Branch Target Buffers: WCET Analysis Framework and Timing Predictability

Daniel Grund[a], Jan Reineke[a], Gernot Gebhard[b]

[a]*Saarland University, Campus E1 3, D-66123 Saarbrücken, Germany*
[b]*AbsInt GmbH, Science Park 1, D-66123 Saarbrücken, Germany*

Abstract

One step in the verification of hard real-time systems is to determine upper bounds on the worst-case execution times (WCET) of tasks. To obtain tight bounds, a WCET analysis has to consider micro-architectural features like caches, branch prediction, and branch target buffers (BTB).

We propose a modular WCET analysis framework for branch target buffers, which allows for easy adaptability to different BTBs. As an example, we investigate the Motorola PowerPC 56x family (MPC56x), which is used in automotive and avionic systems. On a set of avionic and compiler benchmarks, our analysis improves WCET bounds on average by 17% over no BTB analysis.

Capitalizing on the modularity of our framework, we explore alternative hardware designs. We propose more predictable designs, which improve obtainable WCET bounds by up to 20%, reduce analysis time considerably, and simplify the analysis. We generalize our findings and give advice concerning hardware used in real-time systems.

Key words: Worst-case Execution-time (WCET) Analysis, Predictability, Branch Target Buffer (BTB)

1. Introduction

Safety critical embedded systems as found in application domains like aeronautics, automotive, and industrial automation often have to satisfy hard real-time constraints. The systems must react functionally correct and in a timely manner. To verify the latter, one needs to determine upper bounds on the WCETs of all tasks of the system [1]. The execution time of a task depends on its inputs and the initial hardware state of the system. Due to the huge amount of cases, exhaustive testing to obtain the exact WCET is infeasible. Instead, approximative but sound methods have to be applied. To be sound, such methods statically over-approximate all dynamic behavior of a task on all inputs and all initial hardware states.

In today's systems, caches, deep pipelines, BTBs, and all sorts of speculation are used to increase average-case performance. These features are challenging for timing analysis since they cause a large variability in the execution times of instructions. If an analysis cannot safely exclude spurious detrimental behavior (cache misses, pipeline stalls, etc.) the obtained WCET

Email address: grund@cs.uni-saarland.de (Daniel Grund)

bounds may become imprecise and thus useless. It depends on the design of the hardware components how well analyses can exclude such behavior.

BTBs cache addresses of branch targets or instructions at branch targets to reduce the latency when processing branches. Branches occur relatively often and the difference in latency between a BTB hit and a miss is large enough to have a significant influence on the execution time. Thus, a BTB analysis is necessary to obtain precise WCET bounds.

Our first contribution is presented in Section 5. We introduce a modular WCET analysis framework for BTBs that can be adapted to various BTB implementations. It consists of a fixed main module that is the same for all BTBs and two parameter modules each of which answers one of the following questions: For which branches does the BTB contain information? What information is stored in the BTB for a given branch? The modules interact via fixed interfaces such that they can be exchanged independently.

Our second contribution, in Section 6, is an instantiation of our framework for the MPC56x. The MPC56x is used in time-critical automotive and avionics systems and features a branch processing unit (BPU) with branch prediction and a BTB. This case study shows the applicability of our framework for a non-trivial case and demonstrates the effort needed to model a hardware feature. This instantiation improves the WCET bounds by on average 17% on a set of avionic and compiler benchmarks. On a subset of the benchmarks measuring execution time was possible, which yields under-approximations of the WCET but allows to bound the overestimation of our analysis. For this subset, our analysis reduced the average overestimation from 50% to 17%.

Our last contribution is the identification of principles of more predictable hardware designs and their influence on WCET bounds. In Section 8, we identify problems regarding predictability of the example BTB we study. Capitalizing on the modularity of our analysis, we propose alternative hardware designs and evaluate them by additional experiments. In case of the MPC56x, employing a more predictable replacement policy (LRU) in the BTB would improve the computed WCET bounds by 2.9% and reduce analysis time considerably. Minor modifications that increase uniformity by eliminating special cases would not only simplify analysis but also improve the WCET bounds obtained with our analysis by up to 20%. Finally, we generalize our findings and give advice to hardware designers.
Our main contributions are:

- A generic analysis framework for BTBs.
- An instance of the framework for the Motorola PowerPC 56x CPU family, which includes the first non-trivial analysis of FIFO replacement.
- Identification of sources of unpredictability in architectures, motivated at the example of BTBs.

In the next section we repeat basic notions of control-flow prediction, static analysis, and abstract interpretation. To illustrate the analysis problem at hand, Section 3 describes the BTB of the MPC56x, which we also use in our evaluation. Section 4 describes the context of this work, i.e., the overall architecture of the WCET analysis tool which we extended. Section 5 presents exactly that extension: The generic analysis framework for BTBs. As a concrete example, we come back to the MPC56x explained in Section 3 and instantiate our framework for this CPU family. Section 7 describes how and to which extent we validated our analysis and contains evaluation results for our analysis. The predictability of BTBs and additional measurements for alternative hardware designs are discussed in Section 8. Finally, related work is discussed in Section 9 and Section 10 concludes.

2. Foundations

2.1. Static Analysis

Static analysis automatically determines properties of programs without actually executing the programs. Since the properties to determine are commonly incomputable, abstraction has to be employed. In general, there is a trade-off between analysis precision and analysis complexity.

One formal method in static analysis, which our work is based on, is abstract interpretation. Instead of representing concrete semantic information in a concrete domain D, one represents more abstract information in an abstract domain $\widehat{D}$. The relation between concrete and abstract can be given by an abstraction function $\alpha : \mathcal{P}(D) \to \widehat{D}$ and a concretization function $\gamma : \widehat{D} \to \mathcal{P}(D)$.

To determine the properties, a data-flow analysis computes invariants for each program point, which are represented by values of $\widehat{D}$. A *transfer function* $\mathcal{U} : \widehat{D} \times I \to \widehat{D}$ models the effect of instructions I on abstract values. With the transfer function it is possible to set up a system of data-flow equations that correlates values before and after each instruction. If an instruction has multiple predecessors, a *join function* $\mathcal{J} : \widehat{D} \times \widehat{D} \to \widehat{D}$ combines all incoming values into a single one. If a data-flow framework meets certain conditions, the induced system of equations for a given program has a least solution, which can be obtained by a fixed-point computation. If the transfer- and the join-function satisfy certain conditions, the analysis is sound with respect to α and γ: True properties in the abstract map to true properties in the concrete. For an introductory article on abstract interpretation confer Cousot and Cousot [2].

2.2. Control Flow and Prediction Concepts

In general, prediction mechanisms in CPUs enable execution of code that depends on yet unknown facts. The prediction schemes relevant to our work try to mask latencies associated with branches. If the reductions in latency outweigh the penalties of occurring mispredictions, the performance of the system is increased.

Branch prediction. Branch prediction tries to predict whether a conditional branch will be taken or not. This enables further fetching of instructions as soon as the potential branch target has been decoded. There are several static and dynamic branch prediction mechanisms. In static branch prediction, the prediction (predicted taken or predicted not taken) is encoded in the branch instruction and can be easily determined statically. In dynamic branch prediction, the prediction depends on the history of branches that were executed before.

Branch target prediction. Branch target prediction tries to predict the target of a branch given only the address of the branch instruction. The mechanism employed to implement this feature is called *branch target buffer* (BTB). To disambiguate it from other variants of BTBs, we will refer to this kind of BTB as *addr-BTB*. It maps the addresses of branch instructions to the addresses of their respective branch targets. This can be used to speculatively start fetching the instruction at a branch target before even decoding the branch instruction. In contrast to branch prediction this is useful not only for conditional branches, but for *all* so-called *change-of-flow* (COF) instructions, e.g., (un)conditional jumps, calls, etc.

There are other variants of BTBs usually called *branch target instruction caches* (BTIC). Instead of storing branch target *addresses*, BTICs store *instructions* found at the branch targets. One variant, which we will call *src-BTIC*, maps the address of a branch instruction to the instruction at the branch target and the instructions following it. Due to the low latency of the BTIC,

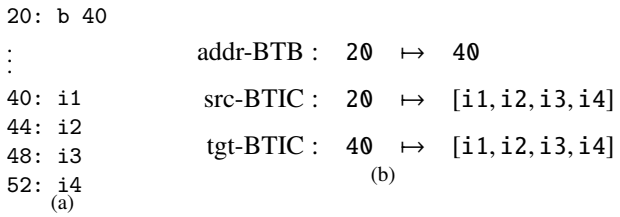


Figure 1: Three different types of BTBs. (a) Example with a single branch instruction and instructions `i1,...,i4` at the branch target. (b) Information stored for this branch in the three different types of BTBs.

retrieving the instructions from the BTIC saves time compared to fetching from memory. A last variant, which we will call *tgt-BTIC*, maps the address of a branch target to the instruction at the branch target and the instructions following it.

To achieve a low latency, BTBs are on-chip. This limits their size. Therefore, BTBs cannot store information for all branches. Similarly to caches, a replacement policy has to decide which information to discard when the BTB is full.

In the following we will use the term BTB for all of these three variants and the term BTIC for the last two variants. See Figure 1 for an illustration of the differences between the three types of BTBs. For details on BTBs confer [3].

3. Example: The MPC56x BTIC

In this section we explain the BTIC of the MPC56x. After describing our framework in Section 5, we will come back to it in our case study.

