# Instruction Caches in Static WCET Analysis of Artificially Diversified Software

**Joachim Fellmuth**
Technical University of Berlin, Berlin, Germany
joachim.fellmuth@tu-berlin.de

**Thomas Göthel**
Technical University of Berlin, Berlin, Germany
thomas.goethel@tu-berlin.de

**Sabine Glesner**
Technical University of Berlin, Berlin, Germany
sabine.glesner@tu-berlin.de

**Abstract**

Artificial Software Diversity is a well-established method to increase security of computer systems by thwarting code-reuse attacks, which is particularly beneficial in safety-critical real-time systems. However, static worst-case execution time (WCET) analysis on complex hardware involving caches only delivers sound results for single versions of the program, as it relies on absolute addresses for all instructions. To overcome this problem, we present an abstract interpretation based instruction cache analysis that provides a safe yet precise upper bound for the execution of all variants of a program. We achieve this by integrating uncertainties in the absolute and relative positioning of code fragments when updating the abstract cache state during the analysis. We demonstrate the effectiveness of our approach in an in-depth evaluation and provide an overview of the impact of different diversity techniques on the WCET estimations.

**2012 ACM Subject Classification** Software and its engineering → Real-time systems software

**Keywords and phrases** WCET, static analysis, abstract interpretation, artificial diversity, cache analysis

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2018.21

## 1 Introduction

Cyber-Physical Systems (CPS) have an ever increasing impact on our life as more systems are controlled by computers, which are highly interconnected and even connected to the internet. Among these systems are hard real-time systems such as airbag or ABS controllers, where missing a deadline is considered a system failure. If such a functionality is safety-critical, e.g. the ignition of an airbag, the developer is required to provide guarantees on safety and timing properties. To provide timing guarantees for given system, the worst-case execution time (WCET) needs to be determined. Static WCET analyses deliver a safe upper bound of the execution time of a task. In contrast, dynamic analyses under-approximate the execution time and are thus not feasible to be used in safety-critical systems.

When safety-critical systems are exposed to potential attackers, assuring safety implies also dealing with security issues. In particular, control-flow attacks are a threat to CPS because approximately 82% of the systems are developed in unsafe languages [12]. Also, CPS are often deployed in hostile environments. Existing run-time countermeasures cannot be applied due to limited resources or limited operating system support. Recent events

© Joachim Fellmuth, Thomas Göthel, and Sabine Glesner;
licensed under Creative Commons License CC-BY
30th Euromicro Conference on Real-Time Systems (ECRTS 2018).
Editor: Sebastian Altmeyer; Article No. 21; pp. 21:1–21:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

demonstrate that even in safety-critical applications, where strict regulations are in place, attacks are successfully mounted [28, 33].

Formal methods to statically prove the absence of vulnerabilities (e.g. Astrée [29] or CPAChecker [3]) often impose high usage effort and cost or considerable limitations in language and development possibilities, which keep them from being widely adapted in areas where it is not enforced by regulations. Also, defensive techniques that specifically target known attacks and vulnerabilities are often circumvented by new kinds of attacks.

Artificial software diversity [17, 23, 32] is an established way to enhance security in general purpose computing systems by thwarting code-reuse attacks such as return-oriented programming [6, 35]. The basic idea is to hide the memory layout from the attacker by compiling or loading semantically equivalent versions of the program with varying memory layout. Without detailed knowledge of the memory layout, it is considerably harder to mount a successful attack using existing code. Hiding the memory layout does not prevent attacks entirely, but it can lower the probability of success so that the attack becomes infeasible. Diversity also copes very well with new kinds of attack, provided the attack relies on knowledge of the memory layout, and, once introduced into the tool chain, does not require additional actions by the developer. In addition, diversity enables redundant systems, where independent replicas show the same intended behavior, but react differently to code-reuse attacks. As long as an attack on a replica does not interfere with the timing of other replicas, the system can tolerate a subset of the replicas to be compromised.

A WCET of a task in a diverse system has to be an upper bound for *all* variants of the program because the timing guarantees are only sound if they are guaranteed for any variant at any time. Existing static WCET analyses that incorporate instruction caches perform a detailed micro-architectural analysis that relies on absolute fixed instruction addresses. Using the diversification techniques we consider, the code is split into a fixed set of code parts, whose order is varied among the variants (we refer to these parts as fragments). Thereby, the absolute positions of fragments and their relative distances are unknown to the WCET analysis. So far, due to this contradiction, state-of-the-art static WCET cache analysis is not applicable to diverse systems, as it cannot guarantee an upper bound for all variants.

To overcome this problem, we introduce an instruction cache analysis, which is based on the abstract interpretation-based approach originally proposed by Ferdinand [15], and later improved by Ballabriga [1]. Our key idea is as follows: To ensure soundness, we assume that all instructions are possibly located in any location in a cache block, and we apply the worst-case cache behavior to all sets of blocks with unknown relative distance to the current basic block. Together, this ensures that our analysis neither relies on absolute instruction addresses nor on their relative distances, which both may be changed by diversification. To still be able to calculate tight upper bounds on the WCET, we retain all relative positioning within a fragment, we consider all possible absolute addresses of a fragment and we tightly limit the impact of cache accesses for cache contents of other fragments to the worst-case cache access.

Our approach universally supports all regular instruction cache architectures. In our experiments with small caches, the WCET estimates are tight, with an average over-approximation of 8.6%, compared to the highest WCET obtained by the non-diverse analysis applied to a number of variants. The estimates are a considerable improvement over an analysis without caches (assuming *all miss* for every memory access). Our benchmark results average at only 39.8% of the WCET without considering a cache.

The rest of this paper is structured as follows: In Section 2, we give an overview of artificial diversity techniques, and we introduce the basic concepts of current instruction cache analyses. We introduce our approach in three steps: First, in Section 3, we discuss the

impact of changes in absolute and relative position of code fragments on the cache behavior and analysis. Second, in Section 4, we present our analysis approach. Third, we introduce our worst-case cache hit classification in Section 5. Section 6 contains a detailed evaluation of our approach using well-known WCET benchmark programs. Section 7 contains a discussion of related work. And, finally, in Section 8, we conclude with a discussion of our findings and future work.

# 2 Background

In this section, we first briefly introduce artificial diversity, which serves as a basis for the different types of diversity we use in our evaluation. Then, we introduce the WCET instruction cache analyses our work is based on in Section 2.2.

## 2.1 Artificial Diversity

Artificial software diversity techniques [17, 23] are run-time countermeasures against control-flow attacks that are based on hiding information from the attacker by automatically creating many variants of the same program. The concept is based on the fact that the attacker needs information such as the detailed layout of parts of the memory to mount certain attacks successfully. Diversity is most useful against code-reuse attacks [4, 7, 34] on systems, where code injection is prevented using data execution prevention (DEP). The diversification can be introduced into every stage of the software development life cycle. Comprehensive overviews of control flow attacks and their countermeasures can be found in [35, 38]. It was demonstrated (e.g., [6]) that code-reuse attacks are a threat to CPS, many of which are safety-critical real-time systems.

