Branch Target Buffers: WCET Analysis Framework and Timing Predictability

Daniel Grund and Jan Reineke
Saarland University, Saarbrücken, Germany

Gernot Gebhard
AbsInt GmbH, 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 microarchitectural features like caches, branch prediction, and branch target buffers (BTB).**

We propose a modular WCET analysis framework for branch target buffers (BTB), 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 13% 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.

***Keywords*-Worst-case Execution Time (WCET) Analysis, Predictability, Branch-target-buffer (BTB)**

I. 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 overapproximate 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 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 IV. 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 V, 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 13% 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 54% to 20%.

Our last contribution is the identification of principles of more predictable hardware designs and their influence on WCET bounds. In Section VII, 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.

II. Foundations

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].

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 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 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.

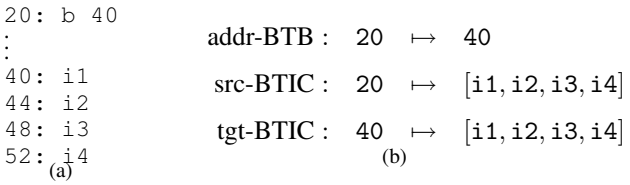


Figure 1. Three different types of BTBs.

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, 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. Figure 1 illustrates the differences between the three types of BTBs: (a) shows a single branch instruction and instructions `i1,...,i4` at the branch target. (b) shows the information stored for this branch in the three different types of BTBs. For details on BTBs confer [3].

III. Example: The MPC56x BTIC

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

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 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.

```
 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:
```

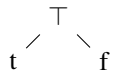
FIFO counter	Keys	Entries			
	4	[4]	[8]	[12]	[16]
	40	[40]	[44]	[48]	$\perp$
	$\perp$	[64]	$\perp$	$\perp$	$\perp$
	80	[80]	[84]	[88]	[92]
	88	[88]	[92]	$\perp$	$\perp$
→	108	[108]	[112]	[116]	[120]
	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Figure 2. State of an initially empty BTIC after execution of a program.

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.

Figure 2 show a program and its effect on an initially empty BTIC. `b` denotes a branch instruction, `bc` a conditional branch instruction. [a] denotes the instruction at address a. Each instruction is 4 bytes long. 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 that uses FIFO replacement. BTBs are also employed in various desktop CPUs, e.g., the Cyrix 6x86 or the Intel Pentium processor. A variation of a BTB is found in the PowerPC 750, which features a 4-way set-associative BTIC with 16 sets that stores up to two instructions per entry.



$\sqcup$	f	t	$\top$
f	f	$\top$	$\top$
t	$\top$	t	$\top$
$\top$	$\top$	$\top$	$\top$

$\wedge$	f	t	$\top$
f	f	f	f
t	f	t	$\top$
$\top$	f	$\top$	$\top$

Figure 3. The $\mathbb{B}^\top$ semi-lattice and its induced join $\sqcup$ and logical and $\wedge$.

IV. WCET Analysis Framework of BTBs

In this section we describe our main contribution, a framework for the WCET analysis of BTBs.

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 $\mathrm{cof} := \mathrm{fetched} \neq \perp \wedge \mathrm{fetch} \neq \mathrm{Succ}(\mathrm{fetched})$, where $\mathrm{Succ}(\mathrm{fetched})$ is the address immediately following fetch.

A. 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.

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.

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

see Figure 3, 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.

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

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

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

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.

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

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

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

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:

$$\mathcal{J}_{\text{BTB}} : \text{BTB} \times \text{BTB} \to \text{BTB}$$
$$\mathcal{J}_{\text{BTB}}(\text{btb}_1, \text{btb}_2) := (\text{inh}_1 \sqcup \text{inh}_2, \text{plb}_1 \cup \text{plb}_2, \mathcal{J}_{\text{Keys}}(\text{keys}_1, \text{keys}_2), \mathcal{J}_{\text{Cont}}(\text{cont}_1, \text{cont}_2))$$

The join $\sqcup$ of the two inhibited status is shown in Figure 3, and the potential last branches are joined by set union $\cup$.