The MPC56x features the following branch instructions: Branch `b`, conditional branch `bc`, and register indirect branches `blr`, `bctr`, with conditional versions `bclr`, `bcctr`. The BPU of the MPC56x employs *programmable static branch prediction*. Unconditional branches and backward conditional branches are predicted taken; all others are predicted not-taken. For conditional branches this behavior can be reversed by changing a bit in the opcode. Prediction only takes place if the branch condition is still unevaluated and the target address is already known. For register indirect branches the target address is known if no other instruction to be executed before the branch may write to the respective register.

The MPC56x features a *fully-associative 8-way 4-entry* FIFO *tgt-BTIC*. It can store information for up to 8 branches in so-called *lines*. Up to 4 subsequent instructions at a branch target may be stored in each line. The BTIC is fully-associative, which means that information for a branch may be stored in any of the 8 lines. If the BTIC is full, lines are replaced in a first-in first-out (FIFO) manner.

On a change of flow (COF), the BTIC is queried. If the BTIC contains information for this branch (i.e., the fetch request is a BTIC hit), the subsequent cached instructions are fetched out of the BTIC with a latency of 1 cycle each. If the BTIC does not contain information for this branch (a BTIC miss), the line whose contents are to be replaced next is freshly allocated. This line is then filled with the instructions that will be fetched next, e.g., from external memory with a latency between 2 and 60 cycles each (depending on the memory type). This filling process will be detailed later. The number of cached instructions in a valid line may vary from 2 to 4. Caching starts with the instruction at the branch target and ends after 4 instructions or when another COF

```
 0: b 4            64:
 4:                68: b 80
 8:
12:                80:
16:                84:
20: b 40           88:
                   92:
40:                96: divw r3, r6, r7
44:               100: cmp cr0, r3, r4
48:               104: bc 88
52: b 64          108:
```

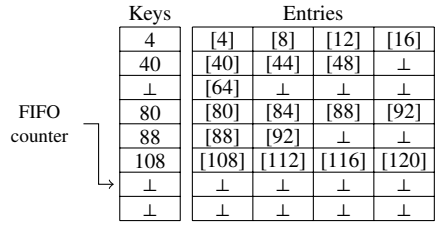


Figure 2: Example program and state of the initially empty BTIC after execution of the program. `b` denotes a branch instruction, `bc` a conditional branch instruction. [a] denotes the instruction at address a. Each instruction is 4 bytes long.

occurs. If only one instruction would be cached, the whole line is marked as invalid. In this case the FIFO counter is incremented nonetheless, i.e., the line is not immediately reused upon the next BTIC miss.

Figure 2 show a program and its effect on an initially empty BTIC. After the first branch the BTIC line is filled with 4 entries. Branching from 20 to 40 only fills the line with 3 instruction because another COF happens at 52. The branch to 64 is directly followed by another branch to 80. Due to this immediate second branch, the line allocated for 64 will be invalidated. The conditional branch at 104 is predicted taken, but assume that it actually is not taken. First, fetching will start at the predicted branch target 88 until the branch condition computed at 100 is evaluated. When the misprediction becomes evident another COF to 108 fills the next line. This is an example of a COF due to branch mispredictions.

Other BTBs. BTBs are found in various embedded processors, e.g., all modern PowerPCs, the MC68060, etc. For instance, the e200z6 PowerPC stores the target addresses of up to 8 branch instructions in a BTB using FIFO replacement. Like the processors of MPC500 family, the e200z6 is normally used in the automotive domain. Common applications are, e.g., fuel injection control or electronically controlled transmissions.

A variation of a BTB is found in the PowerPC 750 family, which feature a 4-way set-associative BTIC with 16 sets that stores up to two instructions per entry. Additionally, a 512-entry branch history table is used to steer dynamic branch prediction. Processors of this family are employed in avionic applications, such as flight control or turbine control.

BTBs are also employed in various desktop CPUs, e.g., the Cyrix 6x86 or the Intel Pentium processor, both using a 256-entry, four-way set associative BTB to store branch target addresses and branch prediction information.

4. WCET Analysis & Tool Architecture

Over the last several years, a more or less standard architecture for timing-analysis tools has emerged. Figure 3 gives a general view on this architecture. First, one can distinguish three major building blocks:

- Control-flow reconstruction and static analyses for control and data flow.

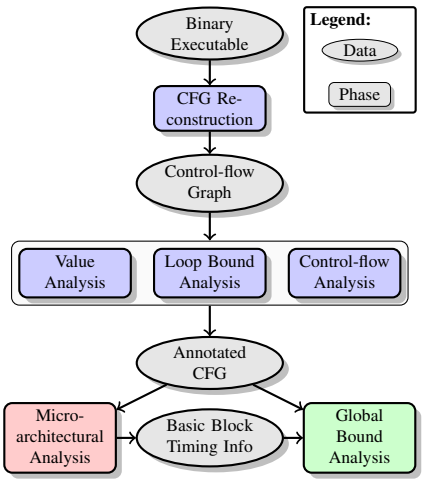


Figure 3: Main components of a timing-analysis framework and their interaction.

- Micro-architectural analysis, which computes upper and lower bounds on execution times of basic blocks.
- Global bounds analysis, which computes upper and lower bounds for the whole program.

The following list presents the individual phases and describes their objectives and problems. Note that the first four phases are part of the first building block.

1. *Control-flow reconstruction* [4] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. WCET analysis on a higher-level program representation is infeasible since different legal translations to binary code may produce very different timing behavior. To perform the analysis on source code, one would need to account for the compiler, which complicates engineering and certification of the tool chain. Additionally, there are advantages in starting with binaries; e.g., all instruction addresses are fixed and known.
The main objective is to construct a control-flow graph of the program and to decode the instructions. Challenges encountered are dynamically computed control-flow successors, e.g., stemming from switch statements, function pointers, etc..
2. *Value analysis* computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis [5, 6]. This information is among others used for a precise data-cache analysis.
3. *Loop bound analysis* [7, 8] identifies loops in the program and tries to determine bounds on the number of loop iterations, information indispensable to bound the execution time. Challenges are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.
4. *Control-flow analysis* [7, 9] narrows down the set of possible paths through the program by eliminating infeasible paths or to determine correlations between the number of executions of different blocks using the results of value analysis results. These constraints will tighten the obtained timing bounds.

5. *Micro-architectural analysis* [10, 11, 12] determines bounds on the execution time of basic blocks. As the analysis presented in Section 5 is part of this phase, we describe it in a little more detail. The micro-architectural analysis accounts for all components of the processor that affect execution time. This includes the pipeline, caches, and speculation concepts, i.e., everything "below" the instruction-level architecture (ISA). Such average-case-enhancing features need to be modeled to a certain extent. Otherwise, the bounds on execution times are likely to be too imprecise and thus of little use. Furthermore, simply ignoring a feature, i.e., not modeling it at all, is generally unsound. Due to interactions between different features of a processor, it is often not sound to apply greedy strategies in the micro-architectural analysis: While a miss in the BTB initially takes more time than a hit, interactions with other components of a processor, like speculative execution, may cause the BTB hit case to take more time overall. Such phenomena are known as *timing anomalies* [13, 14] and have been shown to exist in abstractions of many modern architectures. As it is difficult to exclude the existence of timing anomalies, micro-architectural analyses generally have to consider all cases, i.e., in the BTB example, they have to consider both the BTB miss and the BTB hit case. As a consequence of modeling all these micro-architectural features and not being able to apply greedy strategies, this analysis phase is the computationally most expensive one.
The whole micro-architectural analysis can be performed as a single abstract interpretation. Like actual hardware is made up of components, the abstract domain that is used in this abstract interpretation is made up of abstract domains for individual hardware components: There are abstract (sub-)domains for parts of the pipeline, buffers, caches, buses, peripheral devices etc. The abstract interpretation models the evolution of abstract architectural state at the granularity of CPU cycles. Thereby, to update the abstract architectural state of a component one might need information about the state of other components. Hence, the abstract sub-domains have interdependencies much like in the concrete hardware. By connecting the cycle-level semantics of the CPU to the instruction-level semantics (explained for instance in [11] or [15]), one can attach information to program points rather than to CPU cycles.
Generally, the analysis of a micro-architectural feature, e.g., the cache analysis, is *not* performed in isolation but simultaneously within the single abstract interpretation. Hence, *technically speaking*, the results of a cache analysis are the invariants that are established by the abstract cache sub-domain of the abstract interpretation. The precision of these invariants depends not only on the precision of the abstract cache sub-domain, but also on the precision of other abstract domains it is interacting with.

6. *Global bounds analysis* [16, 17] finally determines bounds on execution time for the whole program. Information about the execution time of basic blocks is combined to compute shortest and longest paths through the program. This phase takes into account information provided by the loop bound- and control-flow analysis.

The commercially available tool `aiT` by AbsInt, confer `http://www.absint.de/wcet.htm`, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [12, 18, 19, 20].

4.1. Integration of the BTB Analysis Into the Framework

Accounting for branch target buffers in such an analysis framework makes it necessary to:

- design abstract domains for BTBs
- integrate these domains into the micro-architectural analysis

The first point boils down to the definition of a representation of BTB states and the definition of update- and join-functions used by the abstract interpreter. As there are many kinds of BTBs that behave identical or similar in part, those abstract domains should be adaptable and extensible.

The second point is straightforward. As explained above, the micro-architectural analysis is a single abstract interpretation. Hence, the first step is to integrate the BTB domain as a sub-domain into the domain of the abstract interpretation. The second step deals with the interfacing between the different abstract domains. In case of our BTB domain, this is rather simple: To update an abstract BTB state, the update function (of the abstract BTB domain) must know whether a branch was taken or not. In the way we model it, it is sufficient to know the current and the previous value of the instruction pointer. In the update functions described later, this will manifest itself in the form of "fetch" and "fetched" parameters.

