based on the implicit path enumeration technique. The goal of this paper is to introduce a framework that allow to generate the IPET model of dynamic branch prediction mechanisms. With this framework we generated the models of some branch predictors and compare them.

1. Introduction

Real-time systems differ from other computer systems by the fact that respecting timing constraints is as important as providing correct results. In other words, the system not only has to deliver correct results, but must also deliver them in time. This must be proved by timing analysis. The objective is to assess that the longest execution time of each task makes it possible to build a task schedule that meets the constraints. When time-critical applications are considered, the timing analysis is expected to produce safe upper bounds of the execution times, also called *WCET*s, for Worst-Case Execution Times.

Techniques for WCET analysis have been investigated for more than twenty years but they always need improvement and extensions to support increasingly complex hardware (needed to fulfill ever growing performance requirements). For example, recent embedded processors implement dynamic branch prediction schemes. Their behavior is to be taken into account when computing WCETs. Several solutions have been proposed in the literature and will be described below. Our contribution in this paper is a general framework to derive models for dynamic branch predictors the most frequently available in embedded cores.

The static analysis of dynamic branch prediction can be *local*, i.e., each branch is considered in isolation and its behavior is determined by the algorithmic structure it implements , or *global*, i.e., all branches are analyzed conjunctly so that their possible interactions can be accounted for.

Local approaches [5, 3, 2] aim at computing the number of mispredictions for each branch by only considering its previous executions. Then, they are limited to simple dynamic branch predictors such as bimodal predictors (see Section 2) that do not rely on a global history of branches. Furthermore, the local approach is only used under the very optimistic assumption that the entries of the prediction table are not shared: The effect of sharing needs to be analysed separately, in a similar way of cache analysis, and in the case of conflicts, the branches are considered as mispredicted.

Global approaches describe the behavior of the branch predictor as a set of constraints [7, 4] that extend the ILP formulation used to estimate the WCET of the program with the IPET method [8]. The solution of the ILP problem includes the execution count of each branch and its number of mispredictions along both directions. Previous work using the global approach introduce a system of constraints to model one specific branch predictor: one based on local histories and 1-bit counters [7] and one based on a global history and 2-bit counters [4].

In this paper, we consider the global approach. The main contribution of the paper is a framework to model branch predictors in a generic way, which extends previous solutions [7, 4]. This framework allows modeling any branch predictor that uses a branch history table (BHT), 16 or 2-bit counters to compute the prediction and vari-

ous ways to index the BHT (address of the branch, local or global history, or any combination). It is structured in three parts, each one being related to a view of the BHT behavior: computation of the BHT index, computation of the interdependencies between branches sharing the same BHT entry, and computation of the prediction by prediction counters.

The framework allows to model a large number of BHT configurations. It is generic on:

- the size of the BHT,
- the branch history (local or global),
- the size of the branch history,
- the matching function used to index the BHT (for instance: exclusive-or),
- the size of prediction counters (1 bit or 2 bits).

Varying these parameters, we generate some models of branch predictors. The second contribution of the paper is an evaluation of the global approach based on the generated models. We use the modeling complexity notion [4] to compare the size of the models. From this comparison we conclude about the global approach and the branch predictors that fit better to be modeled by this approach. Note that the timing analyses of branch prediction using these two approaches are limited to architectures that do not exhibit timing anomalies [9], i.e., a misprediction is the worst case and can be accounted for by a fixed constant penalty.

The paper is organized as follows. Section 2 gives a short overview of dynamic branch prediction and lists the schemes that are implemented in some embedded processors. In Section 4 we introduce our framework and show how it can be used to generate the models of some dynamic branch predictors. The approach is evaluated in Section 5 and we give some concluding remarks in section 6.

2. Dynamic branch prediction

Branch prediction enhances the pipeline performance by allowing the speculative fetching of instructions along the predicted path after a conditional branch has been encountered and until it is resolved. If the branch was mispredicted, the pipeline is flushed and the correct path is fetched and executed.

The BTB (Branch Target Buffer) is used to (i) detect the presence of a branch instruction in the flow of fetched instructions before it has been decoded and (ii) to provide the branch target address in case the branch is predicted as *taken*. The BTB is indexed by the lower bits of the program counter (PC) and each entry it tagged with the branch address (PC) for verification.

Possible directions of a branch instructions are *taken* (usually coded by "1") or *not taken* (coded by "0"). The dynamic prediction of a branch direction is generally based on a counter that reflects the branch history. A 1-bit counter stores the last outcome of the branch. More often, a 2-bit saturating counter is used. Its four possible states are show in Figure 1: *strongly taken* (T), *weakly taken* (t), *weakly not taken* (nt) and *strongly not taken* (NT). The counter is incremented when the branch is taken and decremented otherwise. The next instance of the branch is predicted as taken whenever the upper bit is set [10] and not taken when it it cleared. Several prediction counters(to be used by different branches) are maintained in a BHT (Branch History Table).

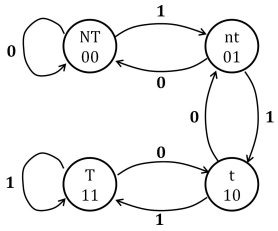


Figure 1. Behavior of a 2-bit branch prediction counter

Dynamic branch predictors differ by the way they index the BHT: the *bimodal* branch predictor uses the branch instruction address (PC) [10], while global branch predictors use an n-bit register that stores the outcomes of the n last executed branches [11]. For these schemes, the BHT can be directly indexed by the global history register (see Figure 2) or by a function (generally an exclusive-OR) of the history and the branch PC. This last scheme is known under the name *gshare*. A local history, private to each branch, can also be used.

Note that more complex schemes exist but we have not found them implemented in any embedded processor. Therefore, we do not consider them in this paper.

Unlike the Branch Target Buffer, the Branch History Table is usually not tagged to reduce costs. Two branches reaching the same entry of the BHT use the same prediction counter. Shared entries are more frequent when the BHT is indexed by a global history, even when this phenomenon – known as *aliasing* – is reduced by XOR-ing the history with the branch PC. Sharing can be advantageous (i.e. a branch takes advantage from the fact that its 2-bit counter has been updated by another branch) but it is more often destructive.

Core	BHT size	BHT Index	Prediction
ARM-CortexA8	4096	history and PC[a]	2-bit
ARM-11[b]	128	PC	2-bit
PowerPC-750	512	PC	2-bit
MIPS-34K	512	PC	2-bit

[a] 10 bits of history and 4 bits of PC used to compute the index
[b] unified BTB and BHT

Table 1. Branch predictors implemented in five embedded processors

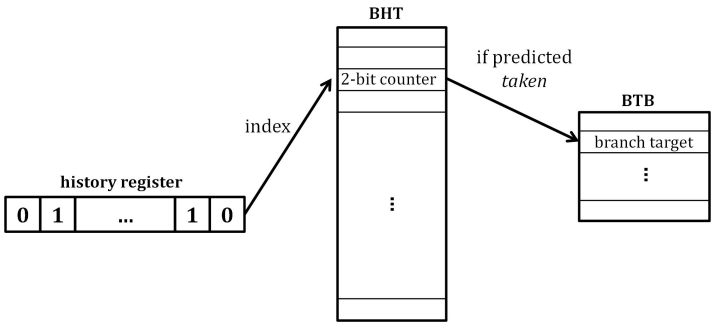


Figure 2. A global-history dynamic branch predictor

Table 1 shows the branch prediction schemes that are implemented in five embedded processors. The BHT is indexed by the branch PC in most of the cases. However, one processor implements a global-history branch predictor.