B. 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:

$$\mathcal{U}_{\text{Keys}} : \text{Keys} \times \text{A} \to \text{Keys} \quad \text{(update function)}$$
$$\mathcal{J}_{\text{Keys}} : \text{Keys} \times \text{Keys} \to \text{Keys} \quad \text{(join function)}$$
$$\mathcal{C}_{\text{Keys}} : \text{Keys} \times \text{A} \to \mathbb{B}^{\top} \quad \text{(classification function)}$$

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 3. 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:

$$\mathcal{U}_{\text{Cont}} : \text{Cont} \times \text{A} \times \text{A} \times \text{Inh} \times \text{Keys} \to \text{Cont}$$
$$\mathcal{J}_{\text{Cont}} : \text{Cont} \times \text{Cont} \to \text{Cont}$$
$$\mathcal{C}_{\text{Cont}} : \text{Cont} \times \text{A} \times \text{A} \to \mathbb{B}^{\top}$$

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 ($\mathcal{C}_{\text{Keys}}(\text{keys}, \text{key}) = \text{t}$), then $\mathcal{C}_{\text{Cont}}(\text{cont}, \text{key}, \text{a})$ 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).

C. 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:

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

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:

$$\mathcal{C}_{\text{BTB}} : \text{BTB} \times \text{A} \to \mathbb{B}^{\top}$$
$$\mathcal{C}_{\text{BTB}}((\text{inh}, \text{plb}, \text{keys}, \text{cont}), \text{a}) :=$$

$$\bigsqcup_{\mathrm{lb}\in\mathrm{plb}} \left\{\mathcal{C}_{\mathrm{Keys}}(\mathrm{keys},\mathrm{lb}) \wedge \mathcal{C}_{\mathrm{Cont}}(\mathrm{cont},\mathrm{lb},\mathrm{a})\right\}$$

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.

V. 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 III and below was distilled from manuals [4], [5] and from measurements (see Section VI) that allow to precisely infer its functioning.

A. 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. [6], [7].

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* [8], 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 [7]. For $[T_0,\ldots,T_{k-1}] \in \mathrm{May}_{\mathrm{LRU}(k)}$, the analysis maintains the invariant that the disjoint union $\biguplus_i T_i$ 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 $\mathrm{Must}_{\mathrm{FIFO}(k)} := \mathcal{P}(\mathrm{T})^k$, 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:

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

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

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 $t \in \text{must}$ for $\exists i : t \in T_i$, where $\text{must} = [T_0,\ldots,T_{k-1}]$ and similarly for $t \in \text{may}$.

BTB interface implementation: For the MPC56X, the key analysis interface can be implemented by (with $\text{T} := \text{A}$):

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

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

B. 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 4:

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.).

3.) The BTIC line is *closed* on any of the following events:
a) 4 instructions have been inserted; the line is full.
b) Another COF happens, and at least 2 entries have been filled.

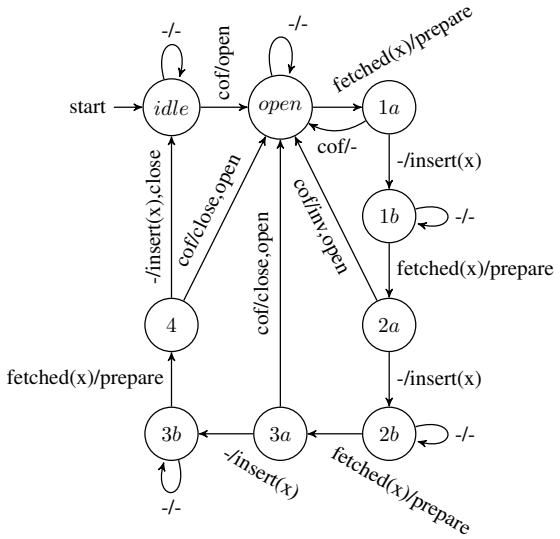


Figure 4. BTIC described by an automaton.

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*.

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:

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

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

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

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:

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

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 4. 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:

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

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 ($\mathcal{C}_{\mathrm{Keys}}(\mathrm{fetch}) \neq \mathrm{t}$) cannot be excluded, fetch is included in pol. The join function is simply the set union:

$$\mathcal{J}_{\mathrm{Pol}}(\mathrm{pol}_1, \mathrm{pol}_2) := \mathrm{pol}_1 \cup \mathrm{pol}_2,$$

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

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

The state of closed lines is conservatively approximated by

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

For instance, $\mathrm{ocl}(\mathrm{a}) = [\mathrm{t}, \mathrm{t}, \mathrm{t}, \mathrm{f}]$ means that *if* there is a line with key a that is closed, *then* its first three entries are occupied. On the other hand, $\mathrm{ool}(\mathrm{a}) = [\mathrm{t}, \mathrm{t}, \mathrm{f}, \mathrm{f}]$ 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. $\mathrm{ocl}(\mathrm{a}) = \bot$ means that no closed line with key a can be in the BTIC. Similarly, $\mathrm{ool}(\mathrm{a}) = \bot$ means that no open line with key a can be in the BTIC. Note that at the same time $\mathrm{ool}(\mathrm{a}) \neq \bot$ and $\mathrm{ocl}(\mathrm{a}) \neq \bot$ 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 $\mathrm{fetched}$ has been fetched. Then:

