WCET Analysis by Model Checking for a Processor with Dynamic Branch Prediction

Armel Mangean, Jean-Luc Béchennec(✉), Mikaël Briday, and Sébastien Faucou

CNRS, École Centrale de Nantes, Université de Nantes, LS2N,
44000 Nantes, France
jean-luc.bechennec@ls2n.fr

Abstract. In this paper, we investigate the case for model checking in the WCET analysis of pipelined processors with dynamic branch and target prediction. We consider a microarchitecture inspired by the e200z4 Power 32-bit architecture, with an instruction cache, a dynamic branch prediction mechanism, a branch target buffer (BTB) and an instruction prefetch buffer. The conjoint operation of all these components produce a very complex behaviour that is difficult to analyse with tight and sound static analysis techniques. We show that model checking techniques can actually be used to compute WCET bounds for this kind of architectures.

1 Introduction

Embedded control systems found in domains like automotive, industrial automation, or robotics, have to satisfy real-time requirements stemming from the dynamics of the physical plant they control. To design these systems, the worst case execution time (WCET) of the tasks must be computed. The execution time of a task is a function of its inputs and the initial state of the microarchitecture. As it is usually not possible to run the real system with all possible combinations of these variables, techniques have been developed to statically estimate upper bounds on the WCET [17].

In this context, different approaches have been investigated. One of them is model checking. Using model checking in the context of WCET analysis has been debated in the scientific community. In [16] it is deemed as ineffective because of the state space explosion problem. In [14], it is claimed that it can actually improve the precision of WCET analysis by leveraging dynamic analysis[1] of microarchitecture features. Both points actually hold: model checking allows to compute more precise bounds but suffers from scalability issues. However, recent results show that model checking sufficiently scales to tackle the WCET analysis of systems based on core such as ARM7 or ARM9 [2,4].

In this paper, we investigate the case for model checking in the WCET analysis of architectures typically found in embedded control systems. We consider a core architecture inspired by the e200z4 Power 32-bit architecture. More precisely,

[1] Metzner use dynamic analysis to designate techniques that analyze concrete paths in the system, as opposed to static analysis that consider abstract paths.

K. Barkaoui et al. (Eds.): VECoS 2017, LNCS 10466, pp. 64–78, 2017.
DOI: 10.1007/978-3-319-66176-6_5

we consider a microarchitecture with an instruction cache (ICache), a dynamic branch prediction mechanism, a branch target buffer (BTB), a prefetch instruction buffer, and a 5-stage pipeline[2]. The conjoint operation of all these components produce a very complex behaviour that is difficult to analyse with tight and sound static analysis techniques. We show that model checking techniques can actually be used to compute WCET bounds for this kind of architectures.

Contribution and Outline. To the best of our knowledge, this paper is the first to propose an analysis integrating at the same time the ICache, the branch target buffer, and the instruction prefetch buffer. Among the work exploring model checking for WCET analysis, it is the first to tackle a dynamic branch prediction mechanism. Based on this analysis, we also provide an evaluation of the impact of dynamic branch prediction and BTB on the estimation of WCET for embedded control systems.

The paper is organized as follows. In Sect. 2 we provide some background and summarize related works. In Sect. 3 we describe our target microarchitecture. In Sect. 4 we describe our WCET analysis framework. In Sect. 5 we give some insights on the models developed for the dynamic analysis of the target microarchitecture. In Sect. 6 we report an evaluation based on benchmarks. In Sect. 7 we conclude the paper.

2 Background and Related Works

2.1 Branch Prediction Basis

In modern processors, pipelines are used to improve the instruction execution rate by executing simultaneously different stages of several instructions at the same time. Each cycle, one (or more in the case of superscalar processor) instruction is fetched sequentially from the memory and fed into the pipeline. When a branch instruction is executed, the outcome (whether the branch is taken or not, and what is the actual target) is usually not known in the lower stages of the pipeline. Thus, bubbles[3] are inserted in these stages until the address of the next instruction is available. These delays are control hazards and have an impact on the execution time.

Branch prediction is a set of techniques used to minimize the occurrence of this situation. It consists in trying to predict the outcome of a branch instruction when it gets into the pipeline in order to fetch the correct following instruction with a high probability. The simplest form of branch prediction is static branch prediction based on the program code only. A straightforward prediction algorithm is to predict all branches as always not taken. If the prediction is correct, no cycle is lost. If the prediction is incorrect, the lower stages of the pipeline

[2] The main difference is that the e200z4 is actually a two issues statically scheduled superscalar processor, whereas we consider a single issue processor.