In this paper, we focus on this kind of dynamic predictors. The prediction is based on 2-bits counters, and the BHT is indexed by a function of the branch instruction address and the global history. In the remainder of this paper, we focus on the analysis of the BHT behavior. Modeling the BTB (that contains the target addresses for branches that are predicted taken) is out of the scope of this work. For this reason, we assume that each branch always hits in the BTB. As future work, we plan to combine our analysis to a BTB analysis such as [5, 6].

3. Modeling dynamic branch prediction by IPET

Throughout this paper, we use the following notation. A control flow graph is given by: $CFG = (\mathrm{X}, \mathrm{E}, st, en)$ where $\mathrm{X} = \{b_0, ..., b_n\}$ denotes the set of basic blocks b_i of the program and $\mathrm{E} \subseteq \mathrm{X} \times \mathrm{X}$ the corresponding set of edges connecting them. The entry node is denoted by *entry* and the exit node by *exit*. Furthermore, we denote the set of basic blocks containing a conditional branch by: $\mathrm{B} \subseteq \mathrm{X}$. We also use the set of predecessors of a block and the set of successors:
$\forall i \in \mathrm{X}, s \in \mathrm{S}_i \Leftrightarrow s \in \mathrm{X} \wedge (i,s) \in \mathrm{E}$
$\forall i \in \mathrm{X}, p \in \mathrm{P}_i \Leftrightarrow p \in \mathrm{X} \wedge (p,i) \in \mathrm{E}$
Finally, we use a label on the edges to mark the direction of the conditional branches. The direction of a conditional branch is taken (1) or not-taken (0). Label u is used for any edge that is not an outgoing edge of a conditional branch. Formally, we get:

$$\mathrm{D} = \{0,1\}, \mathrm{L} = \mathrm{D} \cup \{u\}$$

$$\forall e \in \mathrm{E}, \exists l \in \mathrm{L}, label(e) = l$$

$$\forall b \in \mathrm{B}, \forall s \in \mathrm{S}_b, label((b,s)) \in \mathrm{D}$$

Figure 3 shows a CFG that is used as an example in this paper. For this CFG, we have $\mathrm{X} = \{b_0..b_8\}$, $entry = b_0$, $exit = b_2$, $\mathrm{B} = \{b_1, b_3, b_5\}$, $\mathrm{S}_3 = \{b_4, b_5\}$, $label((b_3,b_4)) = 1$, $label((b_8,b_3)) = u$. Note that we do not write label u in the figure.

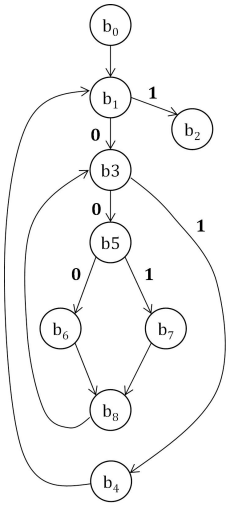


Figure 3. Example CFG

3.1. WCET analysis with the IPET

The IPET method [8] aims at determining the longest execution path and computing an upper bound on the WCET. It considers the control-flow graph of the task under analysis. A set of integer variables stands for the execution counts of the nodes (basic blocks) and edges. The execution time of the task can be expressed by $T = \sum_{i \in \mathrm{X}} x_i t_i$, where x_i and t_i are the execution count and the worst-case execution time of basic block b_i, respectively. Estimating an upper bound on the WCET consists of maximizing this expression under a set of linear constraints. Flow constraints express the control flow structure of the task, i.e., the relations that must exist between the execution counts of nodes (x_i for basic block b_i) and edges ($x_{i \xrightarrow{d} j}$ for the edge from b_i to b_j labeled with direction d – in some equations, we will use the equivalent notation $x_{i \xrightarrow{d}}$). The flow constraints are:

$$\forall i \in \mathrm{X}, x_i = \sum_{p \in \mathrm{P}_i} x_{p \xrightarrow{l} i} + init_i$$
$$\forall i \in \mathrm{X}, x_i = \sum_{s \in \mathrm{S}_i} x_{i \xrightarrow{l} s} + end_i$$
$$\sum_{i \in \mathrm{X}} end_i = 1$$
$$\sum_{i \in \mathrm{X}} init_i = 1$$

Note that $end_{exit} = 1$ and $\forall i \in \mathrm{X}, i \neq exit \Rightarrow end_i = 0$. Similarly, $init_{entry} = 1$ and $\forall i \in \mathrm{X}, i \neq entry \Rightarrow init_i = 0$.

From the CFG illustrated in Figure 3, the flow constraints for block b_3 are:

$$x_3 = x_{1 \overset{0}{\to} 3} + x_{8 \overset{u}{\to} 3} + ent_3$$

$$x_3 = x_{3 \overset{0}{\to} 5} + x_{3 \overset{1}{\to} 4} + ex_3$$

To bound the execution counts, the system needs some additional constraints which we call *semantic* constraints. They express loop bounds, infeasible paths, etc. For our example CFG, the system should least include the following semantic constraints:

$$x_{1 \overset{0}{\to} 3} \leq MAX_1 \times x_{0 \overset{u}{\to} 1}$$

$$x_{3 \overset{0}{\to} 5} \leq MAX_3 \times x_{1 \overset{0}{\to} 3}$$

where MAX_n is a loop bound, i.e. the maximum number of iterations of a loop, that depends on the semantic of the program.

An upper bound on the WCET can be obtained using Integer Linear Program solvers like `lp_solve`[1]. The solution is returned as the set of values of the execution counts that maximize the task execution time and satisfy all constraints.

3.2. Integrated analysis of dynamic branch prediction

The IPET method computes the overall execution time of a program by adding the execution times (costs) of the basics blocks weighted by their respective execution count. To take the impact of branch prediction into account, the expression of the execution time is extended with an additional cost for each block that is the target of a conditional branch whenever it is mispredicted.

This cost includes all the effects of branch prediction on the pipeline state. In this paper, we ignore the impact of branch prediction on cache memories (considered as perfect[2]).

Note that a conditional branch is always the last instruction of a basic block. In the rest of the paper, for the sake of simplicity, $b \in \mathrm{B}$ is used to denote both a basic block and the conditional branch it ends with (if any).

