A Time-predictable Branch Predictor

Schoeberl, Martin; Rouxel, Benjamin; Puaut, Isabelle

Published in:
Proceedings of the ACM Symposium on Applied Computing

Link to article, DOI:
10.1145/3297280.3297337

Publication date:
2019

Document Version
Peer reviewed version

A Time-predictable Branch Predictor

Martin Schoeberl
Technical University of Denmark
Lyngby, Denmark
masca@dtu.dk

Benjamin Rouxel
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
benjamin.rouxel@irisa.fr

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
isabelle.puaut@irisa.fr

ABSTRACT

Long pipelines need good branch predictors to keep the pipeline running. Current branch predictors are optimized for the average case, which might not be a good fit for real-time systems and worst-case execution time analysis.

This paper presents a time-predictable branch predictor co-designed with the associated worst-case execution time analysis. The branch predictor uses a fully-associative cache to track branch outcomes and destination addresses. The fully-associative cache avoids any false sharing of entries between branches. Therefore, we can analyze program scopes that contain a number of branches lower than or equal to the number of branches in the prediction table. Experimental results show that the worst-case execution time bounds of programs using the proposed predictor are lower than using static branch predictors at a moderate hardware cost.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; **Embedded systems**;

KEYWORDS

real-time systems, worst-case execution time

ACM Reference Format:
Martin Schoeberl, Benjamin Rouxel, and Isabelle Puaut. 2019. A Time-predictable Branch Predictor. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3297280.3297337

1 INTRODUCTION

Many features of state-of-the-art microprocessors improve the average case performance, but are hard to model for worst-case execution time (WCET) analysis [23]. Some features may even result in higher WCET bounds than when not using them. Branch prediction is one of those features. It is important in longer processor pipelines to speculate over branches, in order to keep the functional units busy. The challenge for general purpose processors is to keep the misprediction rate low in the average case. A misprediction is then considered just a performance reduction. However, this is not appropriate for real-time systems for which we need to model processor features for their worst-case behavior.

In a branch predictor that is optimized for the average case, two branches might map to the same entry in the branch prediction table and conflict with each other at run-time. In the average case this is just annoying and a possible reduction of performance. For real-time systems, conflicts are a considerable issue as the aim is to have safe and tight WCET bounds.

This paper presents a branch predictor especially designed to enable tight WCET analysis. The predictor consists of a fully-associative prediction table that includes the full address of the branch as tag memory. Therefore, no two branches can share one entry. With that fully-associative table we can use a simple scope-based WCET analysis to determine if all branches at some program scope fit into the branch prediction table. When all branches fit into the branch prediction table, we can model each loop exit branch individually for this scope. Basically, after the first iteration of a loop, the loop exit branch will be predicted correctly for the remaining iterations of the loop (except the loop exit at the last iteration).

We have implemented the branch predictor in hardware in an FPGA to explore the feasibility of using a fully-associative table. Besides storing the branch address as tag memory, the table also contains the branch target address. Therefore, a correctly predicted branch is executed in a single cycle independent of being taken or not. We have integrated the model of this branch predictor and the scope-based analysis in the WCET analysis tool Heptane [10]. We consider this as one example of co-design of time-predictable hardware with WCET analysis.

The hardware implementation shows that a fully-associative table is feasible for up to 64 entries. In that case, the hardware consumption is about 20 % of a Patmos RISC style processor. When connecting the branch predictor to the Patmos processor it is, not even with 128 entries, dominating the (hardware) critical path.

Results from this WCET analysis on Mälardalen and Polybench benchmarks show that the proposed predictor never results in worse WCET bounds than the initial Patmos static branch predictor. Results also show that depending on the loop shapes generated by the compiler, the gap between the two predictors differ a lot (much higher for *do-while* loops than for *for* loops, for which the gain is negligible). Results also show that software-emulated floating point operations introduce a lot of spurious branches that call for a larger branch prediction table than hardware-supported integer operations.

A number of WCET-oriented analyses of branch predictors have been designed in the past [2, 4, 7, 9, 14, 16, 18]. These works aim at analyzing existing predictors, designed for the average-case. In contrast to these works, the proposed predictor is designed with predictability in mind, and is co-designed with the associated analysis.

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297337

The main contributions of this paper are:

- the design of a time-predictable branch predictor,
- the co-designed static scope-based analysis, by cooperation between different research domains,
- experimental evaluation of the hardware cost of the proposed predictor,
- experimental evaluation of the reduction of WCET bounds using the proposed predictor compared to basic static branch prediction, and analysis of the impact of loop shapes on the reduction.

This paper is organized as follows. Section 2 surveys related work. Section 3 describes the proposed time-predictable branch predictor and its different variations. Section 4 presents the associated scope-based static WCET analysis technique. Section 5 analyzes the worst-case behavior of the proposed branch predictor. Finally, Section 6 concludes and gives directions for future work.

2 BACKGROUND AND RELATED WORK

Branch prediction techniques help performing control flow speculation by predicting the outcome of branch instructions. If the prediction is correct, then execution proceeds without interruption, whereas for incorrect predictions, the speculatively executed instructions have to be cancelled, incurring a time penalty. If branch prediction is not modeled, WCET estimation tools must conservatively assume that all branches are mispredicted, resulting in overestimations.

Branch prediction can be either *static* or *dynamic*. Static scheme associate a fixed prediction for every branch (e.g. backward edges always assumed taken, or compiler-guided). Dynamic predictors predict the branch outcome dynamically based on the past execution history. The first dynamic technique proposed is called *local* branch prediction. This scheme uses a branch prediction table indexed by the lower bits of the program counter. Each entry contains a k-bit saturated counter representing the last predictions. *Global* schemes exploit correlation between branches. Global schemes use a single shift register, called *branch history register* to record the outcomes of n most recent branches. As in local schemes, there is a global branch prediction table in which the predictions are stored. The various global schemes differ from each other in the way the prediction table is looked up when a branch is encountered.

Static branch predictors are easy to analyze. However, for compiler-directed static prediction, the compiler has to statically determine the direction of the branch to be predicted to minimize WCET estimates (see [3, 5]). In addition, the performance of static predictors like *backward taken, forward not taken* are highly influenced by the compiler code generation, raising the issue of code generation, in particular basic block re-ordering, for obtaining as low as possible WCET estimates [17].