$$\mathcal{U}_{\mathrm{OOL}} : \mathrm{OOL} \times \mathrm{A} \times \mathrm{Pol} \to \mathrm{OOL}$$
$$\mathcal{U}_{\mathrm{OOL}}(\mathrm{ool}, \mathrm{fetched}, \mathrm{pol}') :=$$
$$\lambda\mathrm{a}.\begin{cases} \mathrm{fillpos}(\mathrm{ool}(\mathrm{lb}), \mathrm{pos}(\mathrm{lb}, \mathrm{fetched})) & : \mathrm{lb} \in \mathrm{pol}' \\ \bot & : \mathrm{else} \end{cases}$$
$$\mathrm{fillpos} : \mathrm{Line} \times \{0, \ldots, 3\} \to \mathrm{Line}$$
$$\mathrm{fillpos}([e_0, e_1, e_2, e_3], n) := [e_0, \ldots, e_{n-1}, \mathrm{t}, \ldots, e_3]$$

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.

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

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 $\mathrm{ocl}(\mathrm{a}) := \bot$ (case 2 of the update). If such a definitely opened line is closed, the join will simply transfer the information from ool to ocl.

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

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

BTB interface implementation: The content analysis interface for the MPC56X can be implemented by:

$$\begin{aligned}
&\mathrm{Cont} := \mathrm{Pol} \times \mathrm{OOL} \times \mathrm{OCL}\\
&\mathcal{U}_{\mathrm{Cont}}((\mathrm{pol},\mathrm{ool},\mathrm{ocl}),\mathrm{fetch},\mathrm{fetched},\mathrm{inh},\mathrm{keys}) :=\\
&\qquad\qquad (\mathrm{pol}',\mathrm{ool}',\mathrm{ocl}')\\
&\mathrm{pol}' := \mathcal{U}_{\mathrm{Pol}}(\mathrm{pol},\mathrm{fetch},\mathrm{fetch},\mathrm{inh},\mathrm{keys})\\
&\mathrm{ool}' := \mathcal{U}_{\mathrm{OOL}}(\mathrm{ool},\mathrm{fetched},\mathrm{pol}')\\
&\mathrm{ocl}' := \mathcal{U}_{\mathrm{OCL}}(\mathrm{ocl},\mathrm{ool}',\mathrm{fetch},\mathrm{fetched},\mathrm{inh},\mathrm{keys})
\end{aligned}$$

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

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

VI. Evaluation

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 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.

Setup: We integrated our BTB analysis framework into the industry-strength timing analyzer aiT (http://www.absint.com). The benchmark programs are examples taken from a compiler test suite, 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 further increase the utility of the BTIC and our analysis.

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 [7] and its results are discussed in Section VII. To quantify the tightness of the analyses we also measured execution times.

Figure 5 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 29%, on average 13%. For the avio4 task NEW cannot improve the guaranteed performance. This is due to imprecise information about register contents, which have an influence on conditional branches. This might be alleviated by additional user annotations or better preceding analyses.

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 63% ($= \frac{54-20}{54}$), from 54% to 20%. Note that 54% and 20% are upper bounds on the overestimation. Since we only measured *some* execution, it is very likely not a worst-case execution time. The true WCET is likely to be higher. Thus, the average overestimation is likely to be smaller than 20%.

VII. Predictability Considerations

Section IV and V 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. Reineke et al. [9] 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.

Table I
Classifications of accesses in %.

	Any			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**

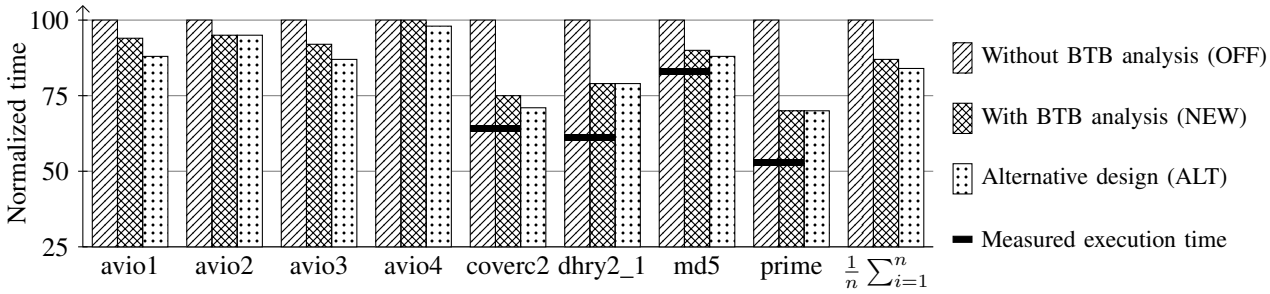


Figure 5. WCET bounds obtained by three analyses. Sub-bars show measured execution times.

