A WCET-Oriented Static Branch Prediction Scheme for Real Time Systems

François BODIN[(1)] - Isabelle PUAUT[(2)]

[(1)] Université de Rennes I/Caps Entreprise - IRISA [(2)] Université de Rennes I - IRISA
Campus Universitaire de Beaulieu, 35042 RENNES Cedex - France
E-mail: {Francois.Bodin|Isabelle.Puaut}@irisa.fr

Abstract

Branch prediction mechanisms are becoming commonplace within current generation processors. Dynamic branch predictors, albeit able to predict branches quite accurately in average, are becoming increasingly complex. Thus, determining their worst-case behavior, which is highly recommended for real-time applications, is getting increasingly difficult and error-prone, and may even be soon impossible for the most complex branch predictors. In contrast, static branch predictors are inherently predictable, *to the detriment of a lower prediction accuracy. In this paper, we propose a WCET-oriented static branch prediction scheme. Unlike related work on compiler-directed static branch prediction, our scheme does not address program average-case performance (i.e. average-case branch misprediction rate) but addresses* worst-case program performance *instead (i.e. branch mispredictions which impact programs WCET estimates). Experimental results on a PowerPC 7451 architecture show that the estimated WCET can be decreased by up to 21% (with an average improvement of 15%) as compared with the method where all branches are conservatively considered mispredicted. Our scheme, although applicable to any processor with support for static branch prediction, is specially suited to processors with complex dynamic predictors, for which safe and tight WCET estimate methods do not exist.*

1 Introduction

Real-time systems differ from general-purpose systems by a stricter criterion of correctness of applications. Actually, the correctness of a real-time application does not only depend on the delivered result but also on the time when it is produced [23]. In *hard real-time* systems, missing a timing constraint (typically a deadline) may be catastrophic. The main problem from a timing perspective is then to verify that a given set of tasks can be assigned to the available system resources such that all tasks meet their timing constraints in all situations, including the worst-case situation. Existing verification techniques require an estimation of the Worst Case Execution Time (WCET) of each task in the system as a prerequisite.

Estimating the WCET of a program is complicated by the fact that many commercially available microprocessors use performance enhancement features which are not designed with real-time systems in mind. The increased performance caused by instruction pipelining, out-of-order execution, data and instruction caching, branch prediction, etc., comes at the cost of a greater variability in execution times and at the cost of a greater complexity of WCET estimation. A number of methods for estimating WCETs for microprocessors using such architectural features have emerged in the recent years, considering: caching [14, 2, 10, 17], pipelining [25], branch prediction [4, 11]. The effect of branch prediction on WCET estimation has received less attention than the effect of the other architectural features (see Section 2 for more details):

- On the one hand, *dynamic* branch predictors are optimized to provide high accuracy. However, they are hard to model and their analysis incur complexity in WCET estimation methods.The current state of the art is that only the most simple dynamic branch predictors are currently modeled and supported by WCET estimation tools. Furthermore, the most complex dynamic branch predictors are not necessarily fully documented due to their inherent complexity and/or due to confidentiality constraints.
- On the other hand, *static* branch predictors are by construction predictable, and thus are easily amenable to WCET analysis. Furthermore, with appropriate instruction-set hooks, the compiler, thanks to its knowledge of code, can improve the predictor performance as compared with the simpler static schemes such as BTFN (Backward Taken, Forward Not taken). Some studies have been undertaken to statically pre-

dict the likelihood that a branch is taken, using either profile data [6] or static program analysis [16]. All of them focus on minimizing the misprediction ratio, which is not the most relevant metric in real-time systems, in which worst-case performance is as important as average-case performance.

We believe that dynamic branch predictors are, or will soon be, too complex to be modeled and accounted for in WCET analysis methods. Instead, for processors with support for compiler-directed branch prediction (e.g. PowerPC, MIPS R4x00), we propose in this paper an algorithm to *statically predict* a subset of the conditional branches of a real-time program (those impacting the program WCET), such that the *WCET estimate is minimized*. The static branch prediction scheme is implemented using an iterative algorithm working on the program control flow graph.