5. WCET Analysis Framework of BTBs

In this section we describe our main contribution, a framework for the WCET analysis of BTBs. Like all timing analyses, a BTB timing-analysis reduces WCET bounds by proving that certain timing-detrimental system behavior is impossible. That is, the BTB analysis statically proves that certain BTB accesses cannot be misses in any program run; it identifies BTB accesses that will always be hits. Whether an access is a hit or a miss depends on the state of the BTB, which in turn depends on the history of accesses that were performed on it. At hand of the binary code, the static analysis gains information about the BTB state by considering BTB accesses as they will occur during program execution. The result of the analysis are invariants about the BTB state for different points in time (program locations). Those invariants allow to classify BTB accesses.

The framework defines the main analysis, which analyzes behavior that is common to BTBs. Additionally, the main analysis takes two parameters that depend on the particular BTB to be analyzed: a *key* and a *content analysis*. BTBs can be seen as associative maps, where the keys are instruction addresses and the values are the associated BTB contents. Which keys to store information for, and what information to store for a particular key, differs from one BTB to another. The instantiations of the two analyses answer these two questions for a particular BTB.

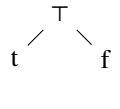
A BTB changes its state depending on the address "*fetch*" for which an instruction fetch is issued and the last address "*fetched*" for which a request was serviced. Likewise, the update of the abstract BTB state depends on these two addresses, which have to be provided by other analyses (i.e., value analysis). The domain of both *fetch* and *fetched* is $\mathrm{A}_\perp = \mathrm{A} \cup \{\perp\}$, where A is the set of addresses and $\perp$ denotes that no instruction fetch is issued or serviced. Changes in control-flow trigger BTB reactions. Derived from *fetch* and *fetched*, we define the *change-of-flow* shortcut *cof* := $\mathit{fetched} \neq \perp \wedge \mathit{fetch} \neq \mathrm{Succ}(\mathit{fetched})$, where Succ(*fetched*) is the address immediately following *fetch*.

5.1. Generic BTB analysis

The domain of the analysis is

$$\mathrm{BTB} := \mathrm{Inh} \times \mathrm{PLB} \times \mathrm{Keys} \times \mathrm{Cont},$$

where the first two components, Inh and PLB, model behavior common to all BTBs and will be explained in the following paragraphs. Keys and Cont are the domains of the *key* and the *content* analyses, which are parameters in the framework and have to be adapted to each specific BTB.



⊔	f	t	⊤
f	f	⊤	⊤
t	⊤	t	⊤
⊤	⊤	⊤	⊤

∧	f	t	⊤
f	f	f	f
t	f	t	⊤
⊤	f	⊤	⊤

Figure 4: The $\mathbb{B}^\top$ semi-lattice and its induced join $\sqcup$ and logical and $\wedge$.

First, there is the concept of *inhibited regions*. It allows to define code regions for which no information should be stored in a BTB. It is useful in case of self-modifying code to prevent the use of outdated BTB contents.

$$\text{Inh} := \mathbb{B}^\top,$$

see Figure 4, expresses whether the fetched address is inside such a region. $\top$ subsumes both possibilities. If the address that is currently fetched is inside an inhibited region, caching is inhibited. Caching is resumed if an instruction outside of all inhibited regions is fetched.

$$\mathcal{U}_{\text{Inh}}(inh, fetch) := \begin{cases} inh & : fetch = \bot \\ isInhibited(fetch) & : \text{else} \end{cases}$$

Second, BTICs may store more than one instruction at a branch target. To find a requested instruction, a tgt-BTIC (src-BTIC) uses the address of the last branch target (branch instruction). Hence, the analysis needs to know this address to classify a request as hit or miss. For a more uniform presentation we define the shortcut

$$key := \begin{cases} fetch & : \text{tgt-BTIC} \\ fetched & : \text{src-BTIC or addr-BTB} \end{cases}$$

Because of control-flow joins the analysis may need to account for several *potential last branches*

$$\text{PLB} := \mathcal{P}(\text{A}).$$

On a change of flow, the set of potential last branches is set to the current instruction; otherwise it is left unchanged. Note that the size of the set may grow through joins.

$$\mathcal{U}_{\text{PLB}}(plb, fetch, fetched) := \begin{cases} \{key\} & : cof \\ plb & : \text{else} \end{cases}$$

The *update function* takes the addresses *fetch* and *fetched* as arguments and is defined as follows:

$$\mathcal{U}_{\text{BTB}} : \text{BTB} \times \text{A} \times \text{A} \to \text{BTB}$$
$$\mathcal{U}_{\text{BTB}}((inh, plb, keys, cont), fetch, fetched) := (inh', plb', keys', cont')$$

where

$$
\begin{aligned}
inh' :=& \mathcal{U}_{\text{Inh}}(inh, fetch) \\
plb' :=& \mathcal{U}_{\text{PLB}}(plb, fetch, fetched) \\
keys' :=& \begin{cases} \mathcal{U}_{\text{Keys}}(keys, key) & : cof \wedge (inh = \text{f}) \\ \mathcal{J}_{\text{Keys}}(keys, \mathcal{U}_{\text{Keys}}(keys, key)) & : cof \wedge (inh = \top) \\ keys & : \text{else} \end{cases} \\
cont' :=& \mathcal{U}_{\text{Cont}}(cont, fetch, fetched, inh, keys)
\end{aligned}
$$

Note that $keys'$ and $cont'$ depend on the update functions defined by the parameter analyses. On a COF, key information is updated if caching is *not* inhibited. In case the status of inhibited is unclear ($\top$), the analysis must conservatively account for both cases; hence the join of the old state with the potentially new state.

The *join function* is straightforward and partly depends on the parameter analyses, too:

$$
\begin{gathered}
\mathcal{J}_{\text{BTB}} : \text{BTB} \times \text{BTB} \to \text{BTB} \\
\mathcal{J}_{\text{BTB}}(btb_1, btb_2) := (inh_1 \sqcup inh_2, plb_1 \cup plb_2, \mathcal{J}_{\text{Keys}}(keys_1, keys_2), \mathcal{J}_{\text{Cont}}(cont_1, cont_2))
\end{gathered}
$$

The join $\sqcup$ of the two inhibited status is shown in Figure 4, and the potential last branches are joined by set union $\cup$.

5.2. Parameter analyses

The *key analysis* determines for which addresses information is or is not contained in a BTB. Depending on the type of BTB, the addresses can be either the address of a branch instruction or the address of a branch target. Additional degrees of freedom in this cache-like structure are the size, number of sets, associativity, and replacement policy. Implementations of the module with abstract domain Keys must implement the following interface:

$$
\begin{aligned}
\mathcal{U}_{\text{Keys}} &: \text{Keys} \times \text{A} \to \text{Keys} && \text{(update function)} \\
\mathcal{J}_{\text{Keys}} &: \text{Keys} \times \text{Keys} \to \text{Keys} && \text{(join function)} \\
C_{\text{Keys}} &: \text{Keys} \times \text{A} \to \mathbb{B}^{\top} && \text{(classification function)}
\end{aligned}
$$

The update function shall take as argument an address, i.e., a BTB key, for which BTB content is queried. It shall conservatively model the update process of BTB keys, including possible replacements. The classification function shall determine for a given address if the BTB contains information for this key. If the classification is t (f) the key must (must not) be contained in the BTB. If it is $\top$ both cases might be possible, see Figure 4. Altogether, a key analysis is similar to a cache analysis.

The *content analysis* determines which information is contained in a BTB for a given key. This may be the address of the branch target (addr-BTB) or one or more instructions at the branch target (BTIC). Additional degrees of freedom are the number of stored instructions, the way entries are filled, and validity information. The interface for this module with abstract domain Cont is:

$$
\begin{aligned}
\mathcal{U}_{\text{Cont}} &: \text{Cont} \times \text{A} \times \text{A} \times \text{Inh} \times \text{Keys} \to \text{Cont} \\
\mathcal{J}_{\text{Cont}} &: \text{Cont} \times \text{Cont} \to \text{Cont} \\
C_{\text{Cont}} &: \text{Cont} \times \text{A} \times \text{A} \to \mathbb{B}^{\top}
\end{aligned}
$$

The update function shall take as parameters the address of the instruction currently being fetched, the address of the last fetched instruction, inhibition- and keys-information. It shall conservatively model the update process of BTB contents for an instruction fetch. The classification function takes a key *key* (e.g., the address of a branch instruction in case of an addr-BTB or a src-BTIC) and another address a as parameters. If the BTB associates information with *key* ($C_{\text{Keys}}(keys, key) = \text{t}$), then $C_{\text{Cont}}(cont, key, a)$ should tell whether the BTB does associate a with *key* or not. In case of a BTIC, it should tell whether *key* is associated with the instruction at address a. Depending on the type of BTB *key* is either the address of a branch instruction (addr-BTB and src-BTIC) or of a branch target (tgt-BTIC).

5.3. Classification

Given the classification functions of the parameter analyses, we can now define the *classification function* of the generic BTB analysis. For an addr-BTB it is:

$$C_{\text{BTB}} : \text{BTB} \times \text{A} \to \mathbb{B}^{\top} \times \mathcal{P}(\text{A})$$
$$C_{\text{BTB}}((inh, plb, keys, cont), a) := \left(C_{\text{Keys}}(keys, a), \begin{cases} \{tgt \mid C_{\text{Cont}}(a, tgt) \neq \text{f}\} & : C_{\text{Keys}}(keys, a) \neq \text{f} \\ \emptyset & : \text{else} \end{cases}\right)$$

Given the address a of a branch instruction, it classifies whether information for this branch is contained in the BTB (key analysis) and which branch target addresses may be associated with a (content analysis).