The most prominent example of artificial diversity is address space layout randomization (ASLR) [5, 32]. In ASLR, the base addresses of some or all segments of the virtual memory of a process are randomized. ASLR is part of standard desktop operating systems such as Windows and Linux.

In addition to the segment-level diversity of ASLR, many other variations have been proposed, such as the substitution of instructions or small sequences with equivalent ones, garbage code insertion, function and function variable reordering, basic block level code shuffling, instruction-level diversity [23]. In earlier work [14], we have proposed a way to apply block-level diversity to safety-critical real-time systems.

In this paper, we concentrate on diversity techniques whose transformations are limited to relocating and reordering fragments of the code without changes in the control flow, code size, and instructions. These can be applied to the entire instruction memory, and enable us to precisely predict the WCET of all tasks of the executable. More specifically, we support the following kinds of diversity:

- **Segment-level diversity**: Similarly to ASLR, the entire text segment is relocated to a random position in memory, assuming a (virtual) memory space that is considerably larger than the program. In contrast to ASLR, segment-level diversity does not have to be aligned to memory pages. The segment (only one fragment segment-level diversity) can be located at any address, which includes addresses that are mapped to any offset in a cache line.
- **Function-level diversity** [22]: Just as the compiler is free to choose the order of functions and global data in the final executable, a variant can contain the functions in random order without any semantic difference to other variants. This enables a much larger number of possible variants than in segment-level diversity. The number of fragments equals the number of functions in the code.

- **Block-level diversity** [14]: The code is split into movable instruction sequences (MIS), which form the fragments, whose last instruction is an unconditional jump (e.g. `jmp`, `ret`). These fragments contain at least one basic block (BB) of the control flow graph (CFG).

The diversity techniques we consider can - for instruction cache analysis - be characterized by the diversity alignment and the fragmentation. We define the diversity alignment as the set of *offsets* $O = \{o_1, ..., o_K\}$ within a cache line an instruction can be located at. The number of offsets is equal to the cache line size divided by the alignment. For example, if a cache line has 16 Bytes, and the placement of code fragments is aligned to 4-Byte instructions (e.g. in ARM binaries), there are $K = 4$ different offsets a basic block can have relative to a cache line. If the alignment is greater or equal to the cache lines (e.g. 4 kByte pages in ASLR), there is only one offset $K = 1$. The number of fragments $N$, or, more specifically, the mapping of instructions and basic blocks to fragments depends on the diversification techniques mentioned above. We assume this information is given during our analysis using a function *frag:* $I \rightarrow F$, where $I$ is the set of instructions the program consists of and $F = \{f_1, ..., f_N\}$ the set of fragments.

## 2.2 Worst-Case Execution Time Analysis

Static WCET analyses typically consist of three parts: First, control flow and data flow analyses are used to create a model of the program in form of a control flow graph, flow facts such as loop bounds, and variable assignments or ranges. In a second phase, the micro-architectural analysis determines local timings, taking into account the actual timing behavior of the processor. Finally, the execution time is maximized over all control-flow paths, usually by representing the findings of the other phases as constraints in a linear program that can be solved by a linear program solver. This technique is called *implicit path enumeration technique (IPET)* [36]. While phase one and three are independent of the actual memory layout of the program on the target machine, phase two depends on the actual hardware and on the granularity of the analysis.

Instruction caches exploit the spatial and temporal proximity of executed instructions in the memory [27]. They are constructed so that instructions that are located close to each other are not conflicting. Therefore, the behavior of caches directly depends on the absolute position of each instruction (or basic block) and on the relative distance of code fragments that are executed on the same path. Any change in the program location or the order of code fragments will directly affect the WCET analysis result.

## 2.3 Instruction Cache Analysis

The state-of-the-art technique to represent caches in WCET analyses is based on Ferdinand et. al. [16]. There, an abstract interpretation based [8] separate cache analysis was introduced, which classifies memory accesses before the global WCET maximization phase. In this section, we briefly introduce the non-diverse analysis. Note that, while non-diverse analyses typically map a cache state from cache to memory, we define it the other way around. This helps in defining our analysis in a more compact way in Section 4.

### 2.3.1 Cache Definition

Caches are small buffer memories with shorter access times than the main memory. They are used to avoid long waiting times when accessing memory locations multiple times. They are mainly defined by the following values: The line size $S_L$ defines the number of bytes that is cached together (cache block), i.e. the size of the portion of main memory that is loaded on a cache miss. The associativity $A$ characterizes the number of locations, into which a memory block can be loaded. The capacity $S_C$ is the total number of bytes in the cache, with $n = \frac{S_C}{S_L}$ blocks in the cache. A set consists of all memory locations that can be loaded into a cache line, and the number of different sets equals $\frac{n}{A}$ . A cache with $A = 1$ is called direct-mapped, and a cache with $A = n$ is called a fully-associative cache.

As a replacement strategy, we focus on the *least recently used (LRU)* as it is most predictable, and we leave the investigation of other replacement strategies to future work. As we are using instruction caches and we assume that no write accesses can be made, write strategies are of no importance to our analysis.

### 2.3.2 Concrete Cache State

The cache itself is constructed as follows: A cache set is a sequence of cache lines $s_x = \{l_{x_1}, ...., l_{x_A}\}$, with x denoting the index of the set. Note that the order of the lines does not correspond to their location in memory. Instead, the index depicts the age of the line content in the LRU replacement, with 1 being the youngest. The whole cache is the union of all sets, $C = \bigcup\{s_1, ...., s_{\frac{n}{A}}\}$. We also assume a special cache line $\{l_\perp\}$, denoting the cache line that all memory blocks map to that are not currently in cache. The main memory is defined as a sequence of cache blocks in memory $M = \{m_1, ...., m_k\}$, with $k * S_L$ as the total program size. A cache state c is a function of each cache block to a cache line:

$$c : M \to C \cup \{l_\perp\}$$

The set of all cache states is denoted $\hat{C}$.