We think that this paper improves the state of the art regarding the use of branch predictors in several respects. First, due to the use of a static branch prediction scheme, the use of branch prediction is inherently *predictable*. Furthermore, there is no need for branch prediction modeling techniques, which simplifies WCET estimation. Finally, the metrics we optimize are relevant in real-time systems, in which worst-case execution time is as important as average-case execution time. Our scheme, although applicable to any processor with support for static branch prediction, is specially suited to processors with complex dynamic branch predictors, which are less and less amenable to WCET estimation. It should be noted that it is not our intent in this paper to argue that high performance processors (such as the PowerPC 7451 used to evaluate our scheme) are predictable enough to be used in hard real-time systems. Instead, as a first step to address the predictability issues of such complex micro-architectures, we concentrate on making their branch prediction mechanism more predictable by using compiler techniques. The other sources of unpredictability of such processors (e.g. the timing anomalies coming from out-of-order execution) are considered outside the scope of this paper.

The rest of this paper is organized as follows. Section 2 first describes the state of the art of branch prediction techniques, their use in real-time systems, as well as the way they can be taken into account when estimating worst-case execution times. Section 3 then describes our proposal for producing static predictions of conditional branches. Section 4 evaluates our scheme on a PowerPC 7451 architecture. Implementation issues are discussed in Section 5. Finally, concluding remarks are given in Section 6.

2 Related work

When designing and verifying real-time systems, it is required or desirable to predict the worst-case timing of software (WCET) [18]. Estimating WCETs requires a knowledge of both the flow of control of the program and the timing of the processor the program is running on. To make the estimation of WCET possible, it is required that the hardware in use is amenable to analysis. Current generation processors incorporate many performance enhancing features like pipelining, caching, parallel execution of instructions and branch prediction. Among these features, the modeling and WCET analysis of branch predictors have received little research interest.

Branch prediction techniques [22] aim at reducing the penalty of executing branch instructions in pipelined processors. A pipelined processor must fetch the next instruction before the current one has finished executing. If the current instruction is a conditional branch, the processor must decide whether to fetch the next instruction from the target address (assuming the branch is taken) or to fetch it from the next address in sequence (assuming the branch is not taken). An incorrect guess causes the pipeline to stall until it is refilled with a valid instruction. The role of the branch prediction mechanism is to guess the instruction to be fetched next.

In *static* branch prediction schemes, every branch instruction is always predicted the same way whenever it is encountered. The simplest static branch predictors assume the same prediction for all branches, whatever their type (e.g. assume that a branch is always not taken). Improvements take into account the direction of the branch: backward branches typically terminate loop iterations and thus are frequently taken. The BTFN (Backward Taken, Forward Not taken), used for instance in the MicroSparc-2 processor uses such a static branch predictor. Extensions to this scheme allow the compiler, through an instruction-set hook, to statically predict a branch as taken or not taken. Such *compiler-directed schemes* are found for instance on the PowerPC, UltraSparc, Intel i960 and MIPS R4x00. Obviously, all these static techniques are easily amenable to WCET analysis since predicted branches are known at compile time. Existing compiler-directed schemes predict the outcome of branches either using profile data [6] or static program analysis [16]. Other studies also consider the use of static branch prediction to enhance the performance of hybrid branch predictors [7, 15]. All compiler-directed static prediction schemes we are aware of methods aim at minimizing the (average-case) misprediction rate, which is not the most relevant metric in real-time systems.