The *classification function* for both variants of BTICs is:

$$C_{\text{BTB}} : \text{BTB} \times \text{A} \to \mathbb{B}^{\top}$$
$$C_{\text{BTB}}((inh, plb, keys, cont), a) := \bigsqcup_{lb \in plb} \left\{C_{\text{Keys}}(keys, lb) \wedge C_{\text{Cont}}(cont, lb, a)\right\}$$

Given the address a of the currently fetched instruction, it classifies whether this fetch can be serviced by the BTIC. In a concrete execution, a BTIC can service a fetch if the BTIC contains information for the last taken branch (key analysis) and if this information contains a (content analysis). Due to joins there may be a set of possible last branches (*plb*). Hence, the analysis has to account for all possibilities and combine ($\sqcup$) the respective classifications.

6. Case Study: Analysis of the MPC56x BTIC

In this section we demonstrate the applicability of our framework. We instantiate it for the BTIC of the MPC56x by defining the two parameter analyses for it. The model of the MPC56x described in Section 3 and below was distilled from manuals [21, 22, 23] and from measurements (see Section 7) that allow to precisely infer its functioning.

6.1. The Key Analysis

In static cache analysis by abstract interpretation, there is a concept of *may*- and *must*-cache information at program points: may- and must-caches are upper and lower approximations, respectively, to the contents of all concrete caches that will occur whenever program execution

reaches a program point. The must-cache at a program point is a set of elements that are definitely in each concrete cache at that point. Analogously, the may-cache is a set of elements that may be in a concrete cache at a program point.

Must-cache information is used to derive safe information about cache hits. May-cache information is used to safely predict cache misses. Predicting misses can be important to obtain more precise must information, so that more hits can be predicted. For details on (LRU) cache analysis see Ferdinand et al. [24, 12].

The employed cache replacement policy has a great influence on (the design of) a cache analysis. The MPC56x employs FIFO replacement. Conceptually, a FIFO buffer maintains a fixed-size queue of elements that are ordered from first-in to last-in. A concrete FIFO cache set s can therefore be modeled as a k-tuple of cache tags: $s = [t_0, \ldots, t_{k-1}] \in \mathrm{S} := \mathrm{T}^k$, where T is the set of tags.

A cache hit does not change the cache set state. A cache miss appends the new tag, shifting the others to the left and evicting the one at the first-in position 0. $\mathcal{U}_\mathrm{S}(s,t)$ computes the effect of accessing tag t on the cache set s:

$$\mathcal{U}_\mathrm{S} : \mathrm{S} \times \mathrm{T} \to \mathrm{S}$$
$$\mathcal{U}_\mathrm{S}([t_0, \ldots, t_{k-1}], t) := \begin{cases} [t_0, \ldots, t_{k-1}] & : \exists i : t = t_i \\ [t_1, \ldots, t_{k-1}, t] & : \text{else} \end{cases}$$

Up to date, there is no published FIFO may analysis and the best FIFO must analysis can in each case only predict the last accessed tag as a hit. Based on the concept of *relative competitiveness* [25], we present the first FIFO may analysis. Furthermore, we propose a new FIFO must analysis that is able to exploit may information.

May Analysis. *Relative competitiveness* is a comparative concept. Relative competitive ratios bounds the performance of one replacement policy relative to the performance of another replacement policy, e.g., 8-way-LRU vs. 4-way-FIFO. This allows to transfer guarantees on the number of misses (hits) obtained for one policy to another policy. The relative competitiveness results of LRU vs. FIFO allow for the following corollary: A may analysis for a $2k-1$-way LRU cache set is also a may analysis for a k-way FIFO cache set.

This result allows us to use any LRU may analysis in a black box manner. The abstract domain for our FIFO may analysis is

$$\mathrm{May}_{\mathrm{FIFO}(8)} := \mathrm{May}_{\mathrm{LRU}(15)},$$

short May. It is based on the abstract domain $\mathrm{May}_{\mathrm{LRU}(k)} := \mathcal{P}(\mathrm{T})^k$ given in [12]:

$$\mathrm{May}_{\mathrm{LRU}(k)} := \mathcal{P}(\mathrm{T})^k$$
$$\mathcal{U}_{\mathrm{May}_{\mathrm{LRU}(k)}} : \mathrm{May}_{\mathrm{LRU}(k)} \times \mathrm{T} \to \mathrm{May}_{\mathrm{LRU}(k)}$$
$$\mathcal{J}_{\mathrm{May}_{\mathrm{LRU}(k)}} : \mathrm{May}_{\mathrm{LRU}(k)} \times \mathrm{May}_{\mathrm{LRU}(k)} \to \mathrm{May}_{\mathrm{LRU}(k)}$$

For $[T_0, \ldots, T_{k-1}] \in \mathrm{May}_{\mathrm{LRU}(k)}$, the analysis maintains the invariant that the disjoint union $\dot{\bigcup}_i T_i$ is an over-approximation of the cache contents.

Must Analysis. Given a concrete cache set state, it is easy to determine the exact number of misses needed to evict a cached tag. To be able to guarantee hits, our abstract cache-set states allow to safely approximate this number from below. We define the abstract domain $\text{Must}_{\text{FIFO}(k)} := \mathcal{P}(\text{T})^k$, short Must. The position of a cached tag in a k-tuple is a lower bound on the number of misses needed to evict it from the cache set. Power sets are necessary because multiple tags may have the same lower bound. Since it is senseless to specify multiple lower bounds for one tag, all k sets are defined to be disjoint. Furthermore, the size of all sets together is bounded by k because no more than k tags can be in any concrete cache set. The meaning of an abstract cache can be illustrated with the concretization function:

$$\gamma_{\text{Must}} : \text{Must} \to \mathcal{P}(\text{S})$$
$$\gamma_{\text{Must}}([T_0, \ldots, T_{k-1}]) := \{[t_0, \ldots, t_{k-1}] \in \text{S} \mid \forall i \forall t \in T_i \exists j \geq i : t_j = t\}$$

For example, consider $s_1 := [\{b\}, \{a, c\}, \{\}, \{f\}]$ and $s_2 := [\{a\}, \{b, c\}, \{d\}, \{\}]$. Their concretizations are $\gamma_{\text{Must}}(s_1) = \{[b, a, c, f], [b, c, a, f]\}$ and $\gamma_{\text{Must}}(s_2) = \{[a, b, d, c], [a, c, d, b], [a, b, c, d], [a, c, b, d]\}$.

The abstract update function has three cases. If the analysis can predict a hit, the must information remains unchanged as FIFO does not change its state upon a hit. If the tag t is not in the may cache the analysis can predict a miss and update the must information similarly to the concrete semantics. If neither hit nor miss can be predicted the analysis has to account for both possibilities: Since the access might be a miss, all sets are shifted to the left. Since it might be a hit on the first-in position, the tag can only be added to the leftmost position. This results in:

$$\mathcal{U}_{\text{Must}} : \text{Must} \times \text{May} \times \text{T} \to \text{Must}$$
$$\mathcal{U}_{\text{Must}}([T_0, \ldots, T_{k-1}], [U_0, \ldots, U_{2k-2}], t) := \begin{cases} [T_0, \ldots, T_{k-1}] & : \exists i : t \in T_i \\ [T_1, \ldots, T_{k-1}, \{t\}] & : \nexists j : t \in U_j \\ [T_1 \cup \{t\}, T_2, \ldots, T_{k-1}, \emptyset] & : \text{else} \end{cases}$$

The join function has to regard the following soundness constraints. A tag may only be contained in the result if it is present in both operands. The position of such a tag must be (at most) the minimum of the two positions in the operands. The best possible join function for our domain is
$\mathcal{J}_{\text{Must}}([X_0, \ldots, X_{k-1}], [Y_0, \ldots, Y_{k-1}]) := [T_0, \ldots, T_{k-1}]$ where $T_l := \{t \in \text{T} \mid \exists i, j : t \in X_i, t \in Y_j, l = min\{i, j\}\}$. For example, consider the join of the two example states from above: $\mathcal{J}_{\text{Must}}(s_1, s_2) = [\{a, b\}, \{c\}, \{\}, \{\}]$.

If we are only interested in whether a tag is contained in the abstract cache we write $t \in must$ for $\exists i : t \in T_i$, where $must = [T_0, \ldots, T_{k-1}]$ and similarly for $t \in may$.

BTB interface implementation. For the MPC56x, the key analysis interface can be implemented by (with T := A):

$$\text{Keys} := \text{May} \times \text{Must}$$
$$C_{\text{Keys}}((may, must), a) := \begin{cases} \text{t} & : a \in must \\ \text{f} & : a \notin may \\ \top & : \text{else} \end{cases}$$

The update and join are defined component-wise on *may* and *must*.

6.2. The Content Analysis

The content analysis has to determine what information is stored in the BTIC for a particular key. In case of the MPC56x these are up to 4 instructions at a branch target. Before describing the analysis, we describe the line filling process in more detail that determines the occupancy of BTIC lines. This process was already foreshadowed in the example in Figure 2.

Line filling process. If the BTIC-logic detects a COF the BTIC is queried. If this results in a miss, the filling process starts, which is described below and illustrated in Figure 5:

1. The BTIC selects a line that will hold the instructions at the branch target. This line is selected following the FIFO policy and the FIFO counter is updated. We will call this event to *open a line*.
2. The next instructions that are fetched will be inserted into the line's entries until it is closed or invalidated. Entries are filled with instructions fetched in the previous cycle. This explains the two-step pattern ($1a/1b, 2a/2b$, etc.) in Figure 5.
3. The BTIC line is *closed* on any of the following events:
   - 4 instructions have been inserted; the line is full.
   - Another COF happens, and at least 2 entries have been filled.
4. The open line is used for a different key if another COF occurs before any entry has been filled. This is modeled by the transition between $1a$ and $open$.
5. The BTIC line is *invalidated* if another COF occurs and only one entry has been filled. This corresponds to the transition between $2a$ and $open$. Note, that in this case the FIFO counter is *not* set back to point to the invalid line.