A *concrete cache state (CCS)* fulfills the property that at most one cache block can be mapped to each cache line:

$$c(m_1) = c(m_2) \to (m_1 = m_2 \lor c(m_1) = c(m_2) = l_\perp) \tag{1}$$

We use auxiliary functions: $set(m) : M \to \mathbb{N}$ maps cache blocks to cache set indices. And $age : \hat{C} \times M \to \mathbb{N}$ delivers the age of a cache block in the cache, or $\infty$ if not cached.

$$age(c, m) = \begin{cases} \infty & |c(m) = l_\perp \\ i & |c(m) = l_{s_i} \end{cases}$$

We define the empty cache as a function that maps all cache blocks to $l_\perp$:

$$c_\perp(m) = l_\perp$$

An update of the cache state $c$ at a memory reference $m$ using the LRU replacement strategy is described by an update function: $U : \hat{C} \times M \to \hat{C}$ that updates the mapping of all cache blocks $m'$ in a cache state $c$, resulting in a new cache state. The currently accessed cache block $m$ is in the first line of its set, $l_{s_1}$, as it is (most recently) accessed or loaded. If it was not in cache, all cache blocks $m' \neq m$ move one cache line "down" in age (degrade from $l_{s_{a'}}$ to $l_{s_{(a'+1)}}$), with the oldest one (with $a' = A$) being evicted (degraded from their previous cache line $l_{s_{a'}}$ to $l_\perp$). If the access to $m$ is a hit, only the cache blocks of the same

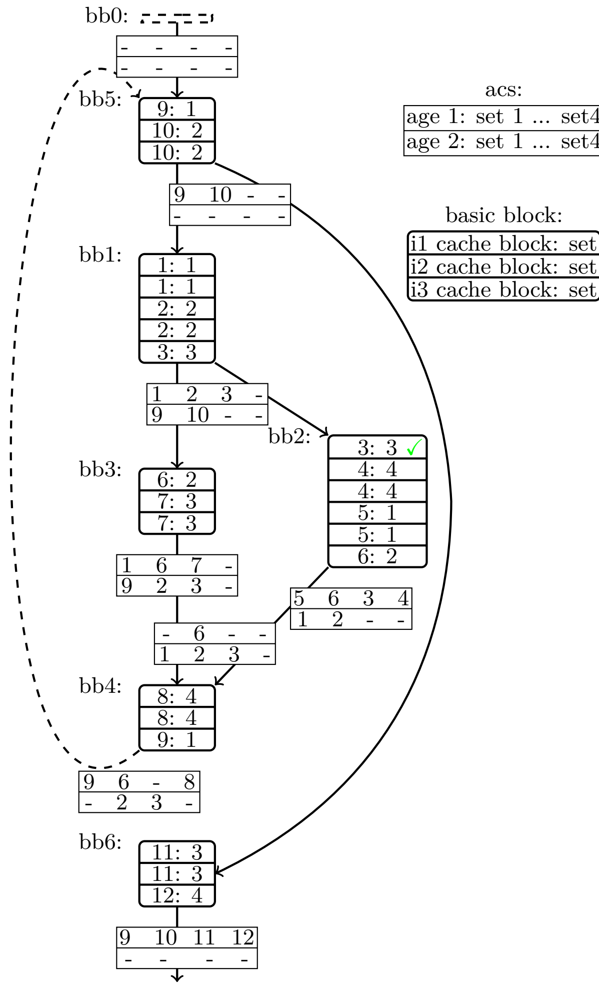



**Figure 1** Must analysis example.

set, which were cached more recently ($a > a'$), are degraded. In any case, cache blocks of other sets are not affected.

$$U(c, m) = U^h(c, m, set(m), age(c, m))$$

$$U^h(c, m, s, a)(m') = \begin{cases} l_{s_1} & |m' = m \\ l_{s_{(a'+1)}} & |m' \neq m \wedge c(m') = l_{s_{a'}} \wedge a' < A \wedge a > a' \\ l_\perp & |m' \neq m \wedge c(m') = l_{s_{a'}} \wedge (a' = A \wedge a = \infty) \\ c(m') & |otherwise \end{cases}$$

The concrete cache state after executing a path that contains a sequence of memory references $P = \langle m_1, ..., m_y \rangle$ is given as $c_P = U(...U(U(c_\perp, m_1), m_2)..., m_y)$.

Note that we define the cache states for the whole cache at once, although the behavior of the different sets is independent. This makes it easier for us to explain our own analysis later on.

### 2.3.3 Must Analysis

In an *abstract cache state* (ACS), more than one cache block can be mapped to a cache line, i.e. property (1) may not hold. From the cache perspective, each cache line can be associated with a set of cache blocks. With these abstract cache states, cache information of all program paths leading to the basic block can be accumulated. Using the theory of abstract interpretation and data flow analysis [9], the abstract cache states of the complete control flow graph are determined using a fix point algorithm.

The *LRU must analysis* is used to identify which cache blocks are *all hit (AH)*, i.e. cache blocks which never generate a cache penalty. The must analysis creates abstract cache states, where each memory block maps to the cache line that corresponds to its oldest possible age in the cache at the given program point.

An update of a cache block $m$ of an ACS in the must analysis $U_{must}$ is the same as the update $U$ of a CCS: Cached blocks of younger age are degraded, and blocks of age $A$ are evicted if there was a miss. An important difference to the CCS is that the ACS might contain cache blocks of the same age as $m$. These are not degraded, because the ACS holds the oldest possible age, so those blocks are in fact either younger than $m$, where degradation would lead to an age not older than $m$'s age, or older, and therefore do not need to be degraded. Given the incoming ACS, the cache behavior of the basic block is independent of the path that is currently executed. The relevant change is the introduction of the *join*-function that is used to merge two abstract cache states at program points where different paths join, i.e. at the start of basic blocks with at least two incoming edges. Here, for each memory block, the most pessimistic cache line is chosen, i.e. the line with the oldest associated age or $l_\perp$ if the memory block is not in the cache in at least one of the outgoing ACS of the basic blocks at the source of the incoming edges.

$$U_{must}(c, m) = U^h(c, m, set(m), age(c, m))$$

$$J_{must}(c_1, c_2)(m) = \begin{cases} l_\perp & |c_1(m) = l_\perp \lor c_2(m) = l_\perp \\ c_2(m) & |age(c_1, m) < age(c_2, m) \land c_1(m) \neq l_\perp \\ c_1(m) & |age(c_1, m) \geq age(c_2, m) \land c_2(m) \neq l_\perp \end{cases}$$

Figure 1 gives an example of the results of the must analysis. It contains a CFG of a short example program, and the corresponding ACS after the fix point of the analysis was reached (assuming `bb0` is the entry node and the cache is empty at start). The cache that is used here is a 2-way associative cache with four sets and the block size $S_L$ is set so that it contains two fixed-size instructions. The boxes with rounded corners are basic blocks, and they are split horizontally into one part per instruction. The first number depicted in each instruction is the cache block it belongs to and the second is its corresponding cache set. For example, the first two instructions of `bb1` are in cache block 1, which belongs to cache set 1. Along the edges of the CFG there are the ACS. The ACS tables contain a row for each age of cache contents with the youngest on top. The cache contents are arranged by set, starting at set 1 from the left. This way conflicting cache blocks are in the same column (e.g. cache blocks 1, 5 and 9 are conflicting in set 1). When a basic block is executed, the update function $U_{must}$ is applied to the incoming ACS, resulting in the ACS depicted at the outgoing edges. For example, in `bb3` the cache blocks 6 and 7 are accessed, which corresponds to changes in sets 2 and 3. After the update, cache blocks 6 and 7 are in the ACS at age 1. The conflicting cache blocks 2 and 3 are degraded from age 1 to age 2, and cache block 10 is evicted because it was already at age 2. The entry ACS of `bb4` shows an example application of the $J_{must}$: The oldest age of cache block 1 is age=2, therefore this is its resulting age. Cache block 7 is only cached after `bb3`, and not after `bb2`, therefore it does

not appear in the resulting ACS. The check mark marks the memory block access that is identified as a hit by the must analysis of this example.