Reaching higher prediction rates requires to employ *dynamic* prediction schemes, which consider the *history* of branches to make their prediction at run-time. The simplest form of dynamic branch predictor is the *single-level* predictor. In such a scheme, a branch history table tracks the previous behavior of the branches of the program. For each branch, a 1-bit or 2-bit saturating counter tracks if the

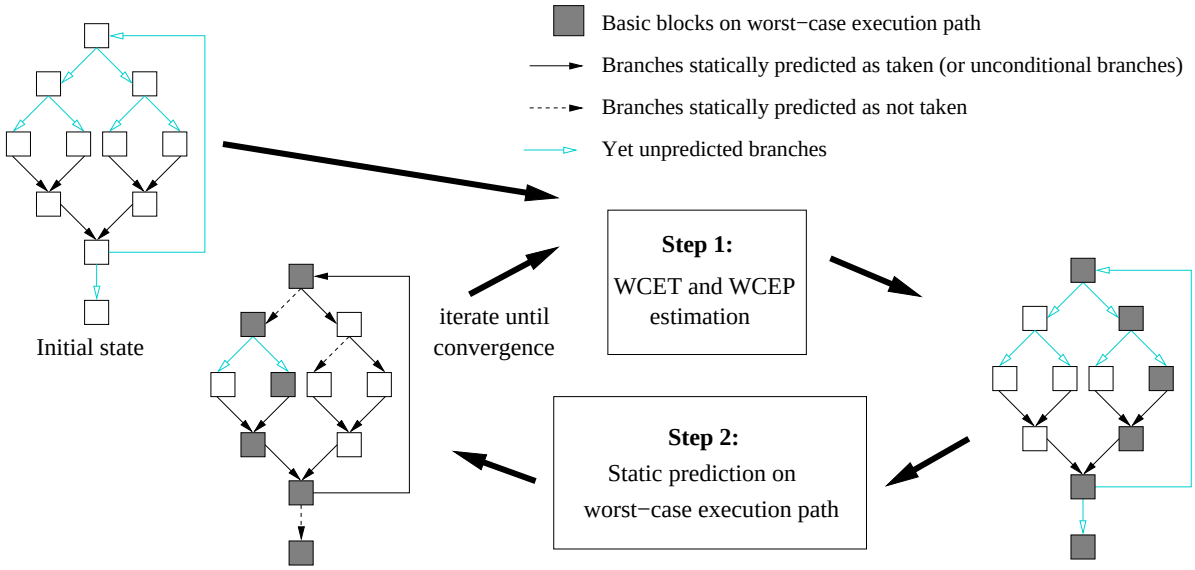

Figure 1. Principle of the algorithm

branch has been taken or not previously. The prediction is then made according to the counter value. Colin and Puaut have shown in [4] that such branch predictors are amenable to modeling and static WCET estimation. More recent processors include the more efficient *two-level* branch prediction schemes. In two-level predictors, the recent history of branches is tracked in order to detect patterns of how branches behave and/or are correlated with each other. The history is used together with the branch address to perform a lookup into a branch prediction table which makes the final decision. The branch prediction table typically contains 2-bits counters. Two-level branch predictors are much harder to analyze than one-level branch predictors, since they rely on the history of executions, which is in general hard to obtain statically. The history can be local (one per branch) or global (a single history for all branches). Very few studies have dealt with WCET analysis of processors with two-level branch predictors. Mitra and Roychoudhury [11] deal with a global-history branch predictor, but their work is limited to one-bit counters and the number of constraints generated to estimate WCETs is important. Bate and Reutemann [3], in the second part of their paper, consider global-history two-level branch prediction schemes but analyze only a subset of conditional branches, qualified as *easy-to-predict*. In particular, conditional branches within loops cannot be analyzed since their branch history cannot in general be known statically. Moreover, [3] assume that there are no conflicts in the branch history table, while it is known that destructive aliasing is the most important limiting factor on global history prediction accuracy [24].

These works on the static analysis of dynamic branch predictions [11, 3], together with the measurements presented by Engblom in [5] and the recent advances in dynamic branch prediction, mixing the use of distinct history lengths [8, 21, 1] support the position that the modeling and worst-case analysis of dynamic branch prediction is becoming harder and harder, and may soon be impossible for the most complex branch predictors.