If the BTIC query results in a hit, cached instructions are fetched out of the BTIC but the line is *not* opened again. That is, possibly missing entries in a line can *not* be completed after a hit, entries can only be filled after a miss.

Summarizing, the number of cached instructions in a valid BTIC line may vary from 2 to 4. Caching starts with the instruction at the branch target and ends after 4 instructions or when another COF occurs. If only one instruction would be cached, the whole line is marked as invalid.

Entries. The goal of the content analysis is to determine the state of the BTIC lines, i.e., which instructions are cached in the entries of a line. In the abstract domain we model an entry by

$$\text{Entry} := \mathbb{B}^{\top}.$$

t indicates that an entry is occupied, f that it is empty, and $\top$ that it might be occupied. $\top$ accounts for possible lack of knowledge in the analysis. A line can thus be modeled by: $\text{Line} := (\text{Entry}^4)_{\bot}$. $\bot$ represents no concrete line. Its use will become clear later. Entries are joined as follows:

$$\mathcal{J}_{\text{Line}}(x, y) := \begin{cases} [x_1 \sqcup y_1, \ldots, x_4 \sqcup y_4] & : x = [x_1, \ldots, x_4] \wedge y = [y_1, \ldots, y_4] \\ x & : y = \bot \\ y & : x = \bot \end{cases}$$

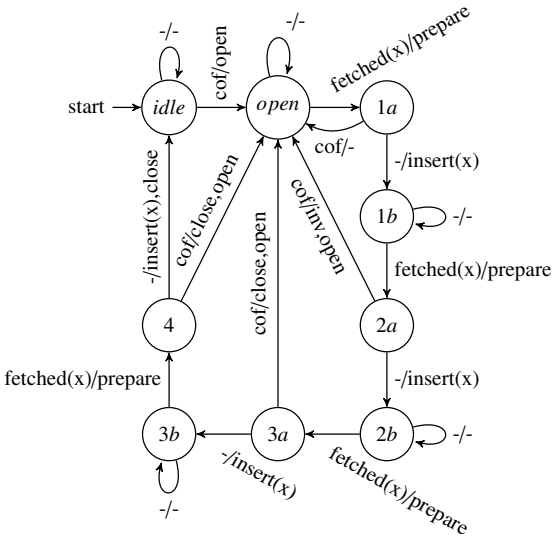


Figure 5: Automaton describing the BTIC behavior by states and event/action pairs.

Potentially open lines. Since only open lines can be filled, the analysis needs to distinguish open and closed lines. Although during execution there can only be one open line at any time, the analysis must be able to handle up to 4 possibly open lines at a program point. This is due to control-flow joins. Consider very short basic blocks that are placed consecutively in the address space. If the blocks are reachable via branches and fall-throughs all basic blocks contribute a potentially open line at the branch target of the consecutive basic blocks. Hence, we define the domain for the potentially open lines *pol* by

$$\text{Pol} := \{X \subseteq \text{A} \mid |X| \leq 4\}.$$

pol contains keys of potentially open lines, i.e., addresses of last branches *lb*. More than 4 lines are impossible because a line is closed if 4 instructions have been fetched.

To update the set of potentially open lines, the analysis needs to know about lines that were completely filled by the last fetch, and are thus being closed now:

$$\mathit{filling} := \{\mathit{lb} \in \mathit{pol} \mid \mathit{pos}(\mathit{lb}, \mathit{fetched}) = 3\}$$
$$\mathit{pos} : \text{A} \times \text{A} \to \mathbb{Z}$$
$$\mathit{pos}(\mathit{lb}, a) := \frac{1}{4}(a - \mathit{lb})$$

If no COF happens, it is these lines that are removed from the *pol* set. Upon a COF all open lines are closed or invalidated, as described in Figure 5. None of them remains opened. If caching is inhibited or the COF results in a BTIC hit, no new line is opened and the set of potentially open lines is empty. Upon a BTIC miss a new line is opened if caching is not inhibited. Hence, the

update function is:

$$\mathcal{U}_{\text{Pol}} : \text{Pol} \times \text{A} \times \text{A} \times \text{Inh} \times \text{Keys} \to \text{Pol}$$

$$\mathcal{U}_{\text{Pol}}(pol, fetch, fetched, inh, keys) := \begin{cases} pol \setminus filling & : \neg cof \\ \emptyset & : cof \ \wedge (C_{\text{Keys}}(keys, fetch) = \text{t} \vee (inh = \text{t})) \\ \{fetch\} & : cof \ \wedge (C_{\text{Keys}}(keys, fetch) \neq \text{t} \wedge (inh \neq \text{t})) \end{cases}$$

As the analysis cannot always classify an access as a definite hit or a definite miss, it has to safely approximate the set of open lines. As the name *potentially* open lines suggests, the set is approximated from above. If a BTIC miss ($C_{\text{Keys}}(fetch) \neq \text{t}$) cannot be excluded, *fetch* is included in *pol*. The join function is simply the set union:

$$\mathcal{J}_{\text{Pol}}(pol_1, pol_2) := pol_1 \cup pol_2,$$

where the cardinality constraint (at most 4 potentially open lines) is always implicitly satisfied.

Occupancy of open and closed lines. Since the filling process is non-trivial the analysis uses two maps, *ool* and *ocl*, to keep track of the occupancy of lines: The state of lines that are open, i.e., currently being filled, is conservatively approximated by

$$ool \in \text{OOL} := \text{A} \to \text{Line}$$

The state of closed lines is conservatively approximated by

$$ocl \in \text{OCL} := \text{A} \to \text{Line}$$

For instance, $ocl(a) = [\text{t}, \text{t}, \text{t}, \text{f}]$ means that *if* there is a line with key a that is closed, *then* its first three entries are occupied. On the other hand, $ool(a) = [\text{t}, \text{t}, \text{f}, \text{f}]$ means that *if* there is a line with key a that is still open, *then* it has been filled with two entries, so far. Further accesses could continue filling it. A line is allocated as soon as filling starts but its final occupancy is only determined when it is closed. Once it is closed the occupancy information about it is transferred from *ool* to *ocl*. $ocl(a) = \bot$ means that no closed line with key a can be in the BTIC. Similarly, $ool(a) = \bot$ means that no open line with key a can be in the BTIC. Note that at the same time $ool(a) \neq \bot$ and $ocl(a) \neq \bot$ is possible.

If the fetched address is within 4 instructions after a branch target *lb*, the respective position within a line is filled. Let pol' be the already updated potentially open lines after address *fetched* has been fetched. Then:

$$\mathcal{U}_{\text{OOL}} : \text{OOL} \times \text{A} \times \text{Pol} \to \text{OOL}$$

$$\mathcal{U}_{\text{OOL}}(ool, fetched, pol') := \lambda a. \begin{cases} fillpos(ool(lb), pos(lb, fetched)) & : lb \in pol' \\ \bot & : \text{else} \end{cases}$$

$$fillpos : \text{Line} \times \{0, \ldots, 3\} \to \text{Line}$$

$$fillpos([e_0, e_1, e_2, e_3], n) := [e_0, \ldots, e_{n-1}, \text{t}, \ldots, e_3]$$

In the update of OCL the analysis needs to distinguish between lines that are being closed and lines that are being invalidated. Lines are invalidated if only one entry is filled and a COF

occurs. The set of closing lines contains all lines that are completely filled or closed due to a COF.

$$invalidating := \begin{cases} \{lb \in pol \mid fetched = lb\} & : cof \\ \emptyset & : \text{else} \end{cases}$$
$$closing := (pol \setminus pol') \setminus invalidating$$

When a line is closed, its occupancy status with respect to the current filling process is known. Since the lines in *ool* are only *potentially* open, they are only potentially closed. Hence, upon closing a line (case 1 of the update) the analysis needs to safely combine the information in *ocl* and ool' (the already updated occupancy information for open lines). Upon a BTIC miss, a line is definitely opened, one can set $ocl(a) := \bot$ (case 2 of the update). If such a definitely opened line is closed, the join will simply transfer the information from *ool* to *ocl*.

$$\mathcal{U}_{\text{OCL}} : \text{OCL} \times \text{OOL} \times \text{A} \times \text{A} \times \text{Inh} \times \text{Keys} \to \text{OCL}$$
$$\mathcal{U}_{\text{OCL}}(ocl, ool', fetch, fetched, inh, keys) :=$$
$$\lambda a. \begin{cases} \mathcal{J}_{\text{Entry}}(ocl(a), ool'(a)) & : a \in closing \\ \bot & : C_{\text{Keys}}(keys, a) = \text{f} \wedge cof \wedge a = fetch \wedge (inh = \text{f}) \\ ocl(a) & : \text{else} \end{cases}$$

Finally, the joins of OCL and OOL are the point-wise joins of their respective entries; see above.