### 2.3.4 May and persistence analysis

The may analysis is used to classify cache blocks as *all miss (AM)*. It is similar to the must analysis, with the difference that it associates each memory block with the cache line of the youngest age the memory block may have. Another important analysis is the persistence analysis to classify cache blocks as *first miss (FM)* [30], causing a cache penalty exactly once in a specific scope. The first abstract interpretation based analysis by Ferdinand [16] uses a converted may analysis, whose ACS contains the oldest possible age of a cache block up to age $A + 1$. Any block that cannot have reached age $A + 1$ at a program point is persistent. This analysis does not perform well on nested loops. Ballabriga [1] proposed multi-level persistence: A persistence level is determined for each loop block $m$ is part of. This is achieved using a stack of persistence ACS, where a new empty element is pushed on loop entry. Using this loop context information, the outermost loop in which the block is persistent can be determined.

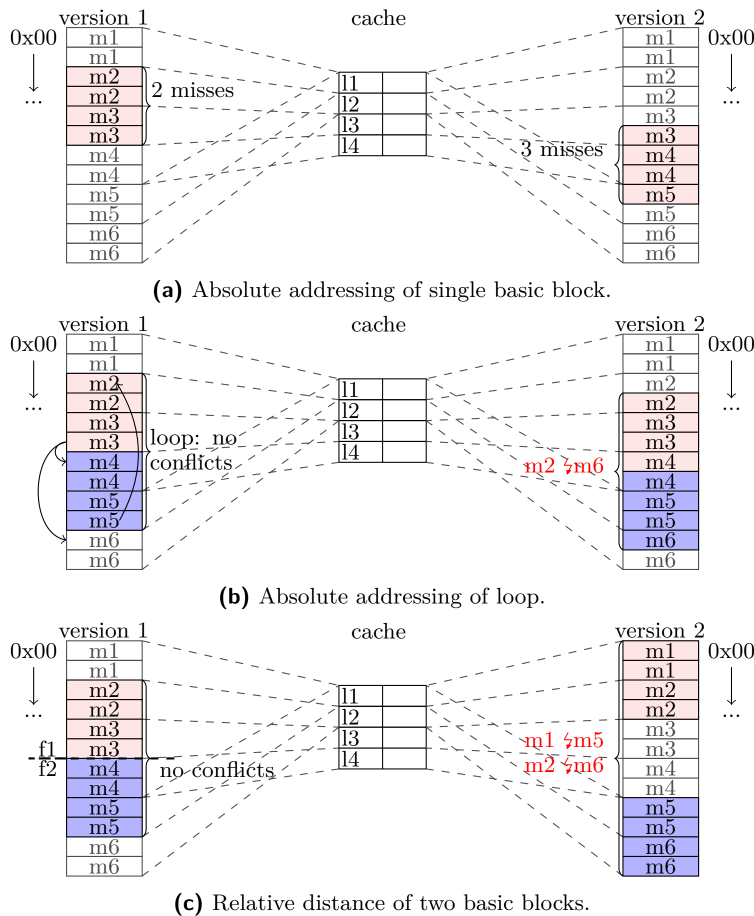
Cache blocks that cannot be identified by any of the analyses are classified as non-classified ($NC$), which, in a WCET analysis, is equal to $AM$.

## 3 Impact of Diversity on Caches

Instruction caches exploit the temporal and spatial proximity of the instructions executed alongside a path. Artificial diversity (relocating and reordering) impairs spatial proximity. Therefore, a negative impact on caching behavior and thus on the average performance can be expected. Moreover, static instruction cache analysis depends on absolute and relative addresses. Diversity worsens the predictability of the cache behavior because for every uncertainty, the worst case has to be assumed.

To clarify what the impact on the instruction cache analysis is, we discuss illustrating examples. Figure 2a and 2b show examples of the different caching behavior of the same basic blocks with different absolute positions. Figure 2a shows a basic block that in version 1 covers two cache blocks (m2, m3) and can therefore cause at worst two cache misses. In version 2, the same basic block is moved to another position where it covers 3 cache blocks (m3, m4, m5). Similar to that, Figure 2b shows a loop that just fits into the cache. Moving it by half a cache line, as in version 2, creates a conflict between the first and the last block. These examples show that the impact of absolute address changes of a basic block corresponds to different offsets of its instructions in the cache line. This is the case for every cache associativity.

Figure 2c shows two basic blocks whose relative distance differs in the two versions. Now the number of cache blocks per basic block is the same, but the two basic blocks cover conflicting cache blocks. In combination with the different possible offsets this means that every instruction of a basic block in fragment $f1$ can be in conflict with every instruction in fragment $f2$, located at every possible offset. The impact on relative addressing is a concern for caches with $A < n$, because changing distances of cache blocks might also change their set associativity.




**(a)** Absolute addressing of single basic block.

**(b)** Absolute addressing of loop.

**(c)** Relative distance of two basic blocks.

**Figure 2** Impact of basic block addresses on cache behavior.

The resulting insights of the impact of diversity are:

- Cache sets cannot be treated independently because every instruction of $f1$ may affect $f2$.
- Fragments can be located at different offsets relative to a cache line. As this may affect cache sets as well, the offsets cannot be handled separately as well.
- The number of cache blocks covered by a single basic block and the mapping of instructions to cache blocks differ in different versions. Therefore, a classification of cache blocks into the classes *AH, AM, FH* and *NC* is not feasible anymore.

## 4 Instruction Cache Analysis for Diversified Programs

Our instruction cache analysis is based on abstract interpretation similar to the analyses described in Section 2.3. We use the insights of Section 3 to define an ACS and its update and join functions so that we can cope with diversity. The key idea is as follows:

- For every fragment in $F$, we assume an own *virtual* memory in our ACS. This way we achieve independence of the fragments and the cache sets within and we get rid of overlapping cache blocks in adjacent basic blocks of different fragments. Note that the

size of the memory representation is similar to the memory before, because of the smaller size of the fragments.
- We create an ACS for every possible offset within cache lines a fragment can start at. That way, we can find the worst-case timing for a basic block.
- At the transition of control flow from one fragment to another, every relative distance of addresses is possible. To address this uncertainty, the worst case (that depends on the analysis) is applied to ACS for *all* offsets of cache blocks of other fragments. That way, we do not have to create a tree of ACS in combination of all possible fragment offsets along a path.