Variables $m_{b \overset{d}{\to}}$ and $m_{b \overset{1}{\to}}$ denote the number of mispredictions for branch $b \in \mathrm{B}$ when it follows direction $d \in \{0,1\}$ (0 is for *not-taken*, 1 for *taken*) while $v_{b \overset{d}{\to}}$ is the related cost. The overall execution-time can be expressed by:

$$T = \sum_{i \in \mathrm{X}} x_i t_i + \sum_{j \in \mathrm{B}} \sum_{d \in \{0,1\}} m_{j \overset{d}{\to}} v_{j \overset{d}{\to}}$$

Besides the addition of branch misprediction penalties to the objective function, additional linear constraints should express the behavior of the branch predictor. These constraints can be generated in a methodical way through the generic framework we introduce in the next section. They bound the worst-case number of mispredictions for both directions of each branch $b \in \mathrm{B}$.

[1] http://lpsolve.sourceforge.net/5.5/

[2] Perfect cache means that each access to the cache is a hit.

4. A framework to model dynamic branch predictor using IPET

In this section, we describe our framework. We show how the behaviour of any branch predictor is modeled by the IPET using the global approach.

When the behaviour of the branch predictor at a given point in the program is to be determined, three questions should be answered.

The first one is: *Which entry of the BHT is indexed to predict the branch?* Formally, the set of all indexes is denoted by $\Lambda = [0.2^n]$ where n is the size of the BHT. For each branch $b \in \mathrm{B}$, Λ_b is the set of possible indexes of the BHT that may be used to predict this branch. The index $\lambda_b \in \Lambda_b$ may depend on the state π_b of the global history and on the branch instruction address @: $\Lambda_b = \{\lambda_b | \exists \pi_b \in \Pi_b, \lambda_b = f(@, \pi_b)\}$ where $\Pi_b \subseteq \Pi$ is the set of possible values of the global history at b (Π is the set of all values of the global history). Note that @ is usually not the whole branch instruction address but only a significant part of it. For a bimodal predictor, the index is computed directly from the branch instruction address: $\Lambda_b = f(@, \pi_b) = \{@\}$. In the case of gshare, the mapping function f is an exclusive-OR: $\Lambda_b = \{\lambda_b | \exists \pi_b \in \Pi_b, \lambda_b = @ \oplus \pi_b\}$. The set of equations related to this first question is given in Section 4.1. Note that these equations are not derived for predictors that do not use a history register (e.g. bimodal).

The second question is: *Which other branches may access the same BHT entry?* Remember that the capacity of the BHT is limited and it is not tagged with branch addresses: some branches may share the same BHT entry. Such branches may update the 2-bit counter according to their own behaviour and thus influence the prediction. Equations that express the aliasing to the BHT and the order in which competing branches access to the shared BHT entry are given in Section 4.2.

The third question is: *What may be the 2-bit counter's state and thus the predicted direction?* Some equations are required to express the possible behaviours of the 2-bit counter stored in the indexed BHT entry. They are derived in Section 4.3.

Finally, all these equations must be related to build the whole dynamic branch prediction model. This is explained in Section 4.4 The framework being generic, the size of the BHT, the size of the history, and the index function are formally described.

4.1. BHT index

In this Section, we show how the evolution of the global history can be modeled by a system of constraints. A global history is a n-bit register. The global history is updated when the branch issue is known ("1" for taken, "0" for not-taken). The update consists of a left shift of the $n-1$ rightmost bits and an insertion of the issue of the last executed branch to the rightmost position. For instance $\pi = 0100$ is the new value of the global history

when $\pi' = 1010$ was the previous value and the last branch is solved as "not-taken". The update of the history is denoted by $\pi = \pi'.d$: the state π is the successor of the state π' (left shift and insert d to the rightmost position). The value of the global history at a given point p in a program depends on the initial value of the global history and on the issue of each executed branch along the path from the *entry* node to p.

We introduce a graph to model the evolution of the global history. It depicts the value of the global history depending on the initial value and the issue of previous branches.

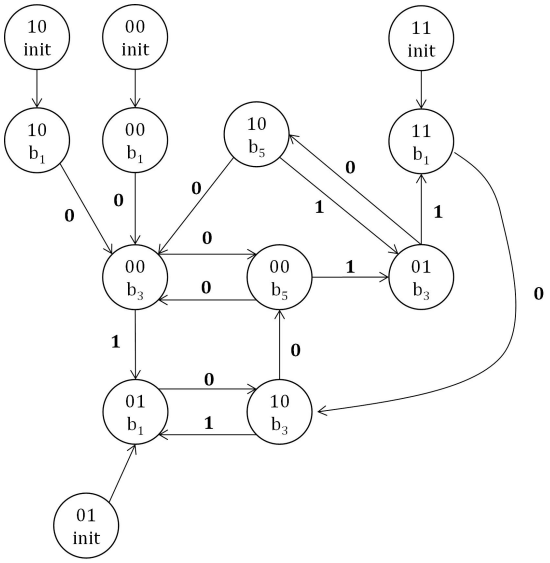


Figure 4. Example: BHG.

The Branch History Graph (BHG) expresses the overall evolution of the global history for a program. Each node of the BHG fits with a pair (b_i, π) where b_i is a basic block of the CFG containing a conditional branch and π is one possible value of the global history at this branch. The nodes labelled "init" target the branches that may be the first branch executed. For the sake of simplicity, end nodes are not shown on this graph and on the following ones. The BHG can be automatically built from the CFG. Figure 4 shows the BHG built for the CFG of Figure 3.

From the BHG, we derive a set of equations that will be included in the IPET formulation. The value of the global history depends on the issue of the last executed branches: we use $p \xrightarrow{d} b$ to denote that $p \in \mathrm{B}$ is the last branch to be executed before $b \in \mathrm{B}$ and d was the direction of branch p (used to update the global history).

One execution of the program under analysis consists of one path in the BHG from a node (b, π), where b is the first executed branch and π the initial value of the branch history, to a node (b', π') where b' is the last executed branch and π' the last value of the history. Flow constraints are derived for each node of the BHG: they link the executions counts of blocks with a given history register value (x_b^π) to the execution counts of their in- and out-edges ($x^\pi_{b \xrightarrow{d}}$). Furthermore, if b is the first executed branch and π the initial value of the global history then $init_b^\pi = 1$. Similarly, $end_b^\pi = 1$ if b is the last executed branch. The set of predecessors of a node (b, π) in the BHG is denoted by P_b^π.