BTB interface implementation. Altogether, the content analysis interface for the MPC56x can be implemented by:

$$\text{Cont} := \text{Pol} \times \text{OOL} \times \text{OCL}$$
$$\mathcal{U}_{\text{Cont}}((pol, ool, ocl), fetch, fetched, inh, keys) := (pol', ool', ocl'),$$

where

$$pol' := \mathcal{U}_{\text{Pol}}(pol, fetch, fetch, inh, keys)$$
$$ool' := \mathcal{U}_{\text{OOL}}(ool, fetched, pol')$$
$$ocl' := \mathcal{U}_{\text{OCL}}(ocl, ool', fetch, fetched, inh, keys)$$

Again, the join $\mathcal{J}_{\text{Cont}}$ is the point-wise join of its components. Finally, the classification function is:

$$C_{\text{Cont}}((pol, ool, ocl), lb, a) := \begin{cases} e_{pos(lb,a)} & : ocl(lb) = [e_1, e_2, e_3, e_4] \\ & \quad \wedge\, 0 \le pos(lb, a) \le 3 \\ \text{f} & : \text{else} \end{cases}$$

7. Evaluation

7.1. Model Validation

To validate our model, we used a phyCORE-MPC565 evaluation board. We configured the MPC565 to fetch instructions from an external SRAM module. This allowed us to observe external bus events for instruction fetches (i.e., transfer-start, -acknowledge, etc.). The validation

```
 0: nop                    32: nop
 4: divw r8,r8,r7          36: nop
 8: cmp r7,r8              40: nop
12: bca 0x0D,0x02,<48>     44: nop

16: nop                    48: nop
20: nop                    52: cmpi r6,0x01
24: nop                    56: li r6,0x00
28: b <48>                 60: beq cr0,<16>
```

Figure 6: Test program to check whether branch-prediction influences the state of the BTB. Due to the wrongly predicted branch at 12, both branch target fetches (at 28 and 60) cause BTB-hits.

consisted in asserting that all observed behavior is covered by our analysis: the sequence of observed external bus events must be contained in the (over-approximation of) predicted behavior. This was true for all of our 35 test cases. We have designed these test cases to precisely answer several questions concerning properties of the MPC56x BTB. Among others, the following questions guided the development of the tests:

1. What is the size of the BTB?
2. How many entries can be stored in a line?
3. What is the employed replacement policy?
4. Is there a difference between conditional and unconditional change-of-flow instructions?
5. Does branch-prediction have an influence on the BTB?
6. What are the tags of the BTB?
7. How does the BTB determine a change-of-flow?

Figure 6 shows one of the test programs we have written to determine properties of the BTB. The measurements of this code snippet reveal that branch-prediction does have an influence on the BTB and shows how the MPC56x BTB determines a change-of-flow. After the branch instruction at location 12 has been fetched, the division instruction `divw` is still executing. Thus, the branch condition cannot be resolved and the MPC56x uses its branch prediction mechanism. In this particular case, the branch is predicted taken. Consequently, the processor fetches the instructions starting at location 48. As the branch condition turns out to be false after the division is finished, the processor flushes the prefetch buffer and then begins to fetch the instructions at 16. The tracing hardware showed that the instructions at the locations 16 and 48 are fetched only once. However, the program branches from 60 to 16 and from 28 to 48 a second time. Hence, one can conclude that both instructions must be cached in the BTB: The speculative branch from 12 to 48 and the "branch" back to the correct branch target 16 upon flushing the prefetch buffer, both influence the BTB as any other branch. Conclusion is that branch prediction does influence the BTB contents. Additionally, the measurements show that actions of the MPC56x BTB are not triggered by branch *instructions*. In fact, the BTB compares the address of last instruction fetch with the address of the current instruction fetch. In case the address of the current instruction fetch is not the logical successor of the previous one, a change-of-flow is detected, which is the trigger for BTB actions.

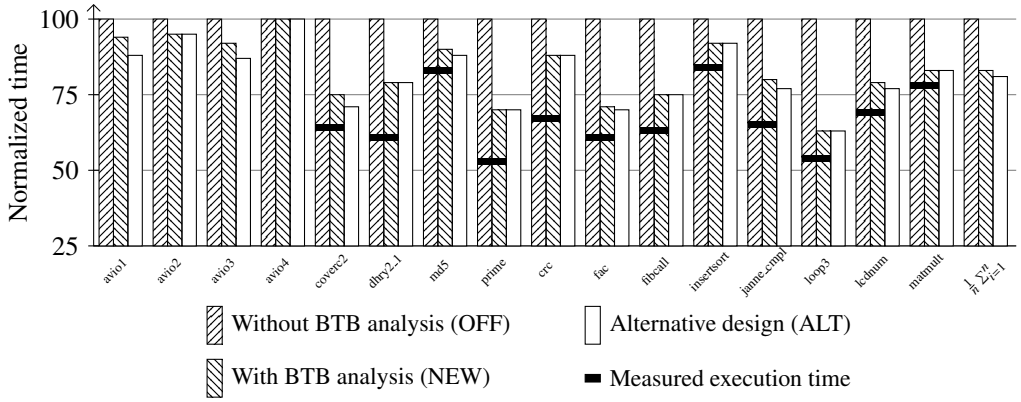


Figure 7: WCET bounds obtained by three different analyses. The sub-bars show measured execution times (with enabled BTB) where available. Time is normalized to the OFF-version.

7.2. Experimental Setup

We integrated our BTB analysis framework into the industry-strength timing analyzer aiT (http://www.absint.com), whose main architecture was already explained in Section 4. The benchmark programs are examples taken from a compiler test suite, the Mälardalen WCET benchmarks [26] (i.e., `crc`, `fac`, `fibcall`, `insertsort`, `janne_complex`, `loop3`, `lcdnum`, and `matmult`), as well as avionics hard real-time tasks running in production use on the MPC56x.

Due to non-disclosure agreements we cannot provide more details on those tasks. The system was configured with a memory latency of 4 cycles, assuming a rather fast external memory. Higher latencies would further increase the utility of the BTIC and our analysis.

Table 1 provides some hints about the expected gains with respect to the computed WCET bounds. For each benchmark, the table shows the number of loops, the average number of COF instructions inside loops, and the total number of COF instructions. The total number of COF instructions includes COFs outside of loops. Intuitively, one would expect (high) performance gains for programs with many loops each containing few COF instructions such that the BTB capacity is sufficient. Note that even if the number of COF instructions per loop is high, it might be that during execution only few of them actually result in a COF. Hence, a high number of COF instructions per loops does not necessarily preclude WCET improvements, which for instance is the case for `coverc2`.

7.3. Results

We examine the WCET bounds and their tightness obtained with three different versions of aiT. The NEW-version uses the instance of our BTB analysis framework for the MPC56x as presented in this paper. To quantify the improvement obtained by NEW, we compare it to OFF, which is identical except that it assumes BTB misses everywhere. The ALT-version (alternative design) is identical to NEW, except that it assumes an LRU-managed BTIC. ALT uses the cache analysis of [12] and its results are discussed in Section 8. To quantify the tightness of the analyses we also measured execution times.

Figure 7 shows the obtained WCET bounds and measured execution times. The difference between OFF and NEW is the improvement achieved by our analysis framework. It varies between the benchmarks: up to 37%, on average 17%. In general, NEW yields better improvements

for the non-avionics benchmarks, which can be explained by Table 1. In contrast to the compiler test suite and the Mälardalen WCET benchmarks, the avionic tasks feature few loops with small iteration counts. Thus, the performance gain inside these loops only has a small effect on the overall execution time (see Section 8 for a precise explanation). For the avio4 task, NEW cannot improve the WCET bound at all. The reason for this is that the there are too few control-flow changes during the execution of that task, such that the static analysis is not able to predict any hit.

Comparing the WCET bounds to measured execution times was only possible for the non-avionic tasks. We cannot execute the avionics tasks since we lack sensor input streams and the aircraft in which the processor is deployed. The overestimation (difference between predicted bound and actual WCET) was reduced by 66% (= $\frac{50-17}{50}$), from 50% to 17%. Note that 50% and 17% are upper bounds on the overestimation. Since we only measured *some* execution, it is very likely not the worst-case execution time. The true WCET is likely to be higher. Thus, the average overestimation is likely to be smaller than 17%.

Table 1: Number of loops, average number of COF instructions per loop, and total number of COF instructions.

		COF Instr.				COF Instr.	
Benchmark	Loops	Per Loop	Total	Benchmark	Loops	Per Loop	Total
avio1	3	5	695	crc	3	4	19
avio2	1	1	101	fac	1	3	6
avio3	1	1	1547	fibcall	1	3	5
avio4	2	2	14	insertsort	2	2	5
coverc2	8	40	336	janne_cmpl	2	4	11
dry2_1	7	3	62	loop3	120	2	241
md5	1	3	47	lcdnum	1	4	24
prime	1	3	13	matmult	5	2	22

8. Predictability Considerations

Section 5 and 6 provide insight about the effort needed to model and analyze the MPC56x BTIC. In this section we investigate the BTIC's predictability, propose alternatives, and quantify their influence on the runtime and the results of timing analysis. We generalize our findings and hope to sensitize hardware designers to problems in our domain.

The main challenge in a timing analysis for the MPC56x BTIC is the employed FIFO replacement policy. To see the difficulty consider the following example. After a first access, one knows that the accessed element must be cached—trivial must information is available. If one cannot classify the first access as a miss a second access (to another element) may actually evict the first element. This is the case if the first access was a hit on the first-in position and the second access is a miss. Thus, without classifying some accesses as misses it is impossible to infer that two or more elements are cached. I.e., non-trivial must-information can only be obtained by inferring and leveraging may-information. Informally speaking, without information about misses one can at any one time only classify the most recent access as a hit. Reineke et al. [27] show that for a k-way associative FIFO one needs to observe at least $2k - 1$ accesses to a set to be able

to classify an access as a miss. For this reason, there is no gain in the guaranteed performance of the avionic task `avio04`.

8.1. Explicitly Controlling Hardware State

Table 2: Classifications of BTIC accesses in % by the NEW-version with *unknown* and *empty* initial hardware state.