[3] A bubble, or pipeline stall, is a delay cycle. When a bubble enters a stage, this stage has no activity during the current cycle.

have to be flushed to mimic the insertion of bubbles. A more efficient algorithm widely used is to predict forward branches (conditional statements) as always not taken and backward branches (loops) as always taken.

Dynamic branch prediction uses runtime information to further improve the prediction accuracy. There is a wide variety of dynamic branch prediction algorithms that cannot be covered here (see [5] for an overview). From now on, we will focuses on the algorithm analysed in this paper. It is based on a set of 2-bit saturating counter as illustrated on Fig. 1. From left to right, the four states are usually called: taken (11), weakly taken (10), weakly not taken (01), not taken (00). A counter is associated with a branch instruction. It is initialized either statically or after the first execution of the branch. Then, the state evolves according to the actual outcomes of the execution of the instruction: it is incremented when the branch is taken and decremented otherwise.

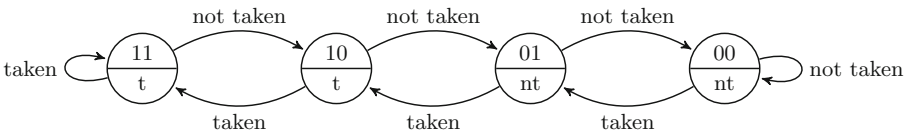


Fig. 1. State machine of a 2-bit saturating counter. Outputs: `t` for taken, `nt` for not taken. The initial state is implementation dependant.

Each of these counters is usually stored along with the target of the branch instruction in an entry of a small cache memory: the BTB. A BTB entry is commonly retrieved with a combination of index and tag computed from the branch instruction. When the BTB is full, a replacement policy is applied to free an entry.

2.2 Analysis of Branch Prediction Techniques

An important body of work is related to WCET analysis for processor with dynamic branch prediction (see for instance [1,3,8,12,15]). Most of these works focus on the analysis of branch prediction (whether a branch is taken or not) but except for [3,8], do not take into account branch target prediction (whether the target of the branch is in the BTB or not). Only [8] analyzes the interactions between the BTB and the instruction prefetch buffer which is mandatory to take into account fine grain penalty for misprediction. When this interaction is not analyzed, a uniform penalty must be used to account for misprediction, and the hypothesis of the timing compositionality of the architecture must be implicitly assumed (*i.e.* the analysis can safely follow local worst case path only [18]). Lastly none of these works tackle the problem of analyzing the interactions between the ICache and the branch prediction mechanism. Our approach integrates in a single analysis the ICache, the BTB, and the pipeline using an instruction prefetch buffer.

2.3 Model Checking and WCET Analysis of Processors

There is a limited body of work on model checking techniques for the WCET analysis of processors [2,4,10,14]. In [2,4], it is shown that UPPAAL, a state-of-the-art symbolic model checker for (networks of) timed automata, can be used to model and analyze real-life processors. The target processor of these works feature instruction and data caches and an in-order 5-stage pipeline without dynamic branch prediction. Our target processor features an ICache and an in-order 5-stage pipeline with a BTB and an instruction prefetch buffer. Our ICache model is original, but close to the model used in [2]. Our pipeline model is fully original, as it integrates an instruction prefetch buffer and interactions with an original BTB model.

Following [2], to improve scalability, our analysis framework uses program slicing to narrow the set of instructions and memory locations that must be accurately modeled in order to compute a sound bound. To improve modularity, we use a standalone, state-of-the-art, program slicer for binary code [13].

3 Description of the Target Microarchitecture

Our target microarchitecture is inspired by the Qorivva MPC5643L microcontroller [7]. It is a dual-core developed for safety critical applications of the automotive domain. The architecture is based on two e200z4 Power cores [6]. The e200z4 core is a 32-bits processor of the Power family, based on the PowerPC instruction set. In this paper, we focus on the model of one core.

3.1 Memory Hierarchy

The Qorivva SoC classically embeds internal (S)RAM and flash. The RAM uses a 32-bit data bus and no data cache is available on the e200z4 core. The flash stores the program instructions and is connected to the ICache using the AHB interface[4]. It supports 64-bit data bus for instruction fetch and 32-bit data bus for CPU loads and DMA access. A burst mode allows to fill cache lines faster by sending only the start address to the flash memory controller and get the data flow of sequential access to memory, instead of reading one 64-bit value at a time and then requesting the data at the following address and so on.

The ICache size is 4Kbyte with 32 bytes lines. It can be configured either has a 2- (64 sets) or 4-ways (32 sets) associative cache. The replacement policy is pseudo round-robin. A global register shared among sets points to the next way to replace. The register is incremented for each cache miss modulo the number of ways.