In general, compiler optimizations aim at reducing programs average-case execution times. A recent and less explored research area for real-time systems is to optimize WCET instead, for instance by code reordering or code duplication techniques [27, 26]. Our work is part of that research direction on WCET-oriented compilation.

3 A WCET-oriented static branch prediction algorithm

The basic principle of our scheme is, for processors with support for compiler-directed branch prediction, to *statically predict* the conditional branches having an impact on the program WCET, such that the *program WCET estimate is minimized*. In order to avoid exploring all possible static predictions (2^n, with n the number of conditional branches), we use a greedy iterative procedure operating on the program control flow graph (see Figure 1).

Initially (top part of the figure), all conditional branches are not assigned a static prediction and are conservatively considered mispredicted (grey arrows in the figure). The algorithm then works iteratively (bottom part of the figure). At every iteration, in a first step, the program WCET, as well as the program worst-case execution path (WCEP) are identified (grey boxes in the figure). Then, in a second step,

```
1  procedure set_predictions
2  bool converged=false; int dir;
3  begin
4    forall BB ∈ CFG do BB.predicted=false; end;
5    // Step 1: WCET estimation
6    WCEP = estimate_WCET(CFG);
7    while not converged do
8      // Step 2: Issue static branch predictions along the worst-case execution path
9      forall BB ∈ WCEP do
10       if (branch(BB) is a conditional branch) then
11         if (BB.fall-through ∈ WCEP.branches) and (BB.taken ∈ WCEP.branches) then
12           if (max_count(BB.taken) ≥ max_count(BB.fall-through)) then dir=taken; else dir=fall-through; end;
13           if (not BB.predicted) then BB.predicted=true; BB.direction = dir; end;
14         else
15           if (BB.taken ∈ WCEP.branches) then dir=taken; else dir=fall-through; end;
16           if (not BB.predicted) then BB.predicted=true; BB.direction = dir; end;
17         end
18       end
19     end forall
20     // Step 1: WCET estimation
21     WCEP = estimate_WCET(CFG);
22     if (∀ BB ∈ WCEP BB.predicted) then converged=true end
23   end while
24 end
```

Figure 2. Algorithm for static branch prediction

all yet unpredicted conditional branches appearing on the WCEP are statically predicted. This way, the misprediction penalty is not tagged on to the WCEP and hence do not inflate the program WCET. Predicted branches may change the WCEP, as depicted in the figure. The process (WCET estimation, branch prediction along the WCEP) is thus repeated until the WCEP becomes stable.

3.1 Algorithm

The algorithm for statically assigning branch predictions is given in Figure 2, (procedure *set_predictions*). The following notations are used. The program control flow graph is denoted by *CFG*; no specific assumption is made on the CFG, except that it is known at compile-time. We denote the set of edges in the control flow graph (each of them corresponding to a control transfer instruction) by *CFG.branches*. If *BB* is a basic block, we note: *branch(BB)* the branch terminating the basic block; *BB.taken* the jumping edge from *BB*; and *BB.fall-through* the edge corresponding to the execution of the next address in sequence.

Each basic block *BB* in the program is assigned a boolean value *BB.predicted*. *BB.predicted=false* means that *BB* has not been assigned a prediction yet, meaning that both *BB.taken* and *BB.fall-through* are conservatively considered mispredicted (a miprediction penalty is considered for both branch directions). *BB.predicted=true* means that *BB* has been assigned a prediction, and in this case *BB.direction* denotes the predicted direction (*taken* or *fall-through*). In the algorithm, *WCEP* denotes the worst-case execution path of the program for a given configuration of predicted branches (*WCEP* is a subgraph of *CFG*). Function *estimate_WCET(CFG)* denotes the WCET estimation of the program thanks to its CFG; it returns the program *WCEP*; *max_count(E)* gives the number of executions of an edge (branch) E along the *WCEP*[1].