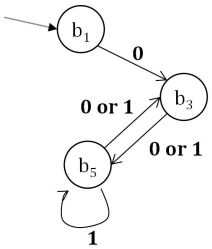


Figure 5. Example: Execution order between branches sharing the entry 00 of the BHT (Branch Conflict Graph)

$$
\begin{array}{l}
\forall b \in \mathrm{B}, \forall \pi \in \Pi_b, \\
\quad x_b^\pi = \sum_{p \in \mathrm{P}_b^\pi} \sum_{\pi' \in \Pi} x^{\pi'}_{p \xrightarrow{d}} + init_b^\pi \\
\qquad \text{where } \pi = \pi'.d \text{ and } p \xrightarrow{d} b \\
\forall b \in \mathrm{B}, \forall \pi \in \Pi_b, \\
\quad x_b^\pi = x^\pi_{b \xrightarrow{0}} + x^\pi_{b \xrightarrow{1}} + end_b^\pi \\
\sum_{b \in \mathrm{B}} end_b^\pi \leq 1 \\
\sum_{b \in \mathrm{B}} init_b^\pi \leq 1
\end{array}
$$

In [7] the value of global history is computed for each basic block of the CFG. As this history may only evolve after a conditional branch, we have enhanced this part of the global approach by only taking into account the basic blocks that contain a conditional branch.

4.2. Execution context

The BHT is a $|\Lambda|$-entry table. Each branch b may access $|\Lambda_b|$ 2-bit counters during the execution of the program. For a given $\lambda \in \Lambda_b$, the prediction of the branch depends on all branches sharing this entry. B^λ denotes the set of branches that may access entry λ of the BHT: $\mathrm{B}^\lambda = \{b \in \mathrm{B} | \lambda \in \Lambda_b\}$. The prediction counter at λ is updated at each execution of one branch accessing it. Then, the prediction depends on the access sequence to the entry λ. In other words, the prediction of a branch accessing the entry λ depends on the execution sequence of branches in B^λ. To model the sharing, we introduce a new graph. It depicts the possible execution sequences between branches in B^λ.

A Branch Conflict Graph (BCG^λ) can automatically be derived for each BHT entry and contains as many blocks as the number of branches sharing this entry ($|\mathrm{B}^\lambda|$). Figure 5 shows $BCG^{\lambda=00}$ for the example program ($\mathrm{B}^{\lambda=00} = \{B_1, B_3, B_5\}$).

Each BCG is described by IPET: One execution of the program under analysis may consist of at most one path in

a BCG from an initial node to a final node. The init edges of a BCG^λ target the possible initial nodes: Branches that may be the first executed with the index value λ ($init_b^\lambda$ = 1 if b is the first executed branch with λ). Similarly, the end edges (not shown) start on the final nodes: Branches that may be the last executed with this history value (end_b^λ=1 if b is the last executed branch with λ). The set of constraints corresponding to BCG^λ links the execution order between branches sharing the λ-th entry of the BHT. P_b^λ (resp. S_b^λ) denote the set of predecessors (resp. successors) of b in BCG^λ.

$$\forall b \in B^\lambda, \quad x_b^\lambda = \sum_{s \in S_b^\lambda} (x^\lambda_{b \xrightarrow{0} s} + x^\lambda_{b \xrightarrow{1} s}) + end_b^\lambda$$
$$\forall b \in B^\lambda, \quad x_b^\lambda = \sum_{p \in P_b^\lambda} (x^\lambda_{p \xrightarrow{0} b} + x^\lambda_{p \xrightarrow{1} b}) + init_b^\lambda$$
$$\sum_{b \in B^\lambda} end_b^\lambda \leq 1$$
$$\sum_{b \in B^\lambda} init_b^\lambda \leq 1$$

4.3. Prediction computation

All branches in B^λ may be predicted by the counter in the λ-th entry of the BHT. The prediction is given by the state c of the counter ($c \in C$ and $C = \{00, 01, 10, 11\}$) and the update of the counter is done according to the update of a saturating 2-bit counter. We introduce a new graph representing the evolution of the 2-bit counter depending on the initial state and the possible transitions.

A Prediction Graph (PG^λ) is built for each prediction counter (each entry of the BHT) and expresses the possible evolution of this counter. Figure 6 shows $PG^{\lambda=00}$ for the example program in case the BHT is indexed by the global history. As in the previous steps, this graph is described by a system of constraints. Note that the set of possible transitions to reach (resp. leave) a state depends on the set of possible successors (resp. predecessors) in the related BCG^λ. One execution of the program under analysis consists of at most one path in a PG: from the initial state to the final state.

$$\begin{aligned}
&\forall b \in B^\lambda, \\
&x_b^{\lambda,00} = \sum_{p \in P_b^\lambda} (x^{\lambda,00}_{p \xrightarrow{0} b} + x^{\lambda,01}_{p \xrightarrow{0} b}) + init_b^{\lambda,00} \\
&x_b^{\lambda,00} = \sum_{s \in S_b^\lambda} (x^{\lambda,00}_{b \xrightarrow{0} s} + x^{\lambda,00}_{b \xrightarrow{1} s}) + end_b^{\lambda,00} \\
&x_b^{\lambda,01} = \sum_{p \in P_b^\lambda} (x^{\lambda,00}_{p \xrightarrow{1} b} + x^{\lambda,10}_{p \xrightarrow{0} b}) + init_b^{\lambda,01} \\
&x_b^{\lambda,01} = \sum_{s \in S_b^\lambda} (x^{\lambda,01}_{b \xrightarrow{0} s} + x^{\lambda,01}_{b \xrightarrow{1} s}) + end_b^{\lambda,01} \\
&x_b^{\lambda,10} = \sum_{p \in P_b^\lambda} (x^{\lambda,01}_{p \xrightarrow{1} b} + x^{\lambda,11}_{p \xrightarrow{0} b}) + init_b^{\lambda,10} \\
&x_b^{\lambda,10} = \sum_{s \in S_b^\lambda} (x^{\lambda,10}_{b \xrightarrow{0} s} + x^{\lambda,10}_{b \xrightarrow{1} s}) + end_b^{\lambda,10} \\
&x_b^{\lambda,11} = \sum_{p \in P_b^\lambda} (x^{\lambda,10}_{p \xrightarrow{1} b} + x^{\lambda,11}_{p \xrightarrow{1} b}) + init_b^{\lambda,11} \\
&x_b^{\lambda,11} = \sum_{s \in S_b^\lambda} (x^{\lambda,11}_{b \xrightarrow{0} s} + x^{\lambda,11}_{b \xrightarrow{1} s}) + end_b^{\lambda,11} \\
&\sum_{b \in B^\lambda} \sum_{c \in C} end_b^{\lambda,c} \leq 1 \\
&\sum_{b \in B^\lambda} \sum_{c \in C} init_b^{\lambda,c} \leq 1
\end{aligned}$$

A misprediction occurs each time a fired transition does not match to the prediction given by its source state, e.g., leaving the state 00 by a transition labeled 1.

$$\begin{aligned}
m^\lambda_{b \xrightarrow{1}} &= \sum_{s \in S_b^\lambda} (x^{\lambda,00}_{b \xrightarrow{1} s} + x^{\lambda,01}_{b \xrightarrow{1} s}) \\
m^\lambda_{b \xrightarrow{0}} &= \sum_{s \in S_b^\lambda} (x^{\lambda,10}_{b \xrightarrow{0} s} + x^{\lambda,11}_{b \xrightarrow{0} s})
\end{aligned}$$

In [7] Li et al. consider a 1-bit counter. In this case, the evolution does not need to be modeled by additional constraints because the counter state is the issue of the last executed branch. The number of mispredictions is given by:

$$\begin{aligned}
m^\lambda_{b \xrightarrow{1}} &= \sum_{s \in S_b^\lambda} x^0_{b \xrightarrow{1} s} \\
m^\lambda_{b \xrightarrow{0}} &= \sum_{s \in S_b^\lambda} x^1_{b \xrightarrow{0} s}
\end{aligned}$$

In the remainder of this paper, we only focus on 2-bit counters which are the most frequently used due to their highest accuracy.