Dynamic branch predictors are less predictable and harder to analyze than static branch predictors, as observed in [8] and [6]. For dynamic cases, the analysis has to determine both if there is a hit in the prediction table and the actual contents of the prediction table. The first work was on local dynamic branch predictors [7]. Static analysis of dynamic branch predictors were proposed in [2, 16], with some restrictions to keep the analysis tractable: absence of conflicts in the prediction table for [2], consideration of conflicts in [16], but only one bit counters to keep the analysis complexity low. Compared to the above-mentioned studies, the main contribution of the work presented in [4] is a precise modeling of misprediction penalties, that depend on the direction of the branches. The work presented in [14] releases these restrictions through the provision of a framework for the analysis of a large set of global branch predictors used in embedded processors. Experimental results using this framework however show that the complexity of models can be significant, in particular when increasing the number of bits in the prediction table or the branch history length. A framework for the static analyses of branch target buffers was presented in [9]. Their analyses are based on abstract interpretation, and are shown to be able to successfully analyze a panel of non trivial branch target buffers. More recently, a technique to support longer branch history length was proposed in [18] at the price of reasonable loss of precision.

In this study, in contrast to existing works whose objective is to analyze increasingly complex dynamic branch predictors, originally designed for the average-case, our focus is on the design of a simple and easy to analyze branch predictor, optimized for the worst-case.

3 BRANCH PREDICTION DESIGN

3.1 Hardware Assumptions

For the rest of the paper we assume a classic RISC style processor with 5 pipeline stages: fetch, decode, execute, memory, and write-back. Figure 1 shows such a pipeline organization, already with a branch prediction unit attached. In that pipeline each instruction has a total latency of 5 clock cycles, but due to pipelining (if there are no stalls) each instruction takes effectively one clock cycle. The added branch predictor is shown as BP1, BP2, and BP3, as it spans the first three pipeline stages.

We consider conditional branches, simply called branches for the rest of the paper, where the condition is a comparison between two register values. It is also common to encode branch targets as offsets from the current address, which is held in the program counter (PC).

A branch instruction may be split into three different steps: (1) evaluate the condition, (2) compute the branch target, and (3) change the program flow to the branch target if the branch is taken. A branch instruction is fetched in the first stage. In the decode stage, besides decoding the branch instruction, the two register values used to evaluate the branch condition are loaded from the register file and the branch target address is computed. In the execution stage, the branch condition is computed. If evaluated to *true*, the target address is loaded into the PC. If the condition evaluates to *false*, the pipeline continues to execute in program order.

While the branch instruction moves through the pipeline, the following sequential instructions are fetched and decoded. If the branch is taken, the following instructions are replaced by a bubble (nop instructions). This behavior results in different execution times for a branch if taken or not. If not taken, it is only one clock cycle; if taken, it is three clock cycles in our example pipeline. This behavior is also called *statically predict not-taken*.

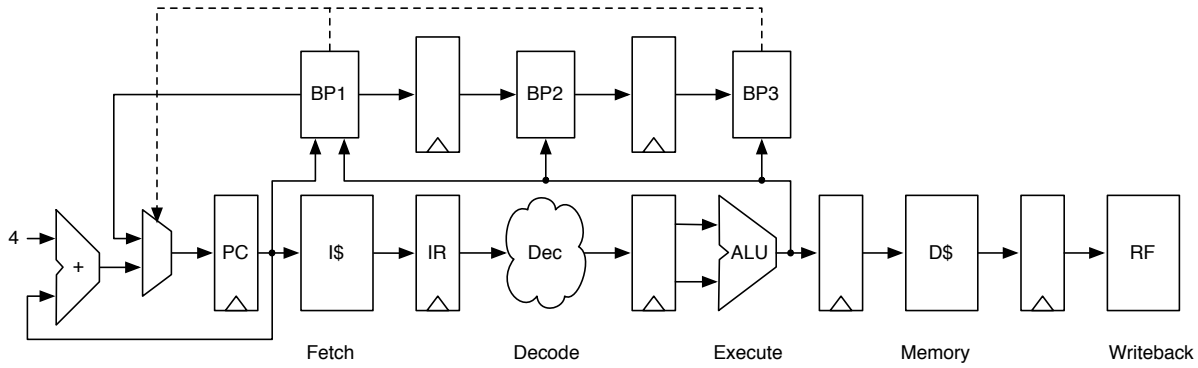


Figure 1: Branch prediction unit for a RISC pipeline.

3.2 Classic (Local) Branch Prediction

A standard local branch predictor uses a table recording estimations of if a branch is taken or not taken. This table is also called *branch history table* as it records the history of branch outcomes. In the simplest case this table contains a single bit per entry, and the table is indexed with the lower bits of the PC. The table contains no tags, as in caches, as reading a wrong entry may "just" result in a performance penalty, but the program executes correctly. When the branch instruction reaches the execution stage, the branch outcome is compared with the prediction. When it is different from the prediction, branch predictor flushes the two instructions in the pipeline, sets the PC to the correct target, and updates the branch history table. Using more than one bit for prediction can add some hysteresis to the predictor to avoid changing the prediction back and forth when executing an inner loop several times.

Besides estimating if a branch is taken or not, the branch target address needs to be computed as well. In a standard 5-stage RISC pipeline this can be performed in the decode or execution stage. Therefore, even on a correctly predicted branch a taken branch takes two clock cycles. Therefore, a branch history table can be extended to also include the *target address* of a taken branch.[1] Again, if two different branches map to the same entry in the table, a branch might branch to a wrong destination. This can be detected in the execution stage and corrected. This results also in "just" a performance penalty.

3.3 A Time-predictable Branch Predictor

In contrast to the classic predictor presented above, the base idea here is to build a branch predictor that is an easy target for WCET analysis. Existing predictors use a prediction table without tags, thus different branches can alias to the same entry. In case of aliasing, the predicted outcome of the branch is wrong, but this only hurts performance. For WCET analysis, aliasing is a source of pessimism when estimating the worst-case outcome of branches (if two branches map to the same entry, the analysis has to consider the worst-case situation). Moreover, detecting the presence of aliasing is a complex and time consuming task. Therefore, our predictor will tag each entry in the prediction table. Furthermore, to avoid conflict misses we aim at a high associativity (for the scope of this paper the prediction table will be fully-associative).

In its default configuration, the proposed branch predictor will use a branch target buffer (BTB) that is fully-associative, and stores in each entry:

- the address of the branch instruction in the tag memory, to avoid aliasing;
- the branch target address; and
- the branch outcome estimation (as one bit or as a two-bit saturating counter).

The BTB stores all conditional branches, and the replacement policy for the BTB is FIFO[2] (First-In First-Out). As the BTB contains a tag entry for the branch, aliasing of two different branches is completely avoided. Unconditional branches are not stored in the BTB. In the following, the term *branch* refers to conditional branches only.