Initially, all branches are considered mispredicted (line 4), leading to a potentially large WCET estimate.

The algorithm then works iteratively. Every iteration (lines 8-19) considers all basic blocks of the program WCEP. The objective of every iteration is to decrease the program WCET estimate. This is achieved by predicting as taken the branches that are executed along the program WCEP. If both the *taken* and *fall-through* edges are on the WCEP[2] the edge appearing the most often on the WCEP is predicted (lines 12 and 13). Otherwise (lines 15 and 16) the

[1] For IPET WCET estimation tools, *max_count(E)* is a direct result of the ILP problem computing the WCET.

[2] This case typically appears in loops, in which both the "looping" branch and the "exit" branch belong to the WCEP.

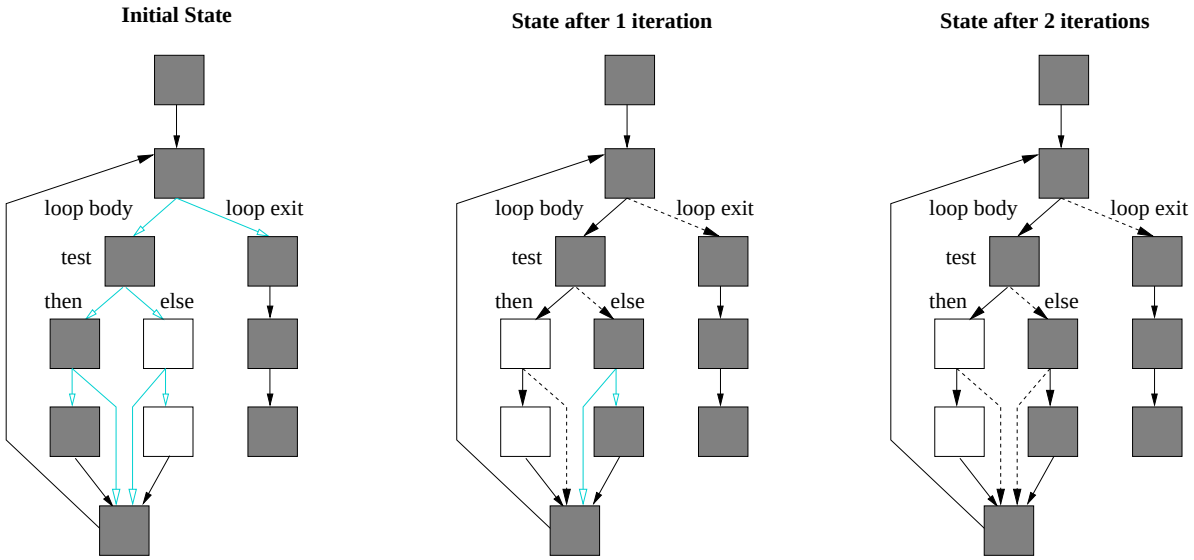

Figure 3. Illustration of the algorithm on a simple example

(unique) branch appearing on the WCEP is predicted. The algorithm is greedy. It never reconsiders a prediction made previously.

Obviously, the algorithm reduces the program WCET estimate as compared to the case where all branches are considered mispredicted. Indeed, a misprediction penalty is considered for only one direction of every conditional branch.

As already noted earlier, the WCEP may change at the end of every iteration of the algorithm. If so, the new WCEP is computed and the yet unpredicted branches on the new WCEP are assigned a prediction. Since assigned predictions are never reconsidered, the algorithm converges. Convergence is reached at worst when all branches have been statically predicted, i.e. when the WCEP is stable on two successive iterations. In practice, very few iterations are actually needed for most programs, as shown in Section 4.

Although the algorithm gives good results in practice, as shown in Section 4, we did not yet study formally the properties in terms of optimality with respect to WCET improvement. This is a direction for future work.