	Unknown			Empty		
Benchmark	Hit	Miss	Uncl	Hit	Miss	Uncl
coverc2	48.8	46.5	4.7	50.9	48.9	0.2
md5	17.1	19.3	63.6	44.3	55.7	0.0
prime	6.5	0.0	93.5	65.9	27.5	6.6
dhry2_1	44.3	22.0	33.7	57.6	42.4	0.0
avio1	25.9	3.1	71.0	36.1	22.5	41.4
avio2	21.0	73.9	5.1	21.0	77.2	1.8
avio3	22.7	1.1	76.2	29.4	27.9	42.7
avio4	0.0	0.0	100.0	4.5	42.4	53.1
Average	23.3	20.8	**55.9**	38.7	43.1	**18.2**

One way to attenuate the lack of FIFO may-information is to invalidate the BTIC contents at the start of the program. This way, one can safely assume an *empty* BTIC, i.e., at program start one gets complete must- and may-information for free. Otherwise, the initial BTIC state is *unknown* and the analysis has to conservatively assume any BTIC state. Note that cache information can be lost during the analysis, e.g., due to control-flow joins. Table 2 gives the respective classification ratios in % of all BTIC accesses classified during the analysis. Note that these are not the hit- or miss-rates on the computed WCET path. Only a fraction of the accesses encountered during the analysis lie on the WCET path. Assuming an empty BTIC results in fewer unclassified accesses, hence in less states to be considered, and hence in a shorter analysis time (see the next paragraph and Table 3). In our case assuming an empty BTIC reduced WCET bounds by 4.8% on average over the programs listed in Table 2. *Static analysis may profit from the availability and application of instructions to set a hardware component to a certain state.*

8.2. Predictable Replacement Policy

Compared to FIFO, for LRU replacement it is rather trivial to gain must-information. The cache state is fully determined after k different accesses. If one would take corrective action and replace FIFO by LRU the obtained WCET bounds would improve up to 5.3% (2.4% on average), as shown in Figure 7. Table 3 reveals that ALT is able to classify more accesses as hits or misses than NEW. Note that the analyses for Table 3 assumed any initial hardware state. This in turn saves analysis time and reduces memory consumption as the analysis has to explore fewer states. The only exception is `avio4`, for which both analyses perform equally bad. *If a hardware component must implement some kind of replacement,* LRU *should be employed.*

Table 3: Percentage of unclassified accesses and analysis time in seconds: NEW vs. ALT

	NEW		ALT	
Benchmark	Uncl [%]	Time [s]	Uncl [%]	Time [s]
coverc2	4.7	169.1	2.3	48.2
md5	63.6	42.5	33.6	0.5
prime	93.5	130.8	67.0	5.0
dhry2_1	33.7	5.3	10.9	0.2
avio1	71.0	12.9	15.9	1.3
avio2	5.1	41.3	2.0	17.2
avio3	76.2	25.9	21.1	1.4
avio4	100.0	2.9	100.0	0.9
Average	**55.9**	53.8	**31.6**	9.3

8.3. Monotonicity of Performance in Architectural Sizes

In another series of experiments, we looked at the influence of architectural size parameters. Table 4 lists the obtainable WCET bounds (using the NEW-version) for different associativities of the BTIC. For some tasks the analysis computes better WCET bounds for smaller associativities! One can observe this non-monotone behavior for the test cases `md5`, `prime` and `dhry2_1`. The reason is that there are not enough COF events for the analysis to infer may information and in turn better must information. Since the number of COF events needed to gain may information increases with the associativity, lower associativity may be better. Yet, decreasing the associativity further, increases the WCET bounds since there are more misses in the concrete execution. Similar phenomenons were observed in the context of paging in operating systems [28]. For an LRU-managed BTIC and an appropriate analysis such behavior is impossible since LRU satisfies the inclusion property [29]: larger buffers always subsume the contents of smaller ones. *The performance of a buffer should increase monotonically with its size.*

8.4. Complicating Interaction with Branch Prediction

The MPC56x uses static branch prediction (which is handled by another analysis of our pipeline analysis), i.e., the direction of the prediction is statically known. The only dynamic aspect regarding prediction is whether and when prediction starts, which depends on the timing of other instructions. This leads to the following uncertainty, i.e., cases to be considered. If branch prediction would falsely predict the branch as taken then the state of the BTB depends on the availability of the branch condition. If the condition is not yet evaluated, fetching continues at the branch target and a BTIC line is allocated. If the condition is already evaluated, fetching continues at the address following the branch instruction and the BTB is left unchanged. Hence, the BTB might cache instructions of wrongly predicted program paths (e.g., instructions 88 et seqq. in Figure 2) or even "dead" code. The number of falsely fetched instructions depends on the time it takes to resolve the respective branch condition. These facts forced us to integrate the BTB analysis into the pipeline analysis to achieve reasonable precision. The alternative would have been a separate analysis that runs prior to the main pipeline analysis. *Refrain from using complicated (dynamic) branch prediction schemes.*

8.5. Complicating Special Cases

The last thing we want to point out here is unexpected behavior of the BTIC. The BTIC logic invalidates a line if only one instruction would be cached in it (and does not reuse this line for the following branch). Firstly, this wastes resources, which harms performance. Secondly, we had to infer this by measurements because this behavior is not documented. Eliminating this behavior and allowing branch instructions at a branch target to be cached as well would improve the WCET bound by up to 20% (`prime`). Building a sound analysis would be much easier if correct and complete documentation be available. Admittedly, special cases can improve performance. But if a static analysis cannot prove that the preconditions for a special case will hold at runtime the guaranteed performance cannot be improved. When designing hardware for real-time systems, designers should reflect on how easy a human/static analysis can prove the preconditions for a performance-increasing feature. *The number of special cases should be kept reasonably small. Hardware components should behave uniformly.*

Table 4: Obtainable WCET bounds (in cycles) for a subset of the benchmarks with different associativities for the BTIC.

	Associativity			
Benchmark	2	4	6	8
coverc2	37477	31733	29871	29196
md5	12605	12569	12673	12687
prime	57605	39447	40546	40535
dhry2_1	13563	13510	13519	13615

9. Related Work

The key analysis part (and only this part) of the BTB analysis can be seen as a cache analysis of a fully-associative cache with FIFO replacement. Static cache analysis has been studied extensively [30, 31, 32, 12]: Mueller et al. [30] present a *static cache simulation* for direct-mapped instruction caches. It classifies instructions as *always-miss*, *always-hit*, *first-miss*, or *conflict*. White et al. [31] extend this work to data caches, where the main challenges lie in the analysis of accessed addresses. Furthermore, an instruction cache analysis for set-associative LRU caches is sketched. Li et al. [32] present a timing analysis based on integer linear programming (ILP) formulations. It can handle set-associative caches by encoding their concrete semantics using linear constraints. However, since this approach integrates pipeline, cache, and path analysis into one ILP, it suffers from complexity problems. In practice it is limited to direct-mapped caches and simple pipelines. Ferdinand et al. [12] introduce the concepts of may- and must-caches and present a precise, context-sensitive LRU analysis based on abstract interpretation.

However, almost all prior cache analyses assumed LRU replacement which is easier to analyze than FIFO. In contrast to FIFO, it is possible to obtain precise must-information for LRU replacement without any may-analysis. Reineke and Grund [25] introduced the notion of *relative competitiveness* between cache replacement policies, which extends the notion of competitiveness by Sleator and Tarjan [33]. The competitiveness of LRU(2k-1) relative to FIFO(k) allows us to use an existing *may*-analysis for LRU by Ferdinand et al. [12] as a *may*-analysis for FIFO. Our experimental evaluation demonstrates the practical utility of the theoretical results on relative

competitiveness. Recently, Grund and Reineke developed specialized analyses for FIFO [34, 35] that, being tailored precisely to FIFO behavior, deliver more precise results than the analysis based on relative competitiveness. It is future work to evaluate these new analyses in the context of the BTB studied in this paper.

We are not aware of previous WCET analyses that model *branch target prediction* or *branch target instruction caches*. There is a considerable body of work on the related topic of *branch prediction* in the context of WCET analysis [36, 37, 38, 39, 40]. Branch prediction is concerned with predicting whether conditional branches are taken or not, while branch target prediction predicts the targets of branches (their addresses or instructions at the target). Colin and Puaut [36] model parts of an LRU-controlled addr-BTB. However, they focus on the employed branch predictor: a local, dynamic, bimodal branch predictor that is contained in an LRU-controlled 4-way set-associative addr-BTB. They have no analogon to our content analysis, which is strictly necessary to analyze BTICs. Furthermore, they assume a uniform penalty and timing compositionality (in the sense of [41]) to compute a WCET penalty from the number of branch mispredictions they obtain. Combining dynamic branch prediction with branch target instruction caches, as the one discussed in this paper, may yield systems that are both highly *unanalyzable*, i.e., their static analysis is too complex and expensive, and *unpredictable*, i.e., computable WCET estimates are imprecise. Bodin and Puaut [39] and Burguière et al. [42] propose a compiler optimization for the more predictable static branch prediction scheme that is aimed at improving WCET estimates. X. Li et al. [37] integrate the analysis of local and global dynamic branch predictors into an ILP-based WCET analysis framework proposed by Y.-T. Li et al. [43]. This approach may suffer from complexity problems if the branch history table is not direct-mapped. Except for [36], the above approaches assume systems with branch prediction but no branch target prediction. Systems that employ branch prediction often also employ branch target prediction to avoid having to decode instructions before being able to speculate. To analyze such systems, our framework can be combined with the above analyses of branch prediction. The complexity of ILP formulations for different dynamic branch predictors is analyzed by Burguière and Rochange in [40]. They conclude that complex branch prediction schemes as found in modern hardware may not be tractable by ILP-based timing analysis.