Branch prediction is performed at the fetch stage, by looking for the current PC in the BTB. In case of a *miss* in the BTB (meaning the instruction is not a branch or the branch is not yet in the BTB), the pipeline proceeds normally (i.e. the pipeline proceeds with the next instructions in sequence). In case of a *hit*, the branch history bits and the branch target address are used to determine *if* and *where* to branch. In case the prediction bits outcome is *taken*, then the pipeline is filled at the fetch stage with the branch target address, else the pipeline proceeds normally.

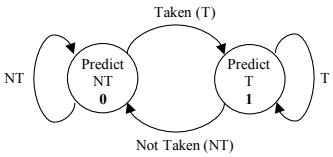
On a miss in the BTB, the BTB is updated for a branch instruction in the execution stage. The execution stage also checks if the prediction was correct, updates the BTB accordingly, and triggers a fetch from the correct address if the prediction was wrong.

3.4 Variations

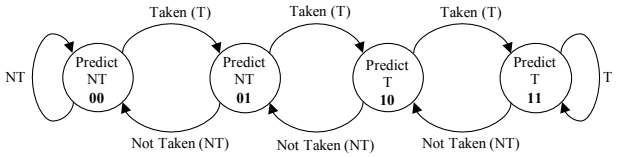
3.4.1 1-bit history. In this variation, the BTB stores a history of one bit used to predict the direction of a branch when found in the BTB (predict *not taken* (NT) if zero, predict *taken* (T) if one). The prediction bit is flipped in case of wrong prediction, as depicted below.

[1] All target addresses of every branch are known at compile time

[2] The analysis presented in this paper is not limited to a BTB using FIFO replacement. It applies to any replacement policy for which the following property holds: if the BTB contains $X \leq N$ values, with N the BTB size, no access to one of the X values will evict any of them (all values will stay in the BTB).



3.4.2 2-bits history. This second variation uses a 2-bits history to predict the outcome of every branch, as depicted below. When the history bits are 00 or 01 (respectively 10 and 11) the branch is predicted *not taken* (respectively *taken*).



4 TIMING ANALYSIS

The proposed branch predictor allows a scope-based WCET analysis. Assuming properly nested loops, the analysis for each loop determines if all branches belonging to the loop scope are guaranteed to stay in the BTB, in order to determine the number of correctly predicted loop exit instructions. The analysis is aware of loop nesting to provide tight estimates, and supports elaborate models of misprediction penalties.

4.1 Assumptions and Notations

Our analysis applies to programs amenable to static WCET analysis (see Section 5.1 for the exact restrictions imposed by the static WCET estimation tool used in the experimental evaluation). The branch prediction analysis needs properly nested loops. Different shapes of loops are supported, according to the following three criteria on the conditional test to exit the loop:

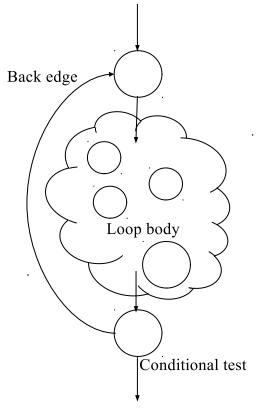
- test executed *before* or *after* the loop body;
- test branches *outside* or *inside* the loop;
- test located at an address *immediately before* or *not immediately before* the start of the loop body.

The analysis is detailed for the following two loop shapes, described below and depicted in Figure 2. These two loop shapes are representative of the most common cases found in compiled code. The analysis implementation supports all loop shapes corresponding to any combination of loop shape criteria.

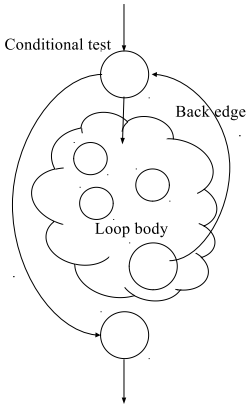
- Figure 2a: conditional test *executed after* the loop body, *branching inside* the loop, conditional test *not immediately before* the loop body;
- Figure 2b: conditional test *executed before* the loop body, *branching outside* the loop, conditional test *immediately before* the loop body.

The following notations are used in the analysis description:

- $freq\_exec_l$ is the loop bound of loop l (maximum number of times the body is executed). The conditional branch is thus executed $freq\_exec_l$ times for loop shape 1 (Figure 2a), and $freq\_exec_l + 1$ for loop shape 2 (Figure 2b). The bound is *local* (it counts the maximum number of iterations for *each* entry in the loop, not the cumulated value for all entries).



(a) Loop shape 1: test after body, branch inside, test not immediately before body



(b) Loop shape 2: test before body, branch outside, test not immediately before body

Figure 2: Considered loop shapes

Table 1: Notations for prediction outcome categories

	Branch taken	Branch not taken
BTB miss	n^T_{miss}	n^{NT}_{miss}
BTB hit, predict taken	n^T_{good}	n^{NT}_{bad}
BTB hit, predict not taken	n^T_{bad}	n^{NT}_{good}

- The scope of a loop, $scope(l)$, is defined as the number of distinct branches *inside* the loop, plus the loop conditional branch itself. By definition it includes all branches contained in functions called (directly or indirectly) from the loop body.
- $antecedents(l)$ refers to all loops including l until the outermost one (all direct and indirect parents, considering function calls).
- Maximum total number of times loop l is entered/exited:

$$\eta_l = \Pi_{p \in antecedents(l)} freq\_exec_p$$

- Maximum total number of entries into the loop l guaranteed to fit in the BTB:

$$\delta_l = (\Pi_{p \in antecedents(l) | scope(p) \leq BTB} freq\_exec_p)$$

These definitions hold for all considered loop shapes.

The analysis provides for the conditional branch of every loop l the total number of times the branch falls into the categories listed in Table 1. In the text of the analysis, the categorization of a branch is a pair (X,Y), with X the prediction (*miss, good, bad*) and Y the actual branch direction (*T,NT*). The corresponding number of occurrences in each category is denoted n^Y_X. For example, n^T_{good} denotes the number of times a branch is predicted taken and is actually taken at runtime. This categorization of branches is fine grain, to allow precise modeling of misprediction penalties such as in [4]. Note that our analysis currently focuses on loop exit

branches, leaving the analysis of the other conditional branches for future work.

We hereafter detail the analysis for loop shape 1 and 2. For space considerations, we only give detailed description of the analysis for loop shape 1 (Figure 2a) and do not describe all the details for the other loop shape.