It should be noted that the algorithm statically predicts only a subset of conditional branches. Due to the way convergence is detected, the conditional branches that are not statically predicted are ensure never to be on the program WCEP. Thus, leaving open the issue of predicting statically or dynamically such branches is not important, since it can be safely considered that all such branches are mispredicted without impacting the program WCET estimate.

3.2 Example

Figure 3 illustrates the algorithm on a simple example, whose source code is composed of a loop whose body contains a small cascade of conditional statements. The same legend as in Figure 1 is used.

Initially (left part of the figure) all conditional branches are considered mispredicted (grey arrows); the WCEP is the path always taking the *then* branch. Branches are then statically predicted along the initial WCEP; the results of the prediction and the new WCEP being depicted in the middle part of the figure. On this example, the WCEP changes due to the just-made prediction, and now takes the *else* branch in the loop. The yet-unclassified conditional branches along this new WCEP are then classified; the result is shown in the right part of the figure. The WCEP then does not change anymore and the algorithm stops iterating.

4 Experimental evaluation

In this section we present the experimental results on a PowerPC 7451 architecture. We first give an overview of the target architecture and WCET analysis method. Then, we present the benchmark programs used. Finally, we give the experimental results.

4.1 Target architecture and WCET analysis

The experiments have been conducted on a PowerPC 7451 processor. The PowerPC provides both a dynamic and a static branch prediction mode, applied to all conditional

branches. When the processor is set in static branch prediction mode, one bit in the branch instructions indicates the direction to predict. If the static prediction is the *fall-through* direction and if that prediction is correct, no time penalty occurs; when predicting statically the *taken* direction, a timing penalty may still occur even in the case of a correct prediction. In this later case, the penalty is due to the fetch of the target instruction that may require 2 cycles (this penalty occurs when the target instruction is not available in the PowerPC 128 entry Branch Target Instruction Cache). If the static prediction is a misprediction then the misprediction penalty is constant and equal to 7 cycles.

The WCET analysis has been performed at the assembly code level, with the help of the binary code to extract variable and function addresses. Taking into account that the size of the data and code of the considered benchmarks fit into the instruction and data caches, the static cache analysis embedded in our WCET analysis tool was disabled. The method for computing the WCET is based on an Implicit Path Enumeration Technique (IPET) method as described in [9, 19], with *lp_solve* used as constraint solver. The instruction pipeline model is conservative: it is assumed that at most one instruction per cycle is issued (the PowerPC can issue up to 3 instructions per cycle). Penalty cycles are added if data dependencies exist between instructions. Due to the difficulty of modeling the complex dynamic branch predictor of the PowerPC 7451, dynamic branch prediction is assumed to always mispredict with a penalty of 7 cycles [13].

The considered timing model is simple, but safe. It may overestimate WCETs, which puts our scheme at a disadvantage. Defining a more precise albeit safe timing model for such a processor is a complex issue and is considered outside the scope of this paper.

4.2 Benchmarks

The benchmarks used cover different code structures:

Control 1: This code is control code (flight control) that mixes floating point computation, switches and conditional statements. It is automatically generated code from the SCADE suite.

Control 2: This code is similar to *Control 1* but contains smaller basic blocks. It comes from the WCET benchmark repository, program *statemate*[3].

Compress: This code is the *compress* code from the UTDSP benchmark code[4]. This code is loop oriented.

Routing: This code is a shortest path routing algorithm for networks. It mixes loops and control.

[3]available at http://www.c-lab.de/home/en/download.html#wcet.

[4]University of Toronto Digital Signal Processing (UTDSP) Benchmark Suite, 1992. Available at http://www.eecg.toronto.edu/.

Jpeg: This code is the Jpeg code from the Powerstone benchmark suite [20]. It mixes loops and control.