4.4. Model of branch predictor

In this subsection we describe the whole framework. First, we focus on the generation of the graphs: branch history graph, branch conflict graphs and prediction graphs. Then, we show how to build the whole system of constraints: it is based on the three subsets of constraints described in previous subsections and on some additional constraints that are necessary to create a link between the three subsets and the IPET from the CFG. Finally, we summarize how to use the framework: what are the parameters and an example of configuration.

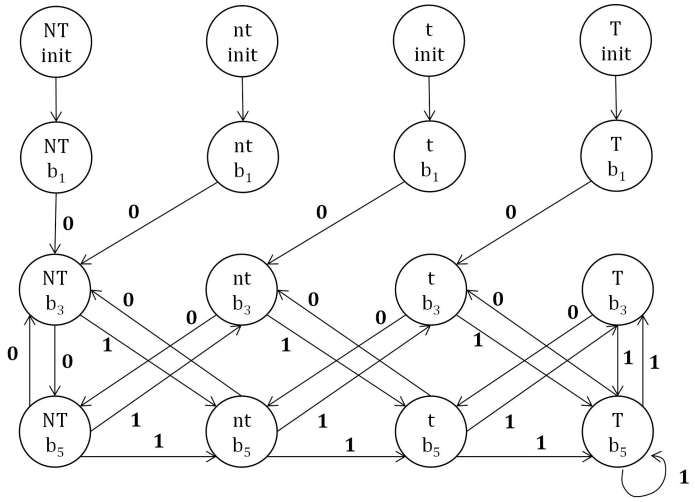


Figure 6. Example: 2-bit counter evolution for $\lambda = 00$ (PG^{00}, history-indexed BHT)

x_b	CFG
x_b^{π}	BHG
x_b^{λ}	BCG^{λ}
$x_b^{\lambda,c}$	PG^{λ}

Table 2. Variable used for the number of occurences of a block in the different graphs.

Graph generation All the graphs on which the generation of constraints is based on are generated from the CFG. The BHG is given by a partition of the CFG block: The BHG contains only the basic blocks with two outgoing edges (conditional branch).

Each BCG is a partition of the BHG or of the CFG. BCG^{λ} is built from:

- all BHG blocks (b,π) with $\lambda = f(\pi)$, in case a global history is used,
- all CFG blocks ending by a conditional branch which BHT index is $@_b = \lambda$ for the bimodal predictor.

From each BCG^{λ}, one PG^{λ} is built: for each edge in BCG^{λ} the corresponding transitions are generated in the related PG^{λ}.

The whole framework In the rest of this section, we explain how to build the whole system of constraints and list the additional constraints generated for that purpose. Table 2 reminds the notation for the number of occurences of a block depending on the corresponding graph.

A BHG is needed when a history is used to index the BHT. If the histories are local, then a BHG is built for each history register[3]. In case of a global history, one BHG is generated. In a BHG, more than one block (b,π) may refer to the same block b in the CFG. The set of constraints generated to link the subsystems related to the CFG and the BHG is:

$$\begin{array}{l}\forall b \in \mathrm{B}, \quad x_b = \sum_{\pi \in \Pi_b} x_b^{\pi} \\ \forall b \in \mathrm{B}, \forall d \in \{0,1\}, \quad x_{b \xrightarrow{d}} = \sum_{\pi \in \Pi_b} x^{\pi}_{b \xrightarrow{d}} + end^{\pi}_{b \xrightarrow{d}} \\ \forall b \in \mathrm{B}, \forall \pi \in \Pi_b, \quad end_b^{\pi} = end^{\pi}_{b \xrightarrow{0}} + end^{\pi}_{b \xrightarrow{1}}\end{array}$$

Note that accounting for the last branch in the BHG does not include the last issue of this branch. However, in the CFG this last issue is needed to obtain the correct number of occurences of the related edge in the CFG.

One BCG^{λ} is generated for each $\lambda \in \Lambda$. The $|\Lambda|$ subsets of constraints derived from those BCGs are linked to the whole system of constraints by:

$$\begin{array}{l}\forall b \in \mathrm{B}, \forall \lambda \in \Lambda_b \\ \quad x_b^{\lambda} = x_b^{\pi} \quad where \quad \lambda = f(\pi, @) \\ \forall b \in \mathrm{B}, \forall d \in \{0,1\}, \\ \quad x_{b \xrightarrow{d}} = \sum_{\lambda \in \Lambda_b} \sum_{s \in \mathrm{S}_b^{\lambda}} x^{\lambda}_{b \xrightarrow{d} s} + end^{\lambda}_{b \xrightarrow{d}} \\ \forall b \in \mathrm{B}, \forall \lambda \in \Lambda_b, \quad end_b^{\lambda} = end^{\lambda}_{b \xrightarrow{0}} + end^{\lambda}_{b \xrightarrow{1}}\end{array}$$

Note that in case of a bimodal branch predictor, $x_b^{\pi} = x_b$ and $f(\pi, @) = @$.

For each BCG^{λ} one PG^{λ} is generated.

$$\begin{array}{l}\forall b \in \mathrm{B}, \forall \lambda \in \Lambda_b \quad x_b^{\lambda} = \sum_{c \in \mathrm{C}} x_b^{\lambda,c} \\ \forall b \in \mathrm{B}, \forall \lambda \in \Lambda_b, \forall s \in \mathrm{S}_b^{\lambda}, \quad x^{\lambda}_{b \xrightarrow{d} s} = \sum_{c \in \mathrm{C}} x^{\lambda,c}_{b \xrightarrow{d} s} \\ \forall b \in \mathrm{B}, \forall \lambda \in \Lambda_b, \quad end_b^{\lambda} = \sum_{c \in \mathrm{C}} end_b^{\lambda,c} \\ \forall b \in \mathrm{B}, \forall \lambda \in \Lambda_b, \quad init_b^{\lambda} = \sum_{c \in \mathrm{C}} init_b^{\lambda,c} \\ \forall b \in \mathrm{B}, \forall d \in \{0,1\}, \quad m_{b \xrightarrow{d}} = \sum_{\lambda \in \Lambda_b} m^{\lambda}_{b \xrightarrow{d}}\end{array}$$

Note that if $b \in \mathrm{B}^{\lambda}$ is the first (resp. last) executed with λ then $init_b^{\lambda} = \sum_{c \in \mathrm{C}} init_b^{\lambda,c} = 1$ (resp. $end_b^{\lambda} = \sum_{c \in \mathrm{C}} end_b^{\lambda,c} = 1$).