The cache is non-blocking so that the execution continues during a cache miss. On a cache miss, four 64-bits memory accesses are required to fill a cache line. These accesses are done starting with the required instruction to decrease access

[4] The Advanced High-performance Bus is part of the open standard ARM-AMBA on-chip interconnect specification.

latency. A line fill buffer stores the memory words retrieved from the memory and the cache line is updated as soon as the line fill buffer load completes. Moreover, a hit under fill feature is implemented to check the line fill buffer instead of waiting for the cache line update.

3.2 Execution Pipeline and Instruction Prefetch Buffer

The e200z4 is a 2-issue static scheduling superscalar core. Instructions are executed on a 5-stage pipeline. The *fetch stage* gets the instruction code from the cache using a 64-bits memory bus. It can retrieve up to two 32-bits instructions to feed them to the decode stage. Fetched instructions are stored in a 32 bytes instruction buffer (8 32-bits instructions). When an instruction enters the fetch stage, the program counter (PC) is updated with the address of the next instruction to fetch. If the instruction is a branch, a BTB lookup is performed. In case of a hit, dynamic branch prediction applies (see below). If the prediction is to take the branch, PC gets the predicted target. Otherwise, PC gets the next address in sequence.

The *decode stage* decodes up to two instructions from the instruction buffer, determine instructions requirements and check register dependencies[5]. In the case of a branch instruction, if it was not found in the BTB when entering the fetch stage, static prediction policy is applied. If the prediction is to take the branch, then the lower stages of the pipeline, including the instruction buffer, are flushed and PC is updated to the target of the branch. The instruction buffer will be refilled either at the next pipeline stall caused by a data or structural hazard (lack of hardware resources).

The next two stages are either the *execute stages* or *data memory accesses stages*. In the case of a branch instruction, the actual outcome of the branch is resolved here. According to the match between the prediction and the actual outcome, the BTB and PC are updated. If the prediction was incorrect, the lower stages of the pipeline, including the instruction fetch buffer, are flushed and PC is updated to the correct address. This is an in-order execution and the last stage is the *write back stage* to update registers.

3.3 Branch Prediction

The branch unit integrates a fully associative 8-entry BTB. The branch unit mixes static and dynamic prediction. Static prediction is used when the branch is not known, *i.e.* not allocated in the BTB. It can be configured to use either the *always not taken* (AN) policy, or the *backward taken forward not taken* (BTFN) policy presented in Sect. 2.1. Dynamic prediction uses a 2-bit saturating counter. Thus each entry of the BTB contains a tag (the full address of the branch

[5] In the case where the instruction in the decode stage requires a result produced by an instruction ahead in the pipeline, bubbles are inserted until the availability of the result. This is a data hazard. Bypasses are used between stages to propagate results and limits these bubbles.

instruction), a 2-bit saturating counter and the target address. In the case of a BTB miss, if the branch is resolved as taken, it is allocated in the BTB using a FIFO replacement policy and its counter is initialized to *weakly taken*. Its target address is also stored.

The reference manual of the e200z4 core does not provide information on the prediction of computed branches, *i.e.* branches for which the target is computed at runtime. This is typically the case of function return, switch statement, or function pointer. In this paper, we consider that these branches are handled in the same way as the other ones. According to this interpretation, for these branches, the branch prediction can be correct and at the same time the target prediction incorrect because the BTB stores the last target address of the branch.

All in all, there are 9 different outcomes for this branch prediction mechanism. They are summarized in Table 1. Notice that in case of misprediction, the instruction buffer has to be flushed. In this case, a memory access is triggered to refill this buffer. In turn, this access can add extra latency when the target instruction is not already in the ICache.

Table 1. The 9 different cases of branch prediction. The given penalties are lower bound corresponding to the case where all involved instructions are already in the ICache. $^{+}$: this case triggers a flush of the instruction buffer because the branch is predicted taken in the decode stage. *: this case triggers a flush of the instruction buffer because of misprediction detected in the execute stage.

BTB	Hit					Miss			
Prediction	Taken			Not taken		Taken		Not taken	
Correct prediction	Yes		No	Yes	No	Yes	No	Yes	No
Target prediction	Correct	Incorrect							
Penalty (in cycle)	0	2^{*}	2^{*}	0	2^{*}	1^{+}	$2^{+,*}$	0	2^{*}

3.4 Analyzability and Predictability

The main challenge concerning the WCET analysis of this architecture lays in the complex interactions between the ICache, the branch prediction unit, and the instruction prefetch buffer. To compute a safe and tight bound on the WCET, an integrated analysis of these 3 units is required. Indeed, only an integrated analysis allows to compute the actual sequence of memory accesses requests and pipeline stall states produced by a given run of the program.