4.2 Analysis for Loop Shape from Figure 2a

For this loop shape, the branching behavior for each execution of loop l is T ... T (repeated $freq\_exec_l - 1$ times) NT. This pattern T ... T NT is repeated several times when l is included in other loops.

For a given loop l, the number of times the branch is correctly predicted or mispredicted depends on how many branches in the loop scope are guaranteed to fit in the BTB, as defined by $scope(l)$.

4.2.1 Case 1: branches are not guaranteed to fit inside the BTB. If $scope(l) > BTB$, branches executed in the scope of loop l are not guaranteed to stay in the BTB once loaded. However, we can guarantee a BTB miss only for loop l once. This is because there may exist conditional branches that are not loop branches (i.e. conditional constructs) whose execution depends on input data, making it hard to ensure that a loop branch is always evicted from the BTB.

With the loop shape we are focusing on (see Figure 2a), the very first entry of the loop conditional branch is *taken*, and results in a BTB miss. It is thus categorized as *(miss,T)*. The following iterations, last iterations excluded, are also *taken*. Since it cannot be guaranteed that the loop branch for these iterations is evicted from the BTB, the most pessimistic situation has to be considered. Assuming that a misprediction is the worst-case, the following iterations are then conservatively considered mispredicted (category *(bad,T)*). Then each last iteration of the loop are *not-taken* and thus predicted correctly (category *(good,NT)*).

The number of occurrences in each category is then as follows (only non zero counts are listed from now on):

- $n^T_{miss} = 1$
- $n^{NT}_{good} = \eta_l$
- $n^T_{bad} = (freq\_exec_l - 1) * \eta_l - 1$

4.2.2 Case 2: branches are guaranteed to fit in the BTB. When $scope(l) \leq BTB$, it means that all branches in the scope of the loop stay in the BTB after first loaded. In the following, we give for the two branch predictor configurations:

- The categorization for all first executions of the loop (the first iteration of a loop might be executed several times in the presence of loop nesting),
- The categorization of the next iterations of the loop (last iteration excluded),
- The categorization of the last iterations of the loop.

The analysis is detailed for the two variations of the predictor (1-bit history and 2-bit history).

Configuration with 1-bit history.

- First iterations. Apart from the very first entry in the loop, which is a BTB miss, all entries are mispredictions (category *(bad,T)*). For the δ_l entries that, by definition of δ_l, stay in the BTB once loaded, the prediction bit is 0, because the last prediction for the branch (last iteration) was T and the actual outcome was NT. Thus the prediction bit was flipped to 0 and the entry is mispredicted (category *(bad,T)*). For the $\eta_l - \delta_l$ that are not guaranteed to stay in the BTB, the branches are conservatively classified as mispredicted, resulting in a the category *(bad,T)* (it cannot be guaranteed they stay in the BTB, and cannot be guaranteed to be evicted from the BTB).
- Next $(freq\_exec_l - 2) * \eta_l$ iterations. For these branches, the prediction bit is 1 (predicted T) and the branch is T. Thus they are predicted correctly (category *(good,T)*).
- Last iterations (executed η_l times). The prediction bit is 1 (T) and the branch is not taken, thus a misprediction and a change of the prediction bit to 0 (category *(bad,NT)*).
- Total:
  - $n^T_{miss} = 1$
  - $n^T_{bad} = \eta_l - 1$
  - $n^{NT}_{bad} = \eta_l$
  - $n^T_{good} = (freq\_exec_l - 2) * \eta_l$

Configuration with 2-bit history.

- First iterations. Similarly to the 1-bit history predictor, the $\eta_l - \delta_l$ that are not guaranteed to stay in the BTB are conservatively classified as mispredicted (category *(bad,T)*). Regarding the δ_l executions that stay in the BTB across executions of the loop, the 2-bit history now allows all of them to be correctly predicted (category *(good,T)*), because the previous branch to exit the loop changed the prediction bits from 11 to 10. When re-entering the loop again, the prediction is T and the branch is T.
- Next $(freq\_exec_l - 2) * \eta_l$ iterations. The prediction bits are 11 (predict T) and the branch is T, the prediction is therefore correct (category *(good,T)*).
- Last iterations (executed η_l times). The prediction bits are 11 (predict T) and the branch is NT, thus the branches are mispredicted (category *(bad,NT)*).
- Total:
  - $n^T_{miss} = 1$
  - $n^{NT}_{bad} = \eta_l$
  - $n^T_{bad} = \eta_l - \delta_l - 1$
  - $n^T_{good} = \delta_l + (freq\_exec_l - 2) * \eta_l$

4.3 Analysis for Loop Shape from Figure 2b

For this loop shape, the branching behavior for each entry of loop l is NT ... NT (repeated $freq\_exec_l$ times) T. This pattern NT ... NT T is repeated when l is included in outer loops. Similarly to the other loop shape, the result of the analysis depends on whether or not the BTB is big enough to store the loops branches.

4.3.1 Case 1: branches are not guaranteed to fit inside the BTB. If the scope is larger that the BTB, for all the two variations (1-bit history and 2-bit history) we have:

- $n^{NT}_{miss} = 1$
- $n^T_{bad} = \eta_l$
- $n^{NT}_{bad} = (freq\_exec_l) * \eta_l - 1$

4.3.2 Case 2: branches are guaranteed to fit in BTB. In case the scope fits in the BTB, we have, for 1-bit history (details omitted for space consideration):

- $n^{NT}_{miss} = 1$
- $n^{T}_{bad} = 2 * \eta_l - 1$
- $n^{NT}_{good} = (freq\_exec_l - 1) * \eta_l$

Then, for a 2-bits history we have:

- $n^{NT}_{miss} = 1$
- $n^{T}_{bad} = \eta_l$
- $n^{NT}_{bad} = \eta_l - \delta_l - 1$
- $n^{NT}_{good} = \delta_l + (freq\_exec_l - 1) * \eta_l$

4.4 WCET Calculation with IPET

The classical IPET (Implicit Path Enumeration Technique [13]) formulation computes WCET estimates using an Integer Linear Programming (ILP). IPET reflects the program structure and the possible execution flows using a set of linear constraints. An upper bound of the program's WCET is obtained by maximizing the objective function $\sum_{i \in BasicBlocks} T_i * f_i$. T_i is the timing information of basic block i, that integrates the effects of micro-architecture. Frequency f_i, a variable in the ILP system, to be instantiated by the ILP solver, corresponds to the number of times basic block i is executed along (one of) the longest path(s). A value of 0 means that the block is not on the critical path.