[3] Note that in case of pattern history table (PHT) containing local histories, some branches may share a PHT entry: A BHG is derived for each entry of the PHT.

Configuration of the framework The global approach was introduced by Li *et al.* [7]. The first model was for a *1-bit* global predictor whose BHT is indexed by the branch history. Their model accounts for interferences due to BHT entries sharing. This model corresponds to the following configuration of our framework:

- branch history: global
- index of the BHT: $\lambda = f(\pi, @) = \pi$
- size of the prediction counter: 1 bit

In [4], a model of a branch predictor based on *2-bit* counters was introduced and we used it to build the models of two 2-bit branch predictors: a bimodal predictor (PC-based indexing) and a global 2-bit predictor whose BHT is indexed by a global branch history. This model corresponds to the following configuration of our framework:

- branch history: global
- index of the BHT: $\lambda = f(\pi, @) = \pi$ or $\lambda = f(\pi, @) = @$
- size of the prediction counter: 2 bits

In this paper, we present the model in a generic way that eases the understanding of the set of constraints. Furthermore, this framework allows to model any branch predictor which would use the BHT. The configuration of the framework depends on:

- size of the BHT
- size of the history
- branch history: global or local
- index of the BHT: $\lambda = f(\pi, @)$
- size of the prediction counter: 2 bits or 1 bit

Note that the index may be any index that could be generated from the branch instruction address and a branch history. Furthermore, they may be more than one history if a PHT is used.

The framework is implemented in OTAWA: a WCET computation tool [1]. For a given branch predictor, this tool may compute the WCET by integrating the constraints in the whole ILP system of constraints. It may also generate separately the system of constraints for the branch predictor model that could be added in any other WCET tool. This only requires to use the same variables for the CFG nodes.

5. Experimental results

In this paper, we introduce a framework that helps in generating models for a variety of dynamic branch predictors. These models are to be included in the IPET formulation of WCET computation to analyse the behavior (quality of the prediction of branch directions).

In this section, our goal is to give an insight into how complex the generated models can be. In other words, we evaluate the global approach by comparing some models generated by our framework. With this information the end-user may retain or reject a possible target core with dynamic branch prediction, depending on the resources (including time) he is ready to allocate to timing analysis. This may be useful when the development processes rely on very short steps for new code generation and analysis between two test phases. Before this, we report some bounds on the WCET that we evaluated for some benchmark codes. They show that taking dynamic branch prediction into account when performing WCET analysis is really needed to avoid the overestimation of considering every conditional branch as mispredicted. Then, we show how to quantify the size of the models we generate,

Using the framework, we have generated models for three branch predictors based on 2-bit counters. They differ in how the BHT is indexed. The first one ($BP_{@}$) is the so-called bimodal predictor that selects a BHT entry from the branch address (direct-mapping scheme: $\lambda = f(\pi, @) = @$). The second one (BP_H) uses the global history (built from the outcomes of the last executed branches) as an index to the BHT ($\lambda = f(\pi, @) = \pi$). The third one ($BP_{@\oplus H}$), known as gshare, computes the index to the BHT by XOR-ing some bits of the global history and some bits of the branch PC ($\lambda = f(\pi, @) = \pi \oplus @$). For the three of them, we assume a 16-entry Branch History Table (choosing such a small size was dictated by the small size of our benchmarks). For the two last ones, we consider a 4-bit global history. Bits 4 to 7 of the branch instrction address are used to index the BHT when the branch predictor is bimodal or gshare.

Our experiments consider a processor with a 2-way superscalar 4-stage pipeline that implements out-of-order execution. We assume that the system includes SRAM memories for instructions and data, with a short (1-cycle) latency. The models for dynamic branch predictors have been implemented in OTAWA, our WCET computation tool [1]. Implementing a branch predictor model consists in adding specific constraints to the basic IPET formulation. We consider a set of benchmarks that belong to the SNU-RT collection[4] and are frequently considered for the evaluation of WCET analysis techniques. They are listed in Table 3.

5.1. Impact of branch prediction modeling on the WCET

Table 4 shows how modeling dynamic branch prediction improves the accuracy of WCET estimates: taking into account the behavior of the branch predictor instead of considering each branch as mispredicted. Results are provided for each of the three predictors considered in this paper. For most of the benchmarks, taking dynamic branch prediction into account improves the estimated WCET by 5% to 10%, and the improvement even reaches more than 25% for the `fibcall` program. The weakest results are for the `jfdctint` program that contains only three branches executed in sequence.

These results show how much important it is to model dynamic branch prediction when the WCET accuracy is critical. However, modeling branch prediction has a cost in terms of time required to perform WCET analysis time.

[4] http://archi.snu.ac.kr/realtime/benchmark/

Benchmark	# bbs	# bchs	Function
fibcall	7	1	Summing the Fibonacci series
insertsort	8	2	Insertion sort for 10 integer numbers
jfdctint	13	3	JPEG slow-but-accurate integer implementation of the forward DCT (Discrete Cosine Transform)
matmul	19	5	Matrix multiplication
qurt	74	21	Root computation of quadratic equations

Table 3. Benchmarks

	$BP_{@}$	BP_{H}	$BP_{@\oplus H}$
fibcall	-27.7%	-25.0%	-27.7%
insertsort	-10.9%	-10.1%	-9.1%
jfdctint	-4.5%	-3.7%	-3.5%
matmul	-8.1%	-7.8%	-6.9%
qurt	-11.2%	-5.4%	-9.8%

Table 4. Impact on the estimated WCET

This will be shown below and should be considered when selecting a processor with dynamic branch prediction.

5.2. Estimating the modeling complexity

In the domain of computer science, the notion of complexity often refers to the number of steps required to solve a problem using a given algorithm. For example, the complexity of solving an Integer Linear Program (ILP) is the time it takes to compute the values of the variables when maximizing (or minimize) the objective function as a function of the problem size. Naturally, this time depends to a large extent on the algorithm implemented by the ILP solver that is used. This is why our objective is mainly to estimate the complexity of the model in terms of the number of variables and constraints. It is expected that the modeling complexity will turn into a timing complexity to solve the problem but there might be some differences according to the solving tool.

Table 5. Sets used in modeling-complexity functions. A is the set of useful indexes in case the predictor is bimodal ($@ \in \mathrm{A}$).)