These features allow us to adapt to diversity and at the same time to keep and use all the information available during updates in the abstract interpretation fix point algorithm: Relative positioning within a fragment is fully available. Relative intra-fragment positioning information and absolute positioning information are reduced to the worst case of spatial locality, which still allows us to exploit the temporal locality of instructions.

Our abstract cache model for the must analysis is as follows: The cache $C$ is characterized as a set of cache lines (see Section 2.3). The memory model consist of $N$ different virtual memories $M$ consisting of cache blocks $m$, one for each fragment in $F$. To also represent all possibilities in absolute addressing ($K$ offsets), the memory representation results in $O \times F \times M$. Our abstract cache state function is adjusted accordingly to map every cache block of the new memory representation to a cache line:

$$\tilde{c} : O \times F \times M \to C \cup \{l_\perp\}$$

$\tilde{C}$ is the set of all cache states, and $\tilde{c}_\perp$ with $\tilde{c}_\perp(o, f, m) = l_\perp$ represents the empty cache state. As described in Section 3, there is no direct affiliation of an instruction with a cache block in our abstract model, as it depends on an offset. In addition to the function $frag : I \to F$, we assume the function $block : O \times I \to M$, which delivers the cache block $m$ of an instruction $i$ within its fragment, given that it starts at an offset $o$. Given the premise that all cache blocks of a fragment $f_1$ are potentially in conflict with the cache blocks of another fragment $f_2$, the notion of cache sets is only valid within a fragment. Also, as only the conflicts between blocks (the distance) are of importance and not the actual set (the absolute set number or address), we set the start of all fragments to the current offset $o$, starting in the first set.

We use the following auxiliary functions, to deliver the set number and the age of a cache block in a cache state analogous to Section 2.3, along with a shorthand notation for the age of an instruction $i$ at offset $o$ fetched from cache:

$$\begin{aligned}
&\tilde{set}(o, f, m) : M \to \mathbb{N} \\
&\tilde{age} : \tilde{C} \times O \times F \times M \to \mathbb{N} \\
&\tilde{age}_{inst} : \tilde{C} \times O \times I \to \mathbb{N} \\
&\tilde{age}_{inst}(\tilde{c}, o, i) = \tilde{age}(\tilde{c}, o, frag(i), block(o, i))
\end{aligned}$$

For better readability of the following definition of our must analysis, we define two additional functions: $deg : C \times \mathbb{N} \to C$ is used to update (degrade) a cache block mapping in a cache state. It selects a new cache line of the same set according to a given age $a$.

$$deg(l, a) = \begin{cases} l_\perp & | l = l_\perp \vee (a = \infty \wedge l = l_{s_A}) \\ l_{s_{(a'+1)}} & | l = l_{s_{a'}} \wedge a > a' \wedge a' < A \\ l_{s_{a'}} & | l = l_{s_{a'}} \wedge (a \geq 0 \wedge a \leq a') \end{cases}$$

$load_{wc} : \tilde{C} \times P(I) \to \mathbb{N}$ determines the oldest age of a hit over a set of instructions and all offsets, or a miss if any instruction misses at any offset. This resembles the worst-case cache degradation of cache blocks whose relative distance to the instructions in $i$ is unknown.

$$load_{wc}(\tilde{c}, \langle i_1, ....i_x \rangle) = \max_{\substack{i \in \{i_1,...,i_x\} \\ o \in O}} (\tilde{age}_{inst}(\tilde{c}, o, i))$$

Our update function $\tilde{U}_{must_{BB}}$ is applied at basic block level in two steps: First, we perform the update of cache block mappings in the ACS that belong to the same fragment as the basic block ($\tilde{U}_{must}$), and in a second step we degrade the cache blocks of the other fragments by the worst-case $\tilde{D}_{must}$.

$$\tilde{U}_{must_{BB}}(\langle i_1, ..., i_x \rangle, \tilde{c}) = \tilde{D}_{must}(\tilde{U}_{must}(...(\tilde{U}_{must}(\tilde{c}, i_1), ..., i_x), \langle i_1, ..., i_x \rangle)$$

As stated earlier, our update function $\tilde{U}_{must}$ has to be applied to instructions rather than cache block references, because the mapping between those differs with different offsets. The update is therefore performed per instruction $i$, which together define the virtual memory space $O \times F \times M$. Cache block mappings for each offset $o'$ in the same fragment are updated in the same way as in the regular must analysis: If they are in another set, they stay unchanged. Otherwise, they get degraded according to the previous cache state of $i$ and $o$.

$$\tilde{U}_{must}(\tilde{c}, i)(o', f', m') = \begin{cases} deg(\tilde{c}(o', f', m'), a) & | f = frag(i) \land m = block(i, o') \\ & \land f' = f \land \tilde{c}(o', f, m) = l_{s_a} \\ & \land set(o', f, m) = set(o', f', m') \\ \tilde{c}(o', f', m') & | otherwise \end{cases}$$

All cache blocks of other fragments get degraded in $\tilde{D}_{must}$ by the worst-case cache access (oldest cache hit or eviction $load_{wc}$) that any of the instructions of that basic block may suffer at any offset, as they potentially conflict with that *worst-case instruction*. This pessimistic way to cope with the uncertainty in relative distances enables us to keep the analyses and the ACS local, combining all possible paths in the original CFG and all possible combinations of offsets of fragments in a memory model that does not grow exponentially with the number of fragments.

Note that this definition of the worst-case cache access assumes that no cache conflicts can occur during the execution of a basic blocks, as the worst case access is a single miss. This can be achieved by limiting the size of a basic block to $\frac{S_C}{A} - \frac{S_L}{2}$ in the preceding CFG construction. The definition uses the implicit property of the diversification that all instructions of a basic block are part of the same fragment.

$$\tilde{D}_{must}(\tilde{c}, \langle i_1, ..., i_x \rangle)(o', f', m') = \begin{cases} deg(\tilde{c}(o', f', m'), load_{wc}(\tilde{c}, \langle i_1, ..., i_x \rangle) & | f' \neq frag(i_1) \\ \tilde{c}(o', f', m') & | otherwise \end{cases}$$

The join function is adjusted to the new memory model in the ACS. No additional pessimism is required at this point, because the worst-case cache behavior is already applied during the updates.

$$\tilde{J}_{must}(\tilde{c}_1, \tilde{c}_2)(o', f,' m') = \begin{cases} l_\perp & | \tilde{c}_1(o', f', m') = l_\perp \lor \tilde{c}_2(o', f', m') = l_\perp \\ \tilde{c}_2(o', f', m') & | \tilde{age}(\tilde{c}_1, o', f', m') < \tilde{age}(\tilde{c}_2, o', f', m') \\ & \land \tilde{c}_1(o', f', m') \neq l_\perp \\ \tilde{c}_1(o', f', m') & | \tilde{age}(\tilde{c}_1, o', f', m') \geq \tilde{age}(\tilde{c}_2, o', f', m') \\ & \land \tilde{c}_2(o', f', m') \neq l_\perp \end{cases}$$