Table 1 gives for each benchmark (including the example program of Section 3.2) the number of assembly instructions (*nb_instr*), the number of basic blocks (*nb_BB*), the average size of a basic block in instructions (*BB_size*), the number of loops (*nb_loops*) and maximum loop nesting (*nesting*).

Name	nb_instr	nb_BB	BB_size	nb_loops	nesting
Example	90	11	9	1	1
Control 1	1843	84	21	0	0
Control 2	3508	343	10	0	0
Routing	417	44	9	8	2
Compress	917	103	8	12	4
Jpeg	2409	164	14	25	4

Table 1. Benchmark characteristics

4.3 Experimental results

Table 2 summarizes the experimental results in terms of improvement of the programs WCET estimates. The columns give respectively: the initial conservative WCET estimate, the WCET estimate after convergence, the corresponding improvement, and the number of iterations until convergence (*niter*).

Name	Initial WCET	Final WCET	Improvement ($\frac{Initial-Final}{Initial}$)	niter
Example	2383	1818	24%	2
Control 1	2108	2001	5%	2
Control 2	2637	2068	21%	2
Routing	87092	73974	15%	1
Compress	21990129	19705072	10%	2
Jpeg	3731691	3047613	18%	2

Table 2. Experimental results

As shown in the table, the WCET estimate can be improved by up to 21%, with an average value of 15%, as compared with the base conservative approach. Moreover, a maximum of two iterations was required on the considered benchmarks for the algorithm to converge, which shows that in practice the iterative algorithm converges rather quickly. The time to execute the algorithm of Figure 2 is negligible compared to the WCET analysis time (only a scan of the basic blocks along the WCEP is required).

The impact of mispredictions is larger for applications with a smaller basic block size. Therefore our scheme naturally provides better results for applications with short basic blocks.

It can be observed from the results in Table 2 that the size of applications has no impact on the performance of our scheme, i.e. our scheme is scalable with code size. This is because static predictions are made on a per-branch basis and thus branches are treated independently. Stated differently, our scheme is insensitive to the conflicts in prediction tables, since no analysis of prediction tables is required.

It may be noticed that the two cycles penalty occurring on the PowerPC when a *taken* branch is predicted could (at least partially) be removed by a modification of the program layout. The modification would consist, similarly to [27], in relocating basic blocks along the worst-case execution path such that conditional branches along that path are *fall-through* branches. This optimization would have to be implemented to evaluate if the improvement of WCET estimate is worth the implementation cost.

5 Implementation considerations

This section discusses implementation issues for the PowerPC 7451 architecture [12]. The processor can be configured to use static branch prediction for *all* conditional branches by clearing a bit (PHT bit) in a specific processor register (HID0). A bit in the instruction encoding of every conditional branch (fifth bit in the BO field of instructions) then identifies which direction of the branch is predicted (*taken* or *fall-through*).

Implementing our scheme on this processor thus simply requires to set/clear the prediction bits of conditional branches according to the prediction made off-line by the algorithm presented in Section 3. This can be achieved in a straightforward manner, either by rewriting the software binary/assembly code, or by slightly modifying the compiler back-end.

Our scheme only predicts branches along the worst-case execution path, so as to optimize the program WCET estimate. It leaves open the issue of optimizing or not the branches that are ensured to never be on the program WCEP. On the PowerPC 7451 architecture, at run-time *all* the conditional branches use the same prediction scheme (static or dynamic). Thus, it may be interesting to set/clear the prediction bit for branches with no impact on the program WCET estimate as cleverly as possible to improve the program behavior. For instance, one could consider using profile data, in the manner of [6] to reduce the average execution time. For architectures allowing to select between static and dynamic branch predictions on a per-branch basis (e.g. the IA-64), an alternative could consider using dynamic prediction for these branches, yielding better average-case performance than a purely static scheme, and this without any impact on the WCET estimate.