We have extended IPET to include the timing penalties of branch prediction (gains if the penalty is negative) by adding a binary variable $isex_l$ to the basic block l containing the loop branch. $isex_l$ is set to 0 if the node is not on the critical path (if $f_l = 0$), and 1 if it is (if $f_l \neq 0$).

To model this *if-then-else* scenario in ILP ($isex_l = 0$ if $f_l = 0$), there exist several ways depending on the used solver. For example, with IBM CPLEX we can use *indicator constraints* (see equation (1)) whereas in the open-source LPSolve solver we can use a Big-M notation (see equation (2)). The choice of a BigM value might be difficult as this value must be bigger than any other integer in the system to solve. One could choose the maximum integer the solving computer can deal with. In our experiments (see details in Section 5.1), we use CPLEX with the indicator constraints.

$$\forall l \in loopbranches; \qquad isex_l = 0 \rightarrow f_l = 0 \tag{1}$$

$$\forall l \in loopbranches; \\ 0 \times isnotex_l + 1 \times isex_l \leq f_l \leq 1 \times isnotex_l + BigM \times isex_l \tag{2}$$

A new term is then added to the objective function, representing the penalty for each loop branch l, according to the branch classification from Table 1. Let us denote by t^Y_X the penalty corresponding to the branch category (X,Y), with X the prediction (*miss, good, bad*) and the actual branch direction (*T, NT*). The new term added is:

$$\sum n^Y_X \times t^Y_X \times isex_l$$

5 EXPERIMENTAL EVALUATION

This paper has presented a co-designed solution for time-predictable branch prediction with (a) a hardware design optimized for WCET analysis and (b) a WCET analysis specific for this hardware design. This section includes an evaluation of our design through a comparison of WCETs for the proposed branch predictors with other hardware designs (a static *backward taken forward not taken* predictor and the static predictor of the Patmos processor [20, 22]). We voluntarily do not include any more complex branch predictor, because state-of-the-art branch predictors are not designed for predictability and therefore are hard to analyze statically. Experiments were performed on standard WCET benchmarks from the Mälardalen benchmark suite [15] and Polybench benchmark suite.

	Logic cells	Registers	Fmax
4 entries	146	124	210 MHz
8 entries	288	245	184 MHz
16 entries	573	486	171 MHz
32 entries	1138	967	151 MHz
64 entries	2275	1928	133 MHz
128 entries	4524	3849	117 MHz
Patmos core	9437	4384	80 MHz

Table 2: Resource consumption and maximum frequency for different branch predictor configurations.

5.1 Experimental Setup

Hardware Implementation. We have implemented the proposed branch predictor targeting the Patmos processor [20, 21]. For the implementation, we use the hardware construction language Chisel [1]. Chisel allows to generate a C++ based tester and Verilog code for synthesis. Besides the branch predictor we implemented a simulation of the first three stages of RISC style processor pipeline (fetch, decode, and execute) to drive the branch predictor. The simplified processor simulation serves as a testbed for functional verification of the branch predictor. The simplified pipeline is also synthesizable and we report on resource consumption and maximum clock frequency when targeting an FPGA.

We are interested in how expensive the fully-associative, single cycle lookup is. When an entry matching the current PC is found, the result may update the PC, depending on the branch prediction. Therefore, the critical path starts from the PC register output, is fanned out to n comperators for an associativity of n, the comperator outputs are then encoded to find one possible match, and an n port multiplexer gives the result, which may update the PC register. As we aim for a single cycle branch predictor, this operation cannot be pipelined. Furthermore, as we aim for a fully-associative table, the table needs to be implemented in dedicated registers instead of an on-chip memory.

We have synthesized the branch predictor for an Altera/Intel Cyclon IV FPGA with the Quartus Prime 16.1 Light edition. Table 2 shows the result of the resource consumption (logic cells, registers) and the maximum clocking frequency, for different configurations. The last row shows for a comparison the resource consumption and maximum frequency of a Patmos processor. The results show that single cycle lookup BTB with up to 128 entries can be implemented. However the larger the BTB, the higher the branch

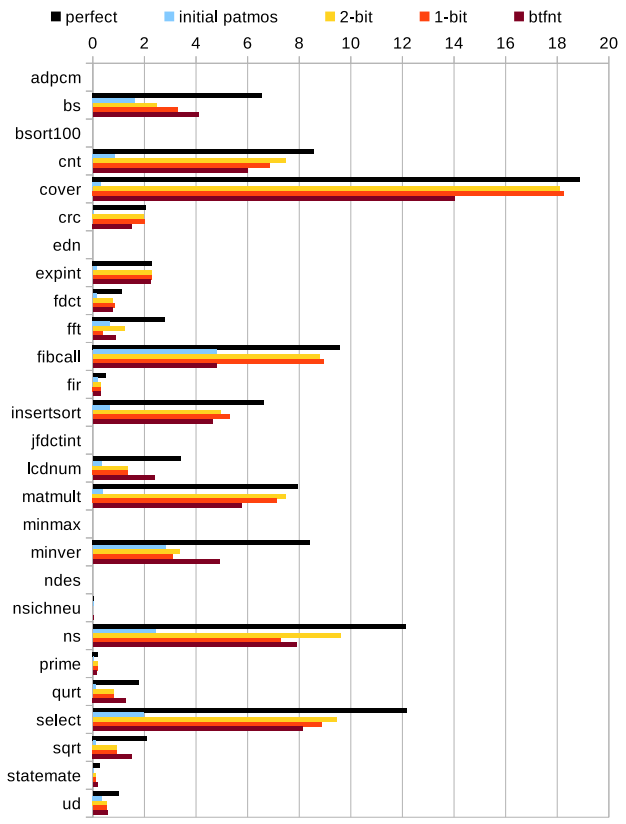


Figure 3: Gain in % over a "pessimistic predictor" with Mälardalen benchmark suite and a 16 entries BTB (loop shape 1)