Figure 3 illustrates the must analysis for diverse software. It contains the same program and cache setting as Figure 1. The cache blocks for each instruction $i$ are named $f.m$, where $f$ refers to the fragment and $m$ is the cache block index within this fragment. The instructions in the basic blocks (rounded rectangles) now contain the cache block every instruction maps to at the two different offsets, separated by slashes. The three dimensions of the ACS are displayed as tables with a column for each offset, and where the rows represent the worst-case cache ages for each cache block in cache, with the youngest at the top. In the update of `bb2`, we can see the regular must analysis within a fragment: In the offset `0` ACS, cache blocks `1.0` and `1.1` are degraded because they conflict with `1.4` and `1.5`. Cache block `1.2` stays in the same line because it gets accessed again. In the offset `0` ACS, `1.2` stays in the top line because there is no conflict with any of accessed blocks, whereas `1.0` and `1.1` are in conflict here as well. The update of the ACS of the different offsets is entirely independent. In `bb3` and `bb4`, we can see how a cache access in one fragment leads to updates of other fragments: the worst-case access to fragment 2 is a cache miss in both cases, and thus all cache blocks of fragment 1 get degraded once in `bb3` and `bb4` each.

The proposed analysis terminates because firstly, the number of memory blocks in the program and the number of blocks in the abstract cache state are both finite and secondly, the update and join functions are monotonous. We refrain from presenting a formal proof and a formal closure of the abstract interpretation, because our analysis differs from Ferdinand's must analysis only in the way the memory blocks are organized and in additional pessimism, resulting in faster evictions of cache blocks. The offsets enhance the ACS by another dimension, which increases the calculation effort, but does not affect the monotony.

## 4.1 May and Persistence Analysis

The *may analysis* and *persistence analysis* can be defined analogously to the *must analysis*. For space reasons, we keep our description brief and without the formal details, but refer the reader to Ferdinand [16] and Ballabriga [1].

In the may analysis, the cache states are updated so that cache blocks of other fragments are degraded using the youngest age of any cache access within the basic block, and the join function selects the youngest age for each cache block in $O \times F \times M$ out of the cache states.

Ballabriga [1] uses an optimization of the update function for efficiency: The persistence analysis is combined with the must analysis. If a block is accessed, that is not in the must ACS, the age of all other cache blocks is reduced.

Our persistence analysis applies the principles described for the must analysis to Ferdinand's and Ballabriga's persistence analysis: The memory is split into virtual memories $M$ that represent the fragments, and duplicated for each offset in $O$. The cache model is extended by an additional cache line of age $A + 1$, which is reached if a cache block, once loaded, may have been evicted. The update and join functions resemble a reversed may analysis, where each cache block in the ACS maps to the oldest age that may have been reached (which is $A + 1$ if the block may have been evicted). Ballabriga [1] uses an optimization of the update function for efficiency, which we adapt: The persistence analysis is combined with the must analysis. If a block is accessed, that is not in the must ACS, the age of all other cache blocks is reduced. Just as in our new must analysis, the update function is applied as usual to all blocks within the same fragment as the current basic block. The age of all other blocks is degraded by the worst possible eviction of all accesses in the basic block (up to $A + 1$).

In our multi-level persistence, analogously to Ballabriga, an stack of the ACS described above is created, where each entry represents loop nesting level. The stack is initialized with a regular persistence analysis at the highest level outside any loop. With every loop entry, an

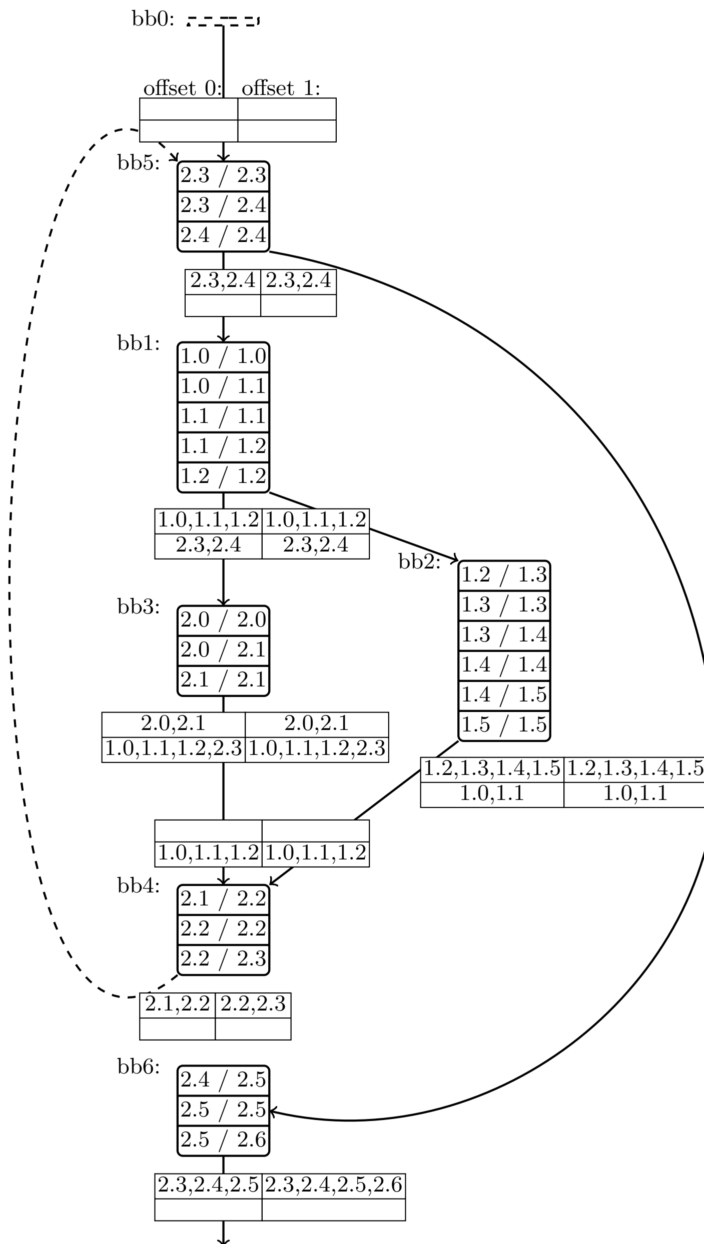



**Figure 3** CFG with ACS (must analysis at fix point) of example program using diversity: Two different offsets and fragments.

empty ACS is pushed onto the stack. The entries in this ACS that are in the same fragment as the accessed block can only be replaced by cache blocks, which are accessed within the loop. A cache block that is persistent with respect to a loop is executed as often as the loop is entered, at the worst. In the update function of the multi-level persistence, the cache blocks belonging to other fragments are degraded by the worst case access of the current basic block over all ACS and all offsets.