There is a considerable amount of work that approaches the notion of predictability that range from reporting on the observation of the phenomenon [20] and its impact on WCET analysis up to a formal definition for the predictability of a single architectural component [27]. However, there is little progress towards a formal and general definition of the notion "predictability".

Thiele and Wilhelm [44] argue for new *design for predictability* discipline. They observe that many features of current hardware, directed at improving average-case performance, like out-of-order execution and dynamic branch prediction, are harmful to timing predictability. One way of maintaining high performance, while improving predictability, is to shift responsibilities from the processor to the compiler, i.e., moving from caches to scratchpad memories or from dynamic to static branch prediction. The PRET architecture is advocated by Lee and Edwards [45]. Its design goal is the *repeatability* of the timing behavior: Timing should be a property of a program, not that of a program executed on a particular hardware architecture. *Correct executions* of a program should have the same timing behavior independently of the particular architecture. To statically prove that *all* executions of a program will be *correct* requires a timing predictable architecture: Wilhelm et al. [41] discusses the influence of architectural features on static timing analysis. It gives recommendations on the kind of pipelines, buses, and memory hierarchies to use in embedded hard real-time systems.

10. Conclusions

We proposed the first framework for the worst-case execution-time analysis of BTBs/BTICs. It is based on abstract interpretation and models the evolution of the state of hardware components at the granularity of single CPU cycles. The framework implements analysis aspects that are common to all BTBs and defines interfaces for two modules, the key and the content analysis modules, that allow to adapt the analysis to the details of particular BTBs.

In a case study we implemented the framework's modules for the Motorola PowerPC 56x family, which is used in avionic and automotive hard real-time systems. The implementation of the key analysis module was particularly challenging: It required to solve the cache analysis problem for FIFO replacement. Based on relative competitiveness, we proposed the first FIFO may-cache analysis. Furthermore, we introduced a FIFO must-cache analysis that can exploit may-cache information.

To evaluate our analysis, we integrated it in our timing analysis framework that provides analyses for other micro-architectural features (pipelines, caches, etc.). Even for a main memory with low latency our BTB analysis improves the WCET bounds by 17% on average and reduces the overestimation by 66%, from 50% to 17%.

Our predictability considerations result in general advice, that, if followed, would make WCET analysis easier, faster and would result in better WCET bounds; in our case up to 20%.

Acknowledgments

We thank AbsInt for providing hardware resources and our colleagues for discussions about drafts of this article. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013, grant agreement n° 216008 (Predator).

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The worst-case execution-time problem—overview of methods and survey of tools, Transactions on Embedded Computing Systems 7 (3) (2008) 1–53.

[2] P. Cousot, R. Cousot, Building the Information Society, Kluwer, 2004, Ch. Basic Concepts of Abstract Interpretation, pp. 359–366.

[3] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[4] H. Theiling, Control flow graphs for real-time systems analysis, Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany (2002).

[5] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Symposium on Principles of Programming Languages POPL, 1977.

[6] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, New York, NY, USA, 1978, pp. 84–96.

[7] A. Ermedahl, J. Gustafsson, Deriving annotations for tight calculation of execution time, in: Euro-Par, 1997, pp. 1298–1307.

[8] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, R. van Engelen, Supporting timing analysis by automatic bounding of loop iterations, The Journal of Real-Time Systems (2000) 121–148.

[9] I. Stein, F. Martin, Analysis of path exclusion at the machine code level, in: C. Rochange (Ed.), WCET '07: Proceedings of 7th International Workshop on Worst-Case Execution Time (WCET) Analysis, 2007.

[10] J. Engblom, Processor pipelines and static worst-case execution time analysis, Ph.D. thesis, Dept. of Information Technology, Uppsala University (2002).

[11] S. Thesing, Safe and precise WCET determinations by abstract interpretation of pipeline models, Ph.D. thesis, Saarland University (2004).
[12] C. Ferdinand, R. Wilhelm, Efficient and precise cache behavior prediction for real-time systems, Real-Time Systems 17 (2-3) (1999) 131–181.
[13] T. Lundqvist, P. Stenström, Timing anomalies in dynamically scheduled microprocessors, in: RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 1999, p. 12.
[14] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, B. Becker, A definition and classification of timing anomalies, in: WCET '06: Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis, 2006.
[15] J. Reineke, R. Sen, Sound and efficient WCET analysis in presence of timing anomalies, in: WCET '09: Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis, 2009.
[16] Y.-T. S. Li, S. Malik, Performance analysis of embedded software using implicit path enumeration, in: DAC '95: Proceedings of the 32nd Design Automation Conference, 1995, pp. 456–461.
[17] H. Theiling, ILP-based interprocedural path analysis, in: EMSOFT, Vol. 2491 of Lecture Notes in Computer Science, Springer, 2002, pp. 349–363.
[18] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, R. Wilhelm, Reliable and precise WCET determination for a real-life processor, in: EMSOFT, Vol. 2211 of LNCS, 2001, pp. 469–485.
[19] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, C. Ferdinand, An abstract interpretation-based timing validation of hard real-time avionics software systems, in: DSN '03: Proceedings of the 2003 International Conference on Dependable Systems and Networks, IEEE Computer Society, 2003, pp. 625–632.
[20] R. Heckmann, M. Langenbach, S. Thesing, R. Wilhelm, The influence of processor architecture on the design and the results of WCET tools, Proceedings of the IEEE 91 (7) (2003) 1038–1054.
[21] Freescale Semiconductor, RCPU – RISC Central Processing Unit Reference Manual (February 1999).
[22] Freescale Semiconductor, MPC565 Reference Manual (November 2007).
[23] Freescale Semiconductor, MPC565.D Customer Errata and Information Sheet (September 2007).
[24] C. Ferdinand, F. Martin, R. Wilhelm, Applying compiler techniques to cache behavior prediction, in: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, ACM SIGPLAN, Las Vegas, Nevada, 1997, pp. 37–46.
[25] J. Reineke, D. Grund, Relative competitive analysis of cache replacement policies, in: LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems, ACM, New York, NY, USA, 2008, pp. 51–60.
[26] Mälardalen WCET benchmarks, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[27] J. Reineke, D. Grund, C. Berg, R. Wilhelm, Timing predictability of cache replacement policies, Real-Time Systems 37 (2) (2007) 99–122.
[28] L. A. Belady, R. A. Nelson, G. S. Shedler, An anomaly in space-time characteristics of certain programs running in a paging machine, Communications of the ACM 12 (6) (1969) 349–353.
[29] R. Mattson, J. Gecsei, D. Slutz, I. Traiger, Evaluation techniques for storage hierarchies., IBM Systems Journal 9 (2) (1970) 78–117.
[30] F. Mueller, D. B. Whalley, M. Harmon, Predicting instruction cache behavior, in: LCTRTS '94: Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, 1994.
[31] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, M. G. Harmon, Timing analysis for data caches and set-associative caches, in: RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium, IEEE Computer Society, Washington, DC, USA, 1997, p. 192.
[32] Y.-T. S. Li, S. Malik, A. Wolfe, Cache modeling for real-time software: Beyond direct mapped instruction caches, in: RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 1996, p. 254.
[33] D. D. Sleator, R. E. Tarjan, Amortized efficiency of list update and paging rules, Communications of the ACM 28 (2) (1985) 202–208.
[34] D. Grund, J. Reineke, Abstract interpretation of FIFO replacement, in: J. Palsberg, Z. Su (Eds.), SAS '09: Proceedings of the 16th International Symposium on Static Analysis, Vol. 5673 of LNCS, Springer-Verlag, 2009, pp. 120–136.
[35] D. Grund, J. Reineke, Precise and efficient FIFO-replacement analysis based on static phase detection, in: ECRTS '10: Proceedings of the 22nd Euromicro Conference on Real-Time Systems, 2010.
[36] A. Colin, I. Puaut, Worst-case execution time analysis for a processor with branch prediction, Real-Time Systems 18 (2-3) (2000) 249–274.
[37] X. Li, T. Mitra, A. Roychoudhury, Modeling control speculation for timing analysis, Real-Time Systems 29 (1) (2005) 27–58.
[38] I. Bate, R. Reutemann, Worst-case execution time analysis for dynamic branch predictors, in: ECRTS '04: Pro-

ceedings of the 16th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, 2004, pp. 215–222.

[39] F. Bodin, I. Puaut, A WCET-oriented static branch prediction scheme for real time systems, in: ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, 2005, pp. 33–40.

[40] C. Burguière, C. Rochange, On the complexity of modeling dynamic branch predictors when computing worst-case execution times, in: ERCIM / DECOS Workshop on "Dependable Embedded Systems", 2007.

[41] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, C. Ferdinand, Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems, IEEE Transactions on CAD of Integrated Circuits and Systems 28 (7) (2009) 966–978.

[42] C. Burguière, C. Rochange, P. Sainrat, A case for static branch prediction modeling in real-time systems, in: RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2005, pp. 33–38.

[43] Y.-T. S. Li, S. Malik, A. Wolfe, Cache modeling for real-time software: Beyond direct-mapped instruction caches, in: IEEE Real-Time Systems Symposium, 1996, pp. 254–263.

[44] L. Thiele, R. Wilhelm, Design for timing predictability, Real-Time Systems 28 (2-3) (2004) 157–177.

[45] S. A. Edwards, E. A. Lee, The case for the precision timed (PRET) machine, in: DAC '07: Proceedings of the 44th Design Automation Conference, 2007, pp. 264–265.