6 Concluding remarks

The increasing complexity of dynamic branch predictors make them less and less amenable to worst-case execution time analysis. Therefore, using dynamic branch predictors in hard real-time systems often means considering that all branches are mispredicted, thus adding another degree of pessimism in WCET estimation. To overcome this problem, we have proposed in this paper the use of static branch predictions for the branches having an impact on the programs WCET estimates. Furthermore, we have proposed an iterative algorithm to select the static branch predictions in order to minimize the programs WCET estimates. Experiments have shown that the estimated WCET can be decreased significantly compared with a conservative classification of all conditional branches as mispredicted. The best results are obtained for programs with small basic blocks, for which the impact of branch prediction is the greatest. The proposed scheme can be implemented rather easily in the compiler back-end. In addition, WCET analysis is simplified since the modeling and analysis of branch predictors is not required anymore. Our proposal, although applicable to any processor with support for static branch prediction, is specially suited to processors with complex dynamic branch predictors, which are less and less amenable to WCET estimation. Nevertheless, it should be noted that running applications in static prediction mode necessarily results in a loss in average-case performance, that still needs to be evaluated quantitatively.

More generally, we believe that many compiler optimizations, originally designed with performance in mind, cannot fully benefit to real-time systems, since they focus on average-case performance which is not the prime metric in real-time systems. Revisiting existing optimizations or designing new ones with worst-case performance in mind is an issue we wish to explore in the future. The work presented in this paper is a first step in that direction.

Acknowledgments

We wish to thank André Seznec, Jakob Engblom and Iain Bate for fruitful feedback on earlier drafts of this paper.

References

[1] *Branch Prediction Contest, Workshop in conjunction with the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro37)*, Portland, Oregon, Dec. 2004.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium (SAS'96)*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66, September 1996.

[3] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 215–222, Catania, Italy, July 2004.

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

[5] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 152–159, Toronto, Canada, May 2003.

[6] J. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural support for programming languages and operating systems (ASPLOS-V)*, pages 85–95, Oct. 1992.

[7] D. Grunwald, D. Lindsay, and B. Zorn. Static methods in hybrid branch prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) 1998*, 1998.

[8] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS95)*, pages 298–307, Pisa, Italy, Dec. 1995.

[10] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, Nov. 1999.

[11] T. Mitra and A. Roychourdhury. A framework to model branch prediction for WCET analysis. In *2nd Workshop on Worst Case Execution Time Analysis (WCET)*, Vienna, Austria, June 2002. Proceedings available as Technical Report YCS 346, University of York, UK.

[12] Motorola. *MPC7450 RISC Microprocessor Family User's Manual*, Dec. 2001. (http://www.motorola.com).

[13] Motorola. *MPC7450 RISC Microprocessor Family Software Optimization Guide, Application Note AN2203/D*, July 2002. (http://www.motorola.com).

[14] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[15] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *6th International Symposium on High-Performance Computer Architecture (HPCA6)*, pages 251–262, Jan. 2000.

[16] J. R. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI)*, pages 67–78, June 1995.

[17] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 114–123, Austin, Texas, Dec. 2002.

[18] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[19] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universitat, Institut fur Technische Informatik, Wien, Apr. 1995.

[20] J. Scott, L. H. Lee, and W. Moyer. Designing the low-power M.CORE architecture. In *Proceedings of Power Driver Microarchitecture*, June 1998.

[21] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design trade-offs on the EV8 branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, May 2002.

[22] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture (ISCA)*, pages 135–148, May 1981.

[23] J. Stankovic and K. Ramamritham. What is predictability for real-time systems ? *Real-Time Systems*, 2:247–254, 1990.

[24] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In 22^{nd} *Annual International Symposium on Computer Architecture*, pages 276–286, 1995.

[25] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, Oct. 1993.

[26] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by optimizing worst-case paths. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Mar. 2005.

[27] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET code positioning. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS04)*, Dec. 2004.