Explicitly Controlling Hardware State: 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 (instead of *any* initial state), i.e., at program start one gets complete must- and may-information for free. Note, that cache information can be lost during the analysis, e.g., due to control-flow joins. Table I 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 II). In our case assuming an empty BTIC reduced WCET bounds by 4.8% on average. *Static analysis may profit from the availability and application of instructions to set a hardware component to a certain state.*

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 replace FIFO by LRU the obtained WCET bounds would improve up to 6% (3% on average), as shown in Figure 5. Table II reveals that ALT is able to classify more accesses as hits or misses than NEW. 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 II
UNCLASSIFIED ACCESSES AND ANALYSIS TIME.

	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

Table III
WCET BOUNDS (IN CYCLES) FOR DIFFERENT ASSOCIATIVITIES

	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

Monotonicity in Architectural Sizes: In another series of experiments, we looked at the influence of architectural size parameters. Table III 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 [10]. For an LRU-managed BTIC and an appropriate analysis such behavior is impossible since LRU satisfies the inclusion property [11]: larger buffers always subsume the contents of smaller ones. *The performance of a buffer should increase monotonically with its size.*

VIII. 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 [12], [7], [13]. Ferdinand [7] introduced the concept of *may*- and *must*-cache analyses, which we adopt. However, prior cache analyses assumed LRU replacement which is easier to analyze than FIFO.

Reineke and Grund [8] introduced the notion of *relative competitiveness* between cache replacement policies, which extends the notion of competitiveness by Sleator and Tarjan [14]. The competitiveness of LRU(2k-1) relative to FIFO(k) allows us to use an existing *may*-analysis for LRU by Ferdinand et al. [7] as a *may*-analysis for FIFO. Our

experimental evaluation demonstrates the practical utility of the theoretical results on relative competitiveness.

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 [15], [16], [17], [18]. 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 [15] model parts of an LRU-controlled addr-BTB. However, they focus on the employed bimodal branch predictor. They have no analogon to our content analysis, which is strictly necessary to analyze BTICs. Furthermore, they assume a uniform penalty and timing compositionality to compute a WCET penalty from the number of branch mispredictions they obtain. X. Li et al. [16] integrate the analysis of local and global dynamic branch predictors into an ILP-based WCET analysis framework proposed by Y.-T. Li et al. [19]. This approach may suffer from complexity problems if the branch history table is not direct-mapped.

Wilhelm et al. [20] 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.

IX. 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 13% on average and reduces the overestimation by 63%, from 54% to 20%.

Our predictability considerations are condensed to 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 this paper. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).

References

[1] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Systems*, vol. 7, no. 3, 2008.

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

[3] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[4] Freescale, *RCPU – RISC CPU Reference Manual*, 1999.

[5] ——, *MPC565 Reference Manual*, 2007.

[6] C. Ferdinand, F. Martin, and R. Wilhelm, "Applying compiler techniques to cache behavior prediction," in *LCRTS*, 1997.

[7] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Syst.*, vol. 17, no. 2-3, 1999.

[8] J. Reineke and D. Grund, "Relative competitive analysis of cache replacement policies," in *LCTES*, 2008.

[9] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, no. 2, 2007.

[10] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *CACM*, vol. 12, no. 6, 1969.

[11] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies." *IBM Syst.*, vol. 9, no. 2, 1970.

[12] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon, "Timing analysis for data caches and set-associative caches," in *RTAS*, 1997.

[13] H. Ramaprasad and F. Mueller, "Bounding worst-case data cache behavior by analytically deriving cache reference patterns," in *RTAS*, 2005.

[14] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *CACM*, vol. 28, no. 2, 1985.

[15] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Syst.*, vol. 18, no. 2-3, 2000.

[16] X. Li, T. Mitra, and A. Roychoudhury, "Modeling control speculation for timing analysis," *Real-Time Syst.*, vol. 29, no. 1, 2005.

[17] I. Bate and R. Reutemann, "Worst-case execution time analysis for dynamic branch predictors," in *ECRTS*, 2004.

[18] F. Bodin and I. Puaut, "A WCET-oriented static branch prediction scheme for real time systems," in *ECRTS*, 2005.

[19] Y.-T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: beyond direct mapped instruction caches," in *RTSS*, 1996.

[20] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, 2009.