B	Set of basic blocks that ends with a conditional branch
D	Set of branch directions $\mathrm{D} = \{0,1\}$
B^G_{init} / B^G_{end}	Set of branches that can be the first/last executed in the related graph ($BHG:\ G = \pi,\quad BCG^{\lambda}:\ G = \lambda$)
S^G_b / P^G_b	Set of successors/predecessors of branch $b \in \mathrm{B}$ in the related graph ($BHG:\ G = \pi,\quad BCG^{\lambda}:\ G = \lambda$)

To express the modeling complexity of a branch prediction scheme, we specify a triplet $< \mathbf{C}, \mathbf{V}, \mathbf{A} >$ where $\mathbf{C}$ is the number of constraints of the ILP formulation, $\mathbf{V}$ is the number of variables and $\mathbf{A}$ the arity of the formulation. The arity of a constraint is the number of variables involved in this constraint. The arity of the formulation is the maximum arity among all the constraints. In [4], we have shown that these three components of the complexity triplet can be computed from characteristics related to the program code under analysis. Tables 5 shows all data used to express the modeling complexity. Table 6, 7 and 8 present the modeling-complexity functions for the three branch predictors (bimodal, history-based indexes, pc-xor-history-based indexes). These modeling-complexity functions show what influences the modeling-complexity variation. To analyse the modeling-complexity variation, we compare modeling complexity of different models. Results are presented in the remainder of the section.

Table 6. Modeling complexity of the PC-indexed BHT (bimodal: $\lambda = @$)

$\mathbf{C} = \sum_{b \in \mathrm{B}} (14 + 4\lvert \mathrm{S}^{\lambda}_b \rvert) + 2\lvert \mathrm{A} \rvert$
$\mathbf{V} = \sum_{b \in \mathrm{B}} (12\lvert \mathrm{S}^{\lambda}_b \rvert + 6) + \sum_{\lambda \in \mathrm{A}} (4\lvert \mathrm{B}^{\lambda}_{init} \rvert)$
$\mathbf{A} = 1 + 4 \times \max_{\lambda \in \mathrm{A}} (\lvert \mathrm{S}^{\lambda}_b \rvert)$

Table 7. Modeling complexity of the history-indexed BHT (2-bit global history, $\lambda = \pi$)

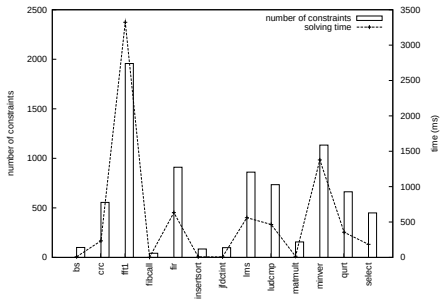
$\mathbf{C} = \sum_{b \in \mathrm{B}} (9 + 12\lvert \Pi_b \rvert + 4 \sum_{\pi \in \Pi_b} \lvert \mathrm{S}^{\pi}_b \rvert) + 2\lvert \Pi \rvert + 2$
$\mathbf{V} = \sum_{b \in \mathrm{B}^{\pi}_{end}} \lvert \Pi_b \rvert + \sum_{b \in \mathrm{B}^{\pi}_{init}} \lvert \Pi_b \rvert + 4 \sum_{\pi \in \Pi} \lvert \mathrm{B}^{\pi}_{init} \rvert + \sum_{b \in \mathrm{B}} (2 + \sum_{\pi \in \Pi_b} (9 + \lvert \mathrm{S}^{\pi}_b \rvert))$
$\mathbf{A} = 1 + \max_{b \in \mathrm{B}, d \in \mathrm{D}} (\sum_{c \in \mathrm{C}} \sum_{\pi \in \Pi_b} \lvert \mathrm{S}^{\pi}_b \rvert)$

Table 8. Modeling complexity of the "history-xor-pc"-indexed BHT (2-bits global history, $\lambda = @ \oplus \pi$)

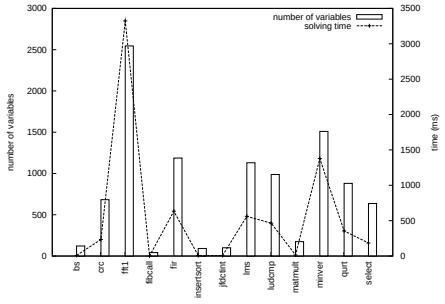
$\mathbf{C} = \sum_{b \in \mathrm{B}} (9 + 2\lvert \Pi_b \rvert + 13\lvert \Lambda_b \rvert + 4 \sum_{\lambda \in \Lambda_b} \lvert \mathrm{S}^{\lambda}_b \rvert) + 2\lvert \Lambda \rvert + 2$
$\mathbf{V} = \sum_{b \in \mathrm{B}^{\pi}_{end}} \lvert \Lambda_b \rvert + \sum_{b \in \mathrm{B}^{\pi}_{init}} \lvert \Lambda_b \rvert + 4 \sum_{\lambda \in \Lambda} \lvert \mathrm{B}^{\lambda}_{init} \rvert + \sum_{b \in \mathrm{B}} (2 + \sum_{\lambda \in \Lambda_b} (9 + 12\lvert \mathrm{S}^{\lambda}_b \rvert) + 3\lvert \Pi_b \rvert)$
$\mathbf{A} = 1 + \max_{b \in \mathrm{B}, d \in \mathrm{D}} (\sum_{c \in \mathrm{C}} \sum_{\lambda \in \Lambda_b} \lvert \mathrm{S}^{\lambda}_b \rvert)$

5.3. Impact of the modeling complexity on the ILP solving time

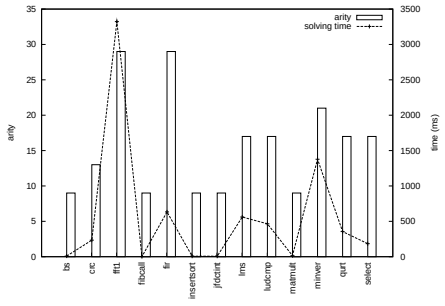
In Figure 7, the solving time of the model of a bimodal branch predictor with a 16-entry BHT is confronted with each component of the modeling complexity. We observe that they all follow the same shape. This clearly suggests that the modeling complexity has an impact on the time needed to solve the integer linear program.



(a) number of constraints



(b) number of variables



(c) arity

Figure 7. Modeling complexity vs solving time

5.4. Modeling complexity of dynamic branch predictors

Table 10 shows the the modeling complexity of three dynamic branch predictors. For all the benchmarks, the bimodal predictor ($BP_{@}$) is the one that exhibits the lower complexity by far. This is due to the absence of constraints that model the changes on the global history (since this predictor does not use it). On the contrary, the history indexed predictor (BP_H) has the largest complexity due to the fact that several branches may share the same BHT entry. This occurs less often with the gshare ($BP_{@\oplus H}$) predictor that enhances the distribution of branches over the table entries. As a result, the complexity of $BP_{@\oplus H}$ is generally lower than that of BP_H. An exception is for `fibcall` that has a very small number of branches. Table 11 shows the mean number of branches that share a BHT entry, which quantifies the so-called phenomenon of *aliasing*. The bimodal ($BP_{@}$) and gshare ($BP_{@\oplus H}$) predictors generally exhibit less conflicts than the pure global history-based predictor (BP_H). It appears that the program with the highest rate of conflicts (`qurt`) is the one that exhibit the highest complexity by far. Our measurements also show that this program is the one that necessitates the longest time to solve the ILP problem ($\sim$ 200 ms while the other programs needed $\sim$ 0.2 ms for the bimodal predictor).