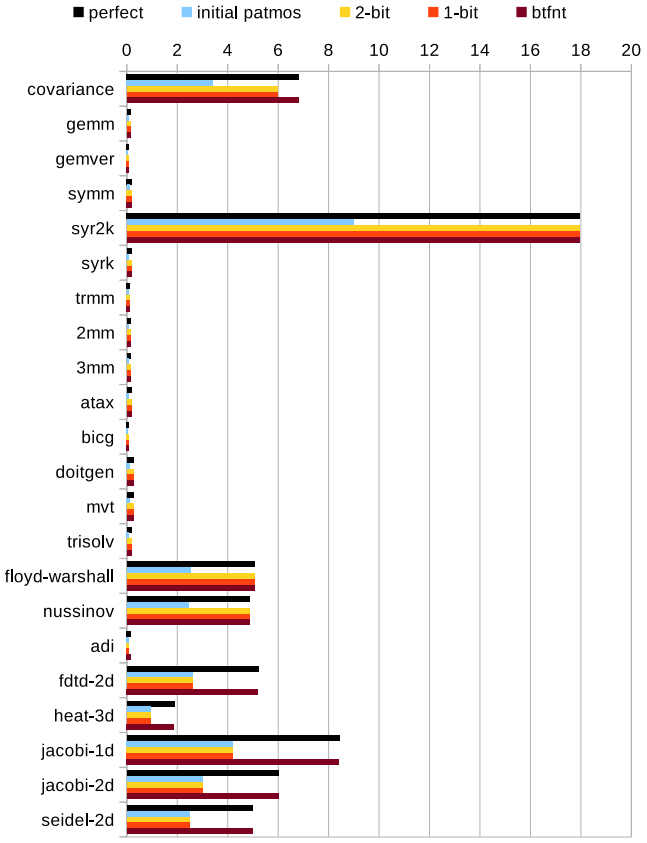


Figure 4: Gain in % over a "pessimistic predictor" with Polybench benchmark suite and a 16 entries BTB (loop shape 1)

Table 3: Modification of branch timing as compared to *predict not taken*

	Branch taken	Branch not taken
BTB miss	2	0
BTB hit, predict taken	-2	2
BTB hit, predict not taken	2	0

prediction hardware cost comparatively to the base Patmos processor. A fully-associative BTB with 16 entries is used by default in the experimental evaluation.

Patmos implements the classic 5-stages pipeline and includes by default the static branch predictor (*statically predict not-taken*) described in Section 3.1. According to the structure of the Patmos pipeline, the timing penalties of branches for the proposed branch predictor as compared to the Patmos static predictor are given in Table 3.

Compilation and WCET Analysis. The analyzed programs were compiled with LLVM for the Patmos processor [19]. the programs were compiled with no optimization (compiler option -O0), except the *-loop-rotate* flag for half of the experiments, that allow to have, when the optimization is applicable, the exit test of loops executed *after* the loop body. A default optimization level of -O0 is used for two main reasons. First, optimizing code is not current practice in safety critical domain, because compilers have to be certified, and compiler optimizations make certification complex if the compiler implements aggressive optimization passes [11]. Second, WCET estimation requires loop bounds to be known, and loop bounds are easier to estimate at source code level (either using static analysis or provided through annotations). Many optimizations change loop bounds, requiring traceability of loop bounds within the compiler when bounds are given at the source code level [12]. Optimization *-loop-rotate* has impact on the shape of loops, transforming almost all loops of shape 1 (Figure 2a) to shape 2 (Figure 2b) without changing the loop bound (except the −1 required in a *do-while* construct).

We have implemented the scope-based analysis in the Heptane static WCET estimation tool [10]. Heptane, like most static WCET analysis tools, imposes a set of constraints on code: no function pointers, no indirect jumps through jump tables, a single entry per loop.

Heptane was modified to support the Patmos instruction set. The analysis of the micro-architecture of Patmos in Heptane being

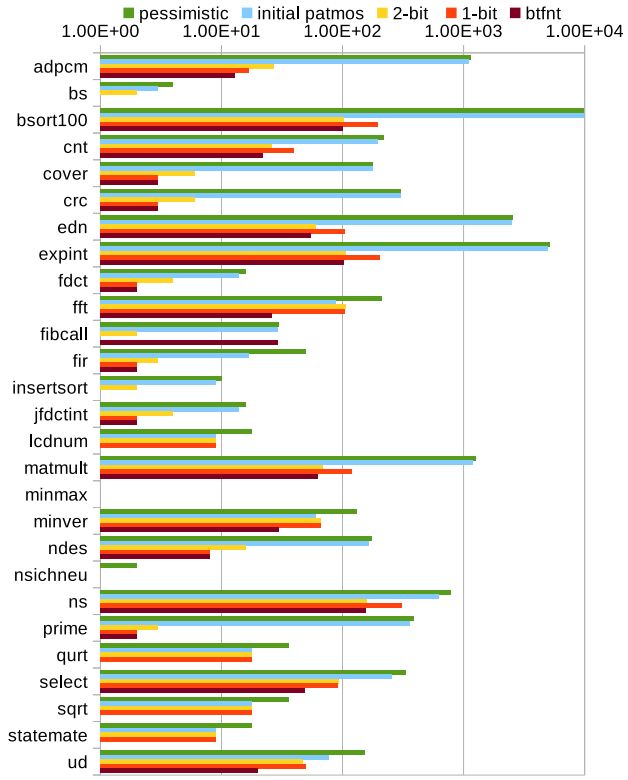


Figure 5: Number of mispredictions with Mälardalen benchmark suite and a 16 entries BTB – logarithmic scale (loop shape 1)

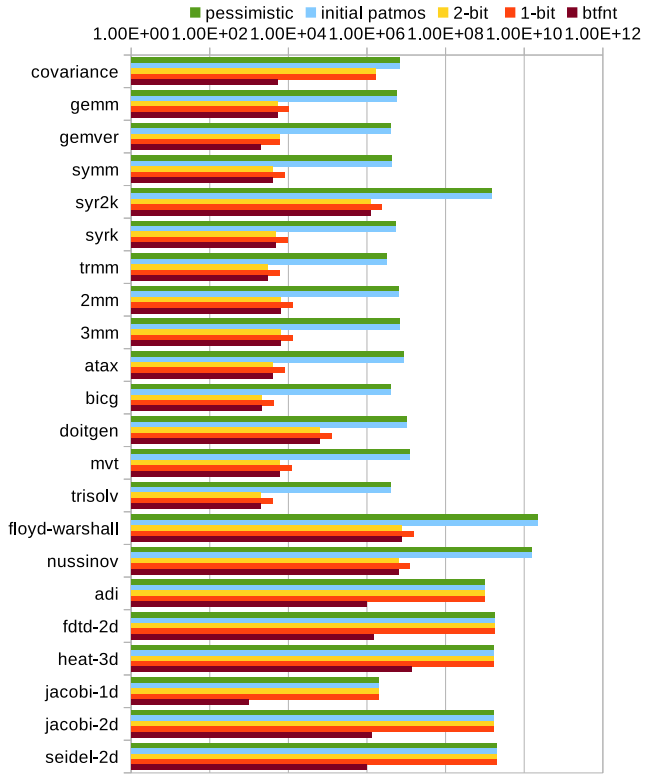


Figure 6: Number of mispredictions Polybench benchmark suite and a 16 entries BTB – logarithmic scale (loop shape 1)

incomplete at the time this paper is written, it is assumed that the Patmos pipeline and memory hierarchy is efficient enough to execute an instruction every clock cycle. Loop bounds are given through annotations.