4 Our WCET Analysis Framework

As shown in Fig. 2, our analysis framework is built around two tools: BEST [13] and UPPAAL [11].

BEST is a program slicing tool for binary code. It is interfaced with a disassembler library for the target instruction set architecture (PowerPC 32-bit in

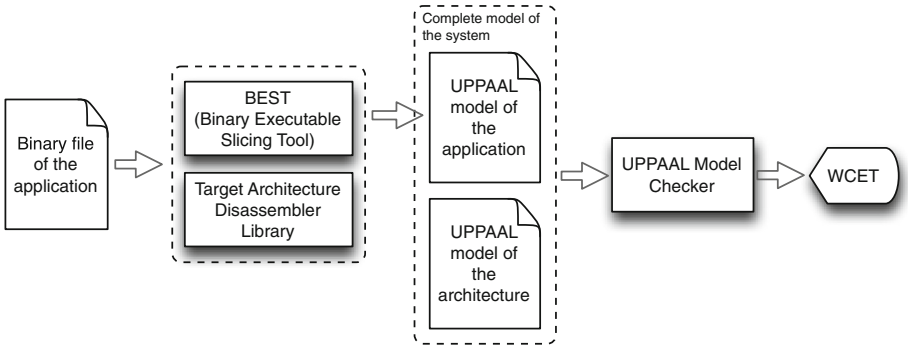


Fig. 2. The UPPAAL model of the application is generated by the BEST tool, from the application binary file. This model is synchronized with the hand-written model of the architecture (that does not depend on the application considered). The WCET is then computed by model checking.

this paper). For each instruction of the program, the disassembler provides a set of semantics information including the opcode, the arguments, the type (branch or not), the set of used and defined registers, etc. BEST uses this information to slice the program, with all the branch instructions as the slice criterion. The resulting sliced program contains all and only the instructions that have an impact on the execution flow. This sliced program is then used to produce a control flow graph where each instruction is tagged as either in or out of the slice. From the set of instructions in the slice, the set of useful memory locations to analyze the control flow of the program is computed. This information is then used to generate an UPPAAL model of the program (see Sect. 5.1). The program slicing is a mandatory phase to limit the state space explosion problem.

The second step consists in analyzing the system with UPPAAL. UPPAAL is a model design and verification tool for networks of timed automata (NTA). Timed automata (TA) are finite state automata augmented with real-valued clocks. The values of these clocks all increase at the same rate. Linear constraints on clocks can be used to guard transition, and clocks can be reset when a transition is taken. In UPPAAL syntax, TA can use boolean and bounded integer variables. These variables can be manipulated through functions specified in a language with a C-like syntax. Moreover, TA can be synchronized over synchronous channels to form a NTA.

In our framework, we have developed a set of TA corresponding to the components of the microarchitecture. These components interact through global variables and synchronizations. The corresponding models are briefly described in Sect. 5. The model of the program generated by BEST is synchronized with the models of the microarchitecture. The resulting NTA models the whole system, hardware and software. This model contains a specific clock reset one time only, at system startup. The model checker is then used to perform a symbolic exploration of the state space of the system and computes the maximal value reached

by this clock over all paths. Our framework (BEST, UPPAAL models and script files) used to produce the experimental data are distributed in open-source[6].

5 Models

Figures 3, 4a, b and 5 show UPPAAL models. Location labels, invariants, guards, synchronisations and updates are displayed respectively in purple, pink, green, cyan and blue.

5.1 Modeling the Program

The model of a program is composed of two main parts: an array of data structures that will be fed to the model of the pipeline to mimic the timing behavior, and an automata to mimic the functional behavior.

A data structure is associated with an instruction of the program. It contains constant information like the instruction address, the number of cycles required to execute the instruction in the execute stage, a flag indicating whether the instruction is a branch instruction and if applicable its target address, a flag indicating whether the instruction is a memory access instruction and the set of defined and used registers.

It contains also a dynamic part called the instruction runtime data structure. For a standard instruction, it contains only the number of remaining execution cycles. In the particular case of a branching instruction, 3 flags are also used: whether the instruction has been predicted taken or not taken, whether the prediction was static or dynamic, and lastly, whether the branch has actually been taken or not taken.

The automata is built from the control flow graph of the program, as illustrated in Fig. 3. In this automata, each location models a breakpoint before the execution of the corresponding instruction and each outcoming transition is associated with the functional effect of the execution of the instruction.