Table 12 gives the modeling complexity of a global branch predictor (BP_H) when considering two sizes for the prediction history register: 2 bits and 4 bits. Whatever the quantifier (C, V or A), the complexity significantly increases with the history register length.

The results presented above highlight the factors that influence the modeling complexity: the indexing scheme to the BHT, the length of the history register, the average number of branches sharing an entry in the BHT. Indexing the BHT with the global history register (BP_H) generates a high complexity, and this is expanded by a larger history register. Using an XOR operator ($BP_{@\oplus H}$) to combine the history and the branch PC reduces the aliasing phenomenon and clearly reduces the complexity.

We claim that these results should be kept in mind when selecting a processor with dynamic branch pre-

Benchmark	**# cons. (C)**	**# vars (V)**	**arity (A)**
`fibcall`	18	16	3
`insertsort`	21	19	3
`jfdctint`	32	30	3
`matmul`	46	44	3
`qurt`	152	170	5

Table 9. Initial complexity of the ILP formulation

	$BP_{@}$	BP_H	$BP_{@\oplus H}$
`fibcall`	1.1	0.8	0.9
`insertsort`	1.4	2.6	1.6
`jfdctint`	1.5	2.3	1.9
`matmul`	1.1	2.7	1.4
`qurt`	2.5	6.5	3.8

Table 11. Aliasing to the BHT (# branches sharing an entry)

	# constraints (C)	# variables (V)	arity (A)
`fibcall`	$\times$ 1.4	$\times$ 1.8	$\times$ 1.9
`insertsort`	$\times$ 3.2	$\times$ 3.6	$\times$ 3.4
`jfdctint`	$\times$ 2.7	$\times$ 3.0	$\times$ 3.3
`matmul`	$\times$ 3.2	$\times$ 4.0	$\times$ 4.4
`qurt`	$\times$ 2.5	$\times$ 3.1	$\times$ 6.7

Table 12. Impact of the history length on the complexity (4-bit vs 2-bit)

	# constraints (C)			# variables (V)			arity (A)		
	$BP_{@}$	$BP_{@\oplus H}/BP_{@}$	$BP_{H}/BP_{@}$	$BP_{@}$	$BP_{@\oplus H}/BP_{@}$	$BP_{H}/BP_{@}$	$BP_{@}$	$BP_{@\oplus H}/BP_{@}$	$BP_{H}/BP_{@}$
`fibcall`	× 2.3	× 6.4	× 4.4	× 2.6	× 8.6	× 6.5	× 3.0	× 5.4	× 5.4
`insertsort`	× 4.0	× 17.2	× 51.2	× 4.8	× 27.1	× 90.2	× 3.0	× 13.5	× 46.8
`jfdctint`	× 3.1	× 10.8	× 28.5	× 3.3	× 14.5	× 42.6	× 3.0	× 10.2	× 40.2
`matmul`	× 3.4	× 7.8	× 24.5	× 4.0	× 9.6	× 40.8	× 3.0	× 6.9	× 56.1
`qurt`	× 4.4	× 10.3	× 28.1	× 5.2	× 13.0	× 45.8	× 3.4	× 10.5	× 81.6
average	× 3.4	× 10.5	× 27.3	× 19.9	× 14.6	× 45.1	× 3.1	× 9.3	× 46.0

Table 10. Modeling complexity of 2-bit dynamic branch predictors

diction, at least when the computational complexity of WCET analysis is an issue.

6. Conclusion

In hard real-time systems, the execution of programs must meet the deadlines. An upper bound on the worst-case execution time is computed to check whether timing constraints can be guaranteed. Static WCET analysis requires a detailed modeling of the execution of basic blocks in the processor, which has long been restricted to very simple cores. However, performance requirements now lead to use more complex processors that implement sophisticated schemes, such as dynamic branch predictors which we focus on in this paper.

Developing a model for a dynamic branch predictor is a huge work. Several solutions have reported in the literature as mentioned in Section 1 but each of them is applicable to a given scheme (i.e. with 1-bit counters and local history registers) and cannot easily be extended to other ones. In this paper, we introduced a framework that accounts for the most common functionalities of dynamic branch predictors and makes it possible to generate a model of a new predictor in an automatic way. We described each step of the framework in sufficient details so that the reader can easily derive its own models.

In the evaluation section, we use our framework to generate models for three kinds of dynamic branch predictors based on 2-bit counters (by far the most frequently used) and, for some of them, on a global history register, which exacerbates the interactions between branches. We show that modeling the branch predictor instead of considering each branch as mispredicted leads to tighter WCET estimates. However, our results also show that the complexity of the models is high, especially for the most sophisticated predictors. It generally overcomes the complexity of the integer linear program used to compute the WCET. Since taking branch prediction into account when determining WCETs may not be affordable, our recommendation is to take this study into consideration when selecting a processor to run time-critical software.

As future work, we plan to extend this framework to complementary solutions that focus on the analysis of the Branch Target Buffer, such as [5, 6].

References

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In S. Min, R. Pettit, P. Puschner, and T. Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin / Heidelberg, 2011.

[2] I. Bate and R. Reutemann. Efficient integration of bimodal prediction and pipeline analysis. In *11^{th} international conference on embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 39–44, august 2005.

[3] C. Burguière and C. Rochange. A Case for Static Branch Prediction Modeling in Real-Time Systems. In *Proceedings of the 11^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA'05)*, pages 33–38, august 2005.

[4] C. Burguière and C. Rochange. On the Complexity of Modeling Dynamic Branch Predictors when Computing Worst-Case Execution Time. In *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*, august 2007.

[5] A. Colin and I. Puaut. Worst-case execution time analysis for a processors with branch prediction. *Journal on Real-Time Systems*, 18(2-3):249–274, may 2000.

[6] D. Grund, J. Reineke, and G. Gebhard. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture*, 2010.

[7] X. Li, T. Mitra, and A. Roychoudhury. Modeling Control Speculation for Timing Analysis. *Journal on Real-Time Systems*, 29(1):27–58, january 2005.

[8] Y.-T. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM/SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'95)*, volume 30, pages 88–98, 1995.

[9] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, december 1999.

[10] J. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8^{th} ACM/IEEE International Symposium on Computer Architecture (ISCA 1981)*, pages 135–148, 1981.

[11] T.-Y. Yeh and Y. N. Patt. Alternative Implementation of the Two-Level Adaptative Branch Prediction. In *Proceedings of the 19^{th} ACM/IEEE International Symposium on Computer Architecture (ISCA 1992)*, pages 124–134, 1992.