5.2 Experimental Results

5.2.1 Results for loop shape 1 from Figure 2a. Figures 3 and 4 present the gain in percentage on the WCET when using the two variations of the proposed branch predictor, as compared with a pessimistic predictor for which all branches are systematically assumed mispredicted (penalty of 2 cycles per branch). The higher the gain, the better the predictor.

Three baselines are used to evaluate the quality of our design:

(1) Perfect branch prediction (noted *perfect* in the figure)
(2) The Patmos original static branch predictor described in Section 3.1 (*statically predict not-taken*)
(3) The simple *backward taken, forward not taken* static branch predictor (BTFNT in the figures). For this predictor, we added a delay of 1 clock cycle per conditional branch, required to know if the instruction is a branch, to compute the branch address, and to detect the branch direction. This knowledge is available at the decode stage only.

Results show that in general WCETs estimates when using the proposed branch predictors are smaller than with the default Patmos predictor. Compared to BTFNT, the proposed predictors are sometimes better, sometimes worse; our predictors have a more regular behavior than BTFNT.

One can notice that in some benchmarks from the Mälardalen benchmark suite, branch prediction does result in any improvement. The cause of this behavior is the small number of iterations in many of these benchmarks (for example 3 in *minver*). In this situation, the cost of the initial BTB miss (or misprediction when re-entering the same loop several times) outweights the gains when re-executing the branch again a too small number of times.

The proposed branch predictor performs better on the Polybench benchmarks than on the Mädardalen benchmarks. This comes from the Polybench benchmarks have loops with a much larger number of iterations than in the Mälardalen benchmarks. Having loops with large loop bounds makes the prediction close to perfect branch prediction, since misses and mispredictions are paid only at startup and when entering a loop several times in the application lifetime; all the branches in the large number of iterations are predicted correctly.

The 2-bit history configuration results in almost all situations in smaller WCET estimates than when using 1-bit history. This is because when exiting a loop inside a loop nest, the prediction is

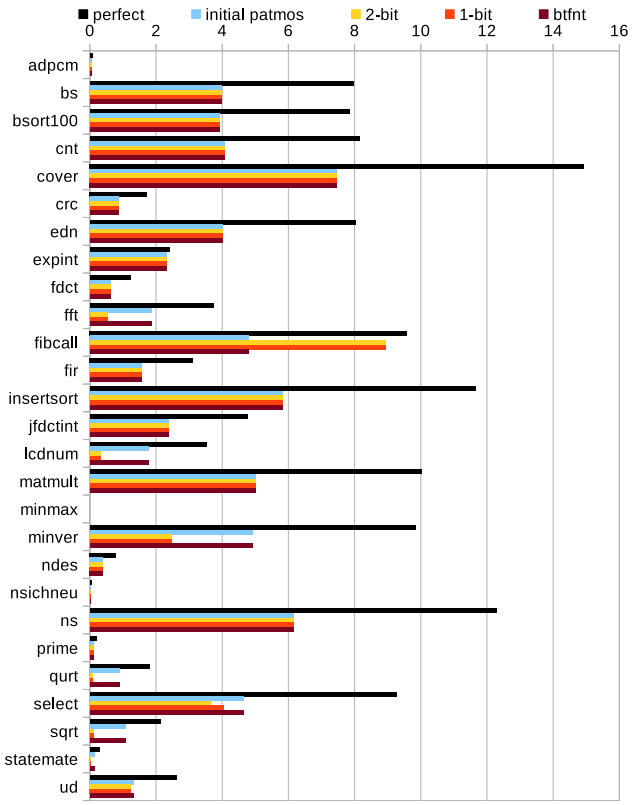


Figure 7: Gain in % over a "pessimistic predictor" with Mälardalen benchmark suite and a 16 entries BTB (loop shape 2)

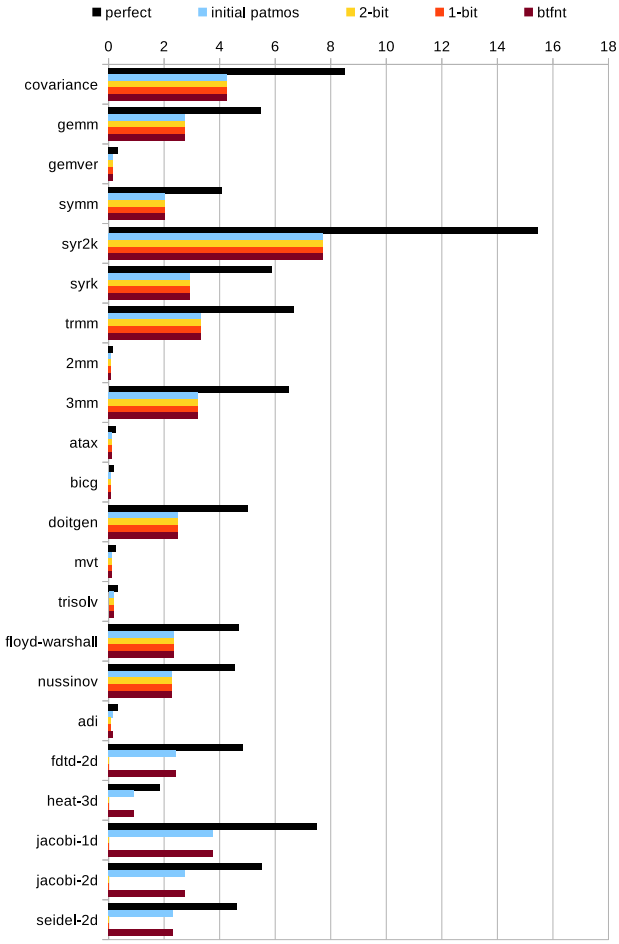


Figure 8: Gain in % over a "pessimistic predictor" with Polybench benchmark suite and a 16 entries BTB (loop shape 2)

correct when re-entering the loop again (provided that all branches fit in the BTB).

There are however some exceptions, when the 1-bit history outperforms the 2-bit history (Mälardalen benchmark *fibcall*). In this benchmark, there is no loop nest (one single loop in the code, iterating 30 times). There is one extra misprediction for the 2-bit counter configuration, because the counter needs two iterations to reach state "predict taken" (one more than the 1-bit history configuration). In other terms, the 2-bit predictor takes longer to warm-up than the 1-bit predictor. The number of loop iterations being moderate, and the cost of a misprediction being larger than the gain for a correct prediction, the impact on the WCET is noticeable.