The label identifies the instruction: thus `BB`x`_Inst`y denotes the y^{th} instruction of the x^{th} basic block of the program. The guard `InE(Inst`y`)` tests if this instruction is currently in the execute stage of the pipeline. The `Tick` signal is used to synchronize the program model with the pipeline update. The update of the transition calls a UPPAAL function that execute the semantics of the instruction. It consists in updating the global variables that model the content of the memory and the status flags of the processor. If the instruction is not part of the slice, then the transition does no update as executing its semantics has no influence on the temporal behavior of the system (*e.g.* the outgoing transition of location `BB10_Inst10` on Fig. 3). For conditional branches, two transitions are provided, corresponding to the two cases: taken or not taken. For these transitions, the guard is completed with a test to select the correct path according to the status of the processor (*e.g.* `nz()` or `!nz()` on the two outgoing transition of location `BB10_Inst12` for Inst12 on Fig. 3).

[6] Available at https://github.com/TrampolineRTOS/BEST.

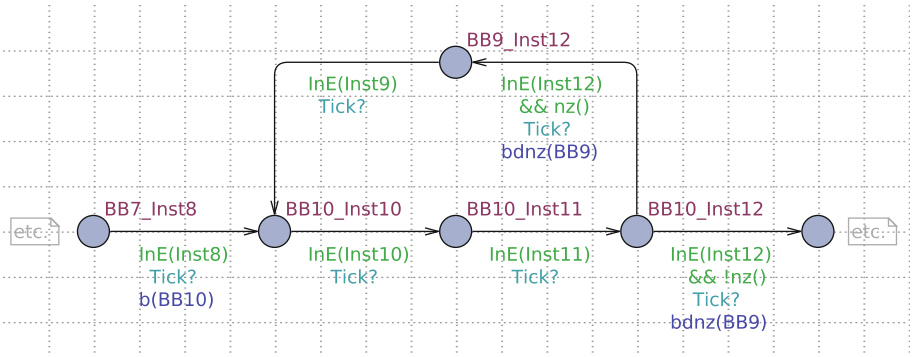
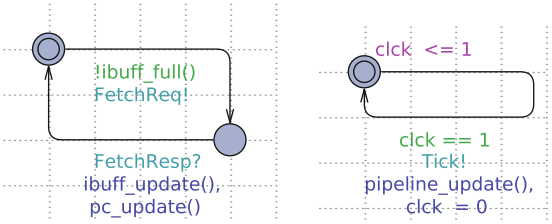


Fig. 3. Part of an automata modeling the functional behavior of a program. The `bdnz` instruction is a branch instruction that decrements a counter register (`CTR`) and branches if this counter is not null. (Color figure online)

5.2 Modeling the Pipeline

The model of the pipeline is composed of a set of data structures that capture the content of the internal memory of the components such as the pipeline stages, the instruction buffer and the BTB; and a set of automata used to synchronize the update of this data structures with the flow of time in order to mimic the timing behavior of the system.



(a) Fetching process control

(b) Pipeline execution control

Fig. 4. Automata controlling the pipeline. (Color figure online)

The first automaton (Fig. 4a) is associated with the (pre)fetching process. When the instruction buffer (IBuff) is not full, it tries to fetch an instruction from the instruction cache. When an instruction is fetched, the function `ibuff_update` updates the instruction buffer and the BTB. The instruction buffer is an array of instruction runtime data structures. The BTB is a circular buffer with each entry composed of a tag and a 2-bit saturating counter. Notice that the target adresses are not actually stored in the BTB because they are already in the

program automata. The `pc_update` updates the PC. It performs a BTB lookup in the case of a branch instruction.

The second automaton (Fig. 4b) controls the execution of the pipeline. Thanks to the invariant `clck <= 1` it generates a `Tick` signal every one unit of time. This is used to synchronize the NTA to the frequency of the pipeline. It also calls the `pipeline_update` function used to update the data structure and variables associated with the pipeline stage following the fetch stage. Each pipeline stage is associated with an instruction runtime data structure. Updating the stages consists in making instructions progress by updating the data structures from stage to stage (including the instruction prefetch buffer), taking into account the possible stall cycles due to data or structural hazards.

Static branch prediction is done when a branch instruction enters the decode stage, if no dynamic branch prediction was available for this instruction at the fetch stage. If the prediction is taken, the lower stages are flushed and PC is updated.

Branch target resolution is done when a branch instruction enters the execute stage. This encompass a potential BTB update, and, in case of misprediction, a flush of the lower stages and a PC update.