With our *must* and *persistence* analyses, we have established an abstract representation of all possible cache states before and after the execution of a basic block. To extract a worst-case timing of the execution of the block, we cannot use a direct mapping between instructions and cache accesses. Instead, we accumulate the overall block timing using a classification algorithm, described in the next section.

# 5 Classification

Our memory model in the ACS causes every instruction to be represented $K$ times, once for every offset, with possibly different cache classifications. These different representations need to be merged in order to be able to deduce a worst-case timing for the basic block. As stated in Section 3, the number of cache blocks may differ over different offsets, and instructions may belong to different cache blocks. Therefore, simply classifying the cache blocks is not feasible. Instead, we determine a worst-case number for each class at basic block level. We nevertheless use the term classification, analogously to previous publications [10, 39].

Finding the worst-case of cache misses $AM$ after applying only the must analysis is straightforward: For each offset $o \in O$, we count the referenced cache blocks that are not classified as a hit ($c(o, f, m) \neq l_\perp$), and use their maximum to calculate the worst-case cache penalty. This penalty is added to the WCET as often as the block is executed.

In the multi-level persistence analysis, the cache block references are additionally classified as persistent, with respect to the outermost loop it is persistent in, adding the classes $P(L0), P(L1)$.... The cache penalty of a cache block classified as $P(L_x)$ is added as often as the outermost loop it is persistent in is executed. The worst-case classification over all offsets is more complicated than maximizing the blocks for each persistence level separately. We use Figure 3 as an example. Basic block `bb3` contains three instructions, hence two cache blocks in each offset. With offset 0, cache block `2.0` is a miss, because after being loaded, it could be evicted by `bb1` and `bb5` (and possibly `bb2`) in the second iteration before being accessed the next time. `2.1`, however, is loaded in `bb4`. In loop iterations other than the first, `bb3`, `2.1` was only degraded once by `bb1`, and is therefore persistent (*P(L0)*). When located at offset 1, `2.1` is not accessed in `bb4`, and therefore both cache blocks cause a miss. To sum it up, at offset 0, cache accesses are classified as $AM = 1, P(L0) = 1$ and at offset 1, $AM = 2, P(L0) = 0$. To determine how many cache loads are necessary at worst for `bb3`, we have to combine the results of all offsets. Now assuming the loop is executed three times, simply counting the worst case of each class, $AM = 2$ and $P(L0) = 1$ would result in $3 * AM + P(L0) = 7$ times the cache penalty, whereas the true worst case is two misses, causing six loads in total. This example did not make use of the fact that the second $AM$ at offset 1, which is included in the combined worst-case, already covers the $P(L0)$ access at offset 0. We solve this by finding the worst case of all accesses for all offsets, instead of a separate worst case of each class.

Assuming a reducible CFG, the execution count of an inner loop is a multiple of the loop entry's execution count. Therefore, we can sort the classes, according to the loop nesting depth, by descending execution count: *AM,...,P(L1),P(L0)*. Our classification works as follows (we denote $c_0$ for the maximal number of misses and $c_i, (i > 0)$ as the maximal number of persistent blocks of each persistence level starting at $c_1$ as the persistence level of the inner loop where the block resides. $a_{i_o}$ are the numbers of misses and persistent blocks in offset $o$):

$$c_i = \begin{cases} \max\limits_{o=1..K}(a_{0_o}) & |i = 0 \\ \max\limits_{o=1..K}(a_{i_o} - \sum_{k=1}^{i-1}(c_k - a_{k_o})) & |i > 0 \end{cases}$$

We find the access counts of each class in the worst case by reducing the count of cache blocks per offset that are in that class by the number of blocks already covered by classes with

higher execution counts. In the following example, $max(P(L2))$ is 1 instead of 2, because in offset 1, one cache block is already covered by an extra miss that was identified for offset 0.

| Offset | Miss | P(L2) | P(L1) | P(L0) |
|---|---|---|---|---|
| 0 | 2 | 1 | 1 | 1 |
| 1 | 1 | 2 | 2 | 0 |
| max | 2 | 1 | 2 | 0 |

Our classification does not contain the classes *NC* and *AH*. The above maximum results for misses and persistence already contain a worst case for the timing behavior of the basic block, and thereby include the blocks before classified as *NC*. Note that we assume that the processor does not exhibit timing anomalies. We leave that investigation to future work.

The blocks classified as *AH* would resemble the "rest" of cache blocks, which are not included in the other classes. This differs between the offsets and its maximum does not represent useful information to the WCET analysis. If we assumed that hits also consume time, we would have to include the hits as an extra class of the above classification algorithm.

Using the preceding analyses and our classification we can generate additional ILP constraints for each class per basic block. These worst-case execution counts that represent a safe upper bound for the execution of all variants of the diverse program. In the following section, we present the evaluation we performed on our approach.

## 6 Evaluation

To evaluate our instruction cache analysis for diversified software, we implemented it in OTAWA, an open research framework for static WCET analysis, [2], and compared its original analyses [1] with our results. We implemented the *must*, *may* and *multi-level persistence analysis* as described in Section 4.

Our experiments were performed using the *Mälardalen benchmark suite* [18]. This set of small example programs is widely used for evaluating static WCET analyses, and therefore enables transparent evaluation results. Out of the Mälardalen benchmarks, nine programs were supported by OTAWA in the publicly available form, when applied to our ARM processor setup including cache analysis. Most of the other benchmarks were dismissed by OTAWA mainly because the ARM processor does not support floating point arithmetic, the flow facts (loop bounds, indirect branches) could not be deduced automatically, or external library functions were used, e.g. to emulate a division operation. In addition, OTAWA's original cache analysis, which we use to compare our results, does not support programs with basic blocks that are too large for the cache size we selected (basic blocks containing conflicting cache blocks cause unnecessary misses).

To increase the number of benchmarks, we slightly modified seven programs by inserting missing compiler functions, simplifying loop bounds and splitting the large basic blocks so that they fit the cache sets without conflict. There were no such fixes available for the remaining benchmark programs. However, the language features being exploited in the benchmarks is not the focus of this evaluation, as we are interested in the binary level. Also, we do not expect the additional pessimism of our analysis compared to state-of-the-art analyses to increase considerably in larger benchmarks, because larger code sections mostly introduce more independent sections with greater temporal distance, less relevant to caching.