Figures 5 and 6 give a different view of the same set of experiments, by showing now the absolute number of mispredictions (the lower the better). Since the percentage of mispredicted branches is low as compared to the *pessimistic* baseline, the figures are presented using a logarithmic scale. One can notice that some benchmarks from the Mälardalen benchmark suite have a low estimated number of mispredictions (*minmax, insertsort*). This comes from two factors: (i) these benchmarks do not contain nested loops, thus once the predictor is warmed-up, it will always issue correct predictions; (ii) BTB misses are not depicted in the figure, only branch mispredictions.

The results show that, although the number of mispredictions from Figure 4 looked low, there are actually a non negligible number of mispredictions for the Polybench benchmarks. However, since these benchmarks are compute-intensive (the WCET is dominated by calculations, not by branches) the impact of misprediction on WCETs is limited.

5.2.2 Results for loop shape 2 from Figure 2b. This second set of experiments is again for a BTB with 16 entries, but now operates on loops with mostly the second loop shape. Figures 7 and 8 depict the WCETs for the Mälardalen and Polybench benchmarks respectively.

The results show that for this loop shape the difference of gain of the proposed predictors as compared to the Patmos default prediction exists but is small. This comes from the conjunction of two factors: (i) the shape of loops in the benchmarks (mostly *for loops*, almost no *do while* loop); (ii) the code generated by default by the compiler, that makes Patmos original predictor perform well in the worst-case.

5.2.3 Experiments for floating point values. In the above experiments, the Polybench benchmarks were configured to operate on *integer* values. We performed the same set of experiments when configuring them with *double precision floating point* values. Since in Patmos floating point operations are emulated in software, we observed an increase of the number of branches per loop, caused by the floating point emulation functions. The loop scopes were significantly increased, requiring a larger BTB to benefit from the branch predictor. I.e., 64 entries in the BTB are required to reach similar gains than using integer values.

6 CONCLUSION

This paper has presented a time-predictable branch predictor co-designed with the associated worst-case execution time analysis. The branch predictor uses a fully-associative cache to track conditional branch outcomes and destination addresses. The prediction table entries are tagged with the branch address to avoid false sharing of entries between branches. The design of the branch predictor allows the definition of a simple scope-based static analysis of the branch predictor. Experimental results show that although the design of the predictor and analysis are simple in order to be predictable, the impact of the branch predictor on worst-case execution time bounds is positive, specially for the predictor with a 2-bit history. However, we noticed that the benefit brought by the predictor highly depends on the code generated by the compiler.

As future work, one direction is to make our analysis more precise by identifying the branches that are guaranteed to be evicted from the branch predictor in a loop. Another extension of the work is to study the impact of different structures for the branch predictor table (set-associative instead of fully-associative) on misprediction rates and worst-case execution time bounds.

Acknowledgment

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT[3] contract no. 4184-00127A.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference (DAC 2012)*, Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun (Eds.). ACM, San Francisco, CA, USA, 1216–1225.

[2] Iain Bate and Ralf Dieter Reutemann. 2004. Worst-Case Execution Time Analysis for Dynamic Branch Predictors. In *16th Euromicro Conference on Real-Time Systems (ECRTS 2004), 30 June - 2 July 1004, Catania, Italy, Proceedings*. 215–222.

[3] François Bodin and Isabelle Puaut. 2005. A WCET-Oriented Static Branch Prediction Scheme for Real Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*. IEEE Computer Society, 33–40.

[4] Claire Burguiere and Christine Rochange. 2005. A contribution to branch prediction modeling in WCET analysis. In *Design, Automation and Test in Europe, 2005. Proceedings*. IEEE, 612–617.

[5] Claire Burguiere, Christine Rochange, and Pascal Sainrat. 2005. A Case for Static Branch Prediction in Real-Time Systems. In *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*. IEEE Computer Society, Washington, DC, USA, 33–38. https://doi.org/10.1109/RTCSA.2005.5

[6] Antoine Colin and Stefan M Petters. 2003. Experimental evaluation of code properties for wcet analysis. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. IEEE, 190–199.

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

[8] Jakob Engblom. 2003. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*. IEEE, 152–159.

[9] Daniel Grund, Jan Reineke, and Gernot Gebhard. 2011. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture* 57, 6 (2011), 625–637.

[10] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. 2017. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*. 8:1–8:12.

[11] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

[12] Hanbing Li, Isabelle Puaut, and Erven Rohou. 2014. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems (RTNS '14)*. ACM, New York, NY, USA, Article 97, 10 pages.

[13] Y.-T. S. Li and S. Malik. 1995. Performance analysis of embedded software using implicit path enumeration, In LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems, Richard Gerber and Thomas Marlowe (Eds.). 30, 11, 88–98.

[14] Claire Maiza and Christine Rochange. 2011. A framework for the timing analysis of dynamic branch predictors. In *RTNS*. 65–74.

[15] Mälardalen Real-Time Research Center. accessed 2009. WCET Benchmarks. Available at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[16] Tulika Mitra and Abhik Roychoudhury. 2002. A Framework to Model Branch Prediction for Worst Case Execution Time Analysis. In *2nd International Workshop on Worst-Case Execution Time Analysis*.

[17] Sascha Plazar, Jan Kleinsorge, Peter Marwedel, and Heiko Falk. 2011. WCET-driven branch prediction aware code positioning. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*. IEEE, 165–174.

[18] Wolfgang Puffitsch. 2016. Efficient Worst-Case Execution Time Analysis of Dynamic Branch Prediction. In *ECRTS*. IEEE Computer Society, 152–162.

[19] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. 2012. Compiling for Time Predictability. In *Computer Safety, Reliability, and Security*, Frank Ortmeier and Peter Daniel (Eds.). Lecture Notes in Computer Science, Vol. 7613. Springer Berlin / Heidelberg, 382–391.

[20] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. 2014. *Patmos Reference Handbook*. Technical Report. Technical University of Denmark.

[21] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. 2018. Patmos: A Time-predictable Microprocessor. *Real-Time Systems* 54(2) (Feb 2018), 389–423. https://doi.org/10.1007/s11241-018-9300-4

[22] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. 2011. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*. Grenoble, France, 11–20.

[23] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.* 7, 3 (2008), 1–53. https://doi.org/10.1145/1347375.1347389

[3] http://predict.compute.dtu.dk/