5.3 Modeling the Memory Hierarchy

The model of the memory hierarchy is composed of a set of data structures and global variables to track the content of the instruction cache and store the content of the useful memory locations (as computed during the program slicing step), and a set of automata to mimic the access times. In this paper, we focus on branch prediction so for the sake of clarity we will not give too many details on these models.

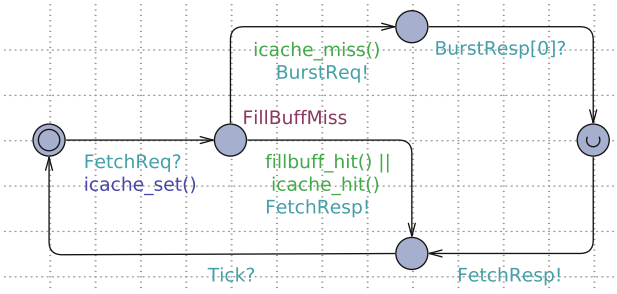


Fig. 5. Instruction cache access time model. (Color figure online)

The automata used to mimic the ICache access time is shown Fig. 5. When a fetch request is received, a cache look up and update is performed by the function `icache_set`. If the instruction is present in the cache or in the fill buffer, the request is acknowledged. Otherwise, a request is sent to the Flash memory. As

explained in Sect. 3.1, the instruction cache line fill requires 4×64 bits memory transactions from the flash, using a burst access. The request is acknowledged before the end of the burst, as soon as the instruction is in the fill buffer. At the end, a synchronization on the `Tick` signal enforces the cycle required to transfer an instruction from the cache to the fetch stage.

6 Experimental Results

We have conducted a set of experiments with the framework described above. The main goals of these experiments are (i) to assess the applicability and scalability of model checking for computing WCET estimation for embedded control systems; and (ii) to evaluate the impact of the branch prediction policy on the WCET.

We used the Mälardalen WCET benchmarks [9] to generate the programs. We excluded certain programs to account for the current limitations of our framework:(i) TAs are not fit to model recursive programs; (ii) our model of the architecture does not manage floating point arithmetic instructions; (iii) BEST does not manage binary executables with indirect branch instructions other than function return instructions (*i.e.* switch-case statements and function pointers); (iv) BEST does not manage slices where instructions depend each others through local variables located on the program stack. This point will be addressed in the future.

We built the binaries with Gcc 5.3.1. Without optimization, Gcc generates code where local variables are loaded from and stored to the stack frame each time they are used. Such binary executables can not be processed by the current version of our framework. Options `-O1` and `-O2` force Gcc to output optimized code that uses registers to load and store local variables. Thus we created two versions of each of the 14 Mälardalen benchmarks fitting our constraints, except for `cnt.c`, `insertsort.c` and `ud.c` which make use of the program stack when compiled with option `-O1`. All in all, we have built 25 binaries and for each one we ran our framework to compute its WCET on an Intel Core i7-3770 (4 cores, 3.40 GHz) with 8 GiB of RAM running Debian 9 (64-bit, Linux 4.9). Each time, we also collected the number of explored states, the time taken by UPPAAL to perform the exploration, and the amount of memory used. The results are summarized in Table 2 and Fig. 6.

Table 2 displays raw data from UPPAAL for models implementing the static always not taken branch prediction policy (AN) which is the worst wrt. resource consumption during the analysis. The worst case for each column is highlighted in bold. It is obtained for the binary built from the program `fir.c` compiled with `-O1`. Even in this case, both the analysis time (less than 5 s) and the amount of memory used (less than 640 MiB) are very reasonable. Our conclusion is that model checking seems to be a promising solution to compute WCET for this type of system. Further experiments should be performed to identify the limits of the scalability of the approach.

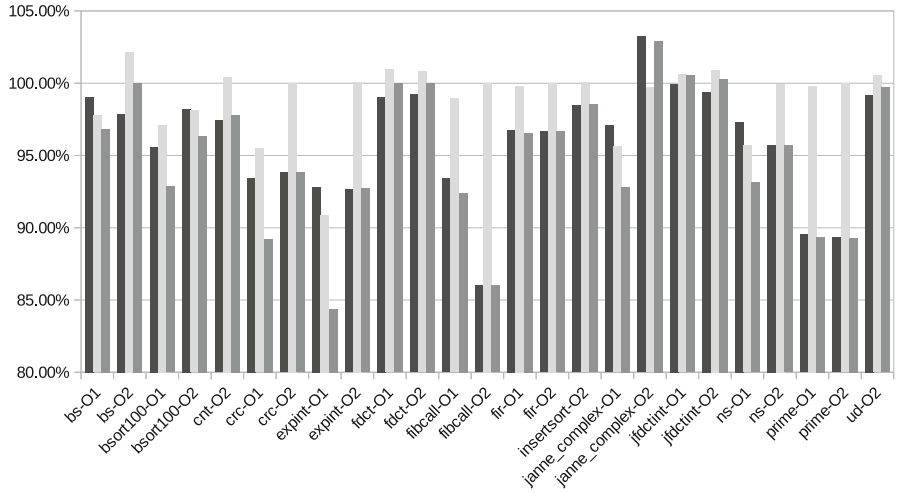
Figure 6 presents the impact of the branch prediction policy on the WCET (Fig. 6a) and the state space (Fig. 6b). Each bar represents the ratio between

Table 2. Consumption of resources by the analysis for the AN prediction policy.

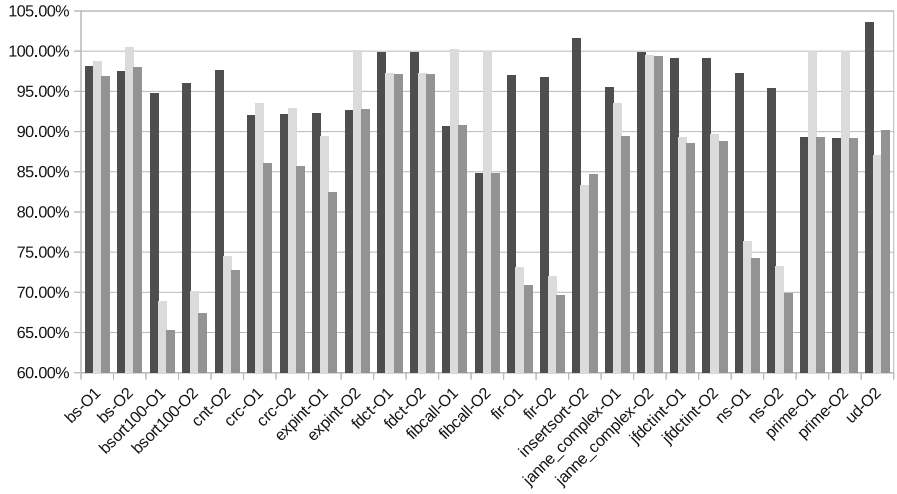
Program	States explored	CPU time (ms)	Memory (KiB)
bs-O1	586	10	12684
bs-O2	451	0	12140
bsort100-O1	12457	130	29192
bsort100-O2	11982	130	27868
cnt-O2	11279	110	30504
crc-O1	157612	1600	550048
crc-O2	144653	1570	494760
expint-O1	5967	40	22236
expint-O2	4118	40	13860
fdct-O1	6823	50	23804
fdct-O2	7080	50	25532
fibcall-O1	949	10	12104
fibcall-O2	590	0	11000
fir-O1	**728321**	**4770**	**655628**
fir-O2	692704	4430	608412
insertsort-O2	2995	20	14116
janne_complex-O1	779	10	12512
janne_complex-O2	594	0	12572
jfdctint-O1	11121	100	31636
jfdctint-O2	11349	100	33796
ns-O1	32482	250	44964
ns-O2	31229	230	40276
prime-O1	12072	80	26560
prime-O2	12056	80	25564
ud-O2	11305	380	491300

two policies: BTFN over AN in dark gray, BTFN+BTB over BTFN in light gray, and BTFN+BTB over AN in medium gray. For instance the ratio between the WCET computed for program `expint-O1.c` with the policy BTFB+BTB (1944 cycles) and the WCET computed for the same program with the policy AN (2304 cycles) is 84% (medium gray bar in slot `expint-O1` of Fig. 6a). When a bar is above 100%, it means that the numerator policy performs worst than the denominator policy. On the contrary, if the bar is below 100%, it means that it performs better.

Concerning the WCET, we first remark that most ratios are smaller than 100%. It means that branch prediction policies designed to improve the average case also have a positive impact on the WCET. Second, we remark that no branch prediction policy dominates the others: for each case, we have at least



(a) WCET ratios



(b) State space ratios

Fig. 6. Impact of the branch prediction policy on the WCET and the size of the state space. Each bar represent the ratio between two policies for a given binary: BTFN over AN in dark gray, BTFN+BTB over BTFN in light gray, BTFN+BTB over AN in medium gray.

one bar below the 100% threshold and one bar above. In the context of WCET analysis, it means that there is no worst policy that could be used to always estimate a worst case upper bound.

Concerning the state space, we remark that adding the BTB to the model of the architecture does not results in an increase of its size. On the contrary we note that the average number of states decreases while using a model simulating a more complex behavior. For example, Fig. 6b shows a decrease of the state space when using BTFN+BTB over AN policies (medium gray bars) up to 35% (for `bsort100-O1`), with an average around 15%. A better prediction policy decreases the number of control hazards and thus the number of configurations of the lower stages of the pipeline, thus reducing the size of the state space. In addition, we note that the WCET bound and the size of the state space do not always change in the same direction. For instance, in the case of `fdct-O2`, using BTFN+BTB over BTFN (light gray bar) increases the WCET bound (bar above 100%) but decreases the size of the state space (bar below 100%). Further experiments should be performed to collect lower level events (eg. cache accesses, memory accesses, flushes of the prefetch buffer, etc.) to better understand this type of phenomenon.

7 Conclusion

In this paper we show that model checking can be used to analyze the complex interactions between the components of a microarchitecture used in safety critical embedded control systems. We focus on the interaction between the instruction cache, the branch prediction unit, and a pipeline with an instruction buffer. Model checking provides a solution to perform an integrated analysis of the whole system. This integrated analysis allows to explore only feasible traces of the system and to compute the actual sequence of memory access requests and pipeline stall states corresponding to each trace. Our result are promising concerning the scalability of the approach for such systems.

In future works, we shall extend our analysis framework to support programs that use the stack to store data that impact the control flow. We also want to produce results using more complex benchmarks, and explore the impact of non-determinism concerning the initial state of the micro-architecture (eg. cache and BTB state). We will also tend toward having a model aligned with the actual e200z4 core (*i.e.* adding a second way to the pipeline) in order to validate our model against a real system through microbenchmarks. Our long term objective is to model and analyze a multiprocessor architecture based on e200z4 core such as the MPC5643L.

References

1. Bate, I., Reutemann, R.D.: Worst-case execution time analysis for dynamic branch predictors. In: 16th Euromicro Conference on Real-Time Systems, ECRTS, pp. 215–222 (2004)
2. Cassez, F., Béchennec, J.: Timing analysis of binary programs with UPPAAL. In: 13th International Conference on Application of Concurrency to System Design, ACSD, pp. 41–50 (2013)

3. Colin, A., Puaut, I.: Worst case execution time analysis for a processor with branch prediction. Real-Time Syst. **18**(2/3), 249–274 (2000)
4. Dalsgaard, A.E., Olesen, M.C., Toft, M., Hansen, R.R., Larsen, K.G.: METAMOC: modular execution time analysis using model checking. In: 10th International Workshop on Worst-Case Execution Time Analysis, WCET, pp. 113–123 (2010)
5. Engblom, J.: Analysis of the execution time unpredictability caused by dynamic branch prediction. In: 9th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, pp. 152–159 (2003)
6. Freescale semiconductors/NXP: e200z4 Power Architecture™ Core Reference Manual, rev. 0 edn., October 2009
7. Freescale semiconductors/NXP: MPC5643L Microcontroller Reference Manual, rev, 10 edn., June 2013
8. Grund, D., Reineke, J., Gebhard, G.: Branch target buffers: WCET analysis framework and timing predictability. J. Syst. Architect. **57**(6), 625–637 (2011)
9. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks - past, present and future. In: International Workshop on Worst-Case Execution Time Analysis (WCET) (2010)
10. Gustavsson, A., Ermedahl, A., Lisper, B., Pettersson, P.: Towards WCET analysis of multicore architectures using UPPAAL. In: 10th International Workshop on Worst-Case Execution Time Analysis, WCET, pp. 101–112 (2010)
11. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT **1**(1–2), 134–152 (1997)
12. Maiza, C., Rochange, C.: A framework for the timing analysis of dynamic branch predictors. In: 19th International Conference on Real-Time and Network Systems, RTNS, pp. 65–74 (2011)
13. Mangean, A., Béchennec, J.L., Briday, M., Faucou, S.: BEST: a binary executable slicing tool. In: 16th International Workshop on Worst-Case Execution Time Analysis, WCET, pp. 7:1–7:10 (2016)
14. Metzner, A.: Why model checking can improve WCET analysis. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 334–347. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27813-9_26
15. Puffitsch, W.: Efficient worst-case execution time analysis of dynamic branch prediction. In: 28th Euromicro Conference on Real-Time Systems, ECRTS, pp. 152–162 (2016)
16. Wilhelm, R.: Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 309–322. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24622-0_25
17. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. **7**(3), 36:1–36:53 (2008)
18. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. CAD Integr. Circuits Syst. **28**(7), 966–978 (2009)