The final set of benchmarks we used for this evaluation is summarized in Table 1. In addition to the benchmark name and its size in bytes, the table contains the number of fragments in function-level diversification (Functions) and in block-level diversification

**Table 1** Evaluated Malärdalen benchmarks. (*) marks benchmarks with minor changes.

| Benchmark | Size(B) | Functions | MIS | $WCET_{AM}$ |
|---|---|---|---|---|
| bs | 256 | 3 | 6 | 2445 |
| bsort100 | 492 | 3 | 8 | 7411365 |
| cnt | 676 | 7 | 12 | 111195 |
| crc | 1092 | 4 | 11 | 889455 |
| edn* | 4104 | 10 | 29 | 2208345 |
| expint* | 1332 | 6 | 19 | 6660570 |
| fac | 192 | 2 | 4 | 5550 |
| fdct* | 2796 | 3 | 15 | 81795 |
| fir* | 1152 | 4 | 14 | 25809855 |
| fibcall | 192 | 2 | 3 | 8370 |
| insertsort | 324 | 2 | 4 | 25815 |
| jfdctint* | 2792 | 3 | 16 | 105675 |
| matmult | 676 | 6 | 11 | 5122410 |
| ndes | 3180 | 6 | 27 | 1440465 |
| nsichneu* | 30316 | 1 | 2 | 227130 |
| ud* | 2320 | 5 | 22 | 839430 |

(Movable instruction sequences - MIS). At last, the table shows the $WCET_{AM}$ that has to be assumed without our analysis, where every access is a miss ($AM$). The benchmark programs, which had to be modified for being supported are marked with an asterisk.

For our experiments, we used a simple 5-stage architecture, where each instruction consumes five cycles and where cache miss adds a latency of ten cycles. We selected a cache size so that the small benchmark programs do not fit entirely and caching effects are visible: The line size $S_L$ is 16 Bytes, i.e. four instructions in the ARM instruction set. The diversity is aligned at the instruction size, resulting in *four offsets*. The number of sets (rows) is 16, and the associativity varies in the experiments. For our experiments, we created different variants of the benchmark programs using different offsets and random order of fragments: 30 for block-level diversity and 10 for function and segment level, respectively. To include the full range of possible offsets for all fragments, we added a random number of `NOP` instructions before the start of the program. The variants were created using the diversification program similar to the one we in introduced in [14].

Using the given setup, we analyzed each benchmark version using different cache dimensions. To obtain a comprehensive insight into the timing effects of our cache analysis, we needed to measure both the absolute timing improvements - compared to a system without caches - and the relative effect - compared to the cache analysis without diversity. We can measure the absolute effect by comparing to the WCET of a system without caches (*all miss* assumption) $WCET_{AM}$, as depicted in Table 1. The relative comparison is done using the WCET obtained by the original analysis without diversity, as implemented in the publicly available version of OTAWA. We calculated this value for each variant of each benchmark, together with the result of our analysis. We define $WCET_{max}$ the highest WCET obtained by that original analysis across all variants of a benchmark using the same cache setup. $WCET_{min}$, accordingly, is defined as the lowest WCET across such variants. Average values across all benchmarks are given as arithmetic mean. OTAWA also offers the possibility of using loop unrolling. We disabled that option in all our experiments.

One particularly interesting cache architecture for our evaluation is a cache with only one line (an instruction buffer). Diversity generally impacts spatial locality, because the execution order of the instructions remains intact and therefore subsequent executions of

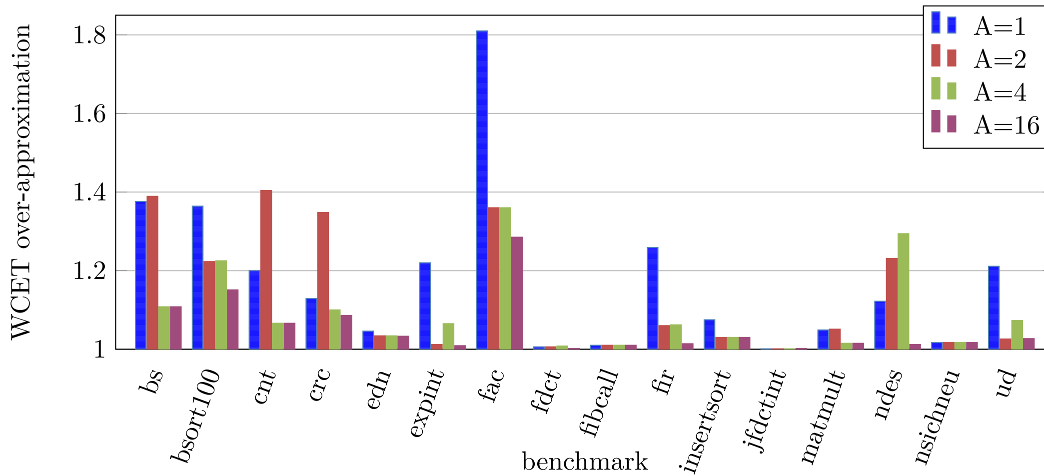



**Figure 4** Diverse WCET pessimism with different associativity A with 16 Bytes per line and 16 sets.

instructions have the same temporal distance. In the case of subsequent instructions the temporal locality of the cache block of an instruction is impacted as well, because, depending on the current offset, two instructions may or may not cause a cache line to be accessed twice in a row. To evaluate the performance effect of subsequent accesses to the same cache line, we calculate $WCET_{buf}$, using our analysis on a one-line cache with four instructions (16B).

A necessary requirement for our analysis is that it delivers a safe upper bound that is the same for all versions of a program given a cache size. This requirement was fulfilled in all our experiments: The estimates were always "worse" than those of the original analysis ($WCET_{div} > WCET_{max}$), and they were equal across all versions in the same experiment.

Figure 4 shows the tightness of the estimates using different associativities $A$ and block-level diversity. The WCET estimates $WCET_{div}$ are depicted relative to the corresponding $WCET_{max}$. The results show that the tightness of our analysis greatly depends on the associativity. The over-approximation is the highest for direct-mapped caches, and gets considerably tighter with $A > 2$. The average factor by which our analysis over-approximated in all experiments was 11.6% for basic-block level diversity, 4.3% for function-level, and 1.1% for segment-level diversity, which gives a total of 5.7%.

Figure 5 compares the results of our analysis with the range of non-diverse WCET estimates for all versions of a benchmark, using associativity $A = 2$ and basic block shuffling. There, $WCET_{max}$ and $WCET_{min}$ are depicted alongside the result of our analysis $WCET_{div}$. All results are shown relative to $WCET_{AM}$, the WCET using the *all miss* assumption that is the only choice for a static WCET analysis of diverse programs using the existing approaches. The results show that our analysis is a drastic improvement over assuming *all miss*: The estimates are on average at 44.8% of $WCET_{AM}$. Averaging the results of all experiments with all diversity types and associativities yields 40.8%. Furthermore, the diagram shows the impact of diversity on the cache analysis: $WCET_{max}$ is in average 5.3% higher than $WCET_{min}$. The diagram also shows $WCET_{buf}$, demonstrating the impact of immediate temporal locality. As expected, this effect is responsible for the bulk of the WCET gains, and in some benchmarks (such as `bs`) $WCET_{buf}$ is almost equal to $WCET_{div}$. However, this is not the case for most benchmarks, so that $WCET_{buf}$ averages at only 56% of $WCET_{AM}$.

Figure 6 shows the impact of fragmentation on the analysis result. The WCET estimates for associativity $A = 2$ are depicted for all diversity types, again with $WCET_{div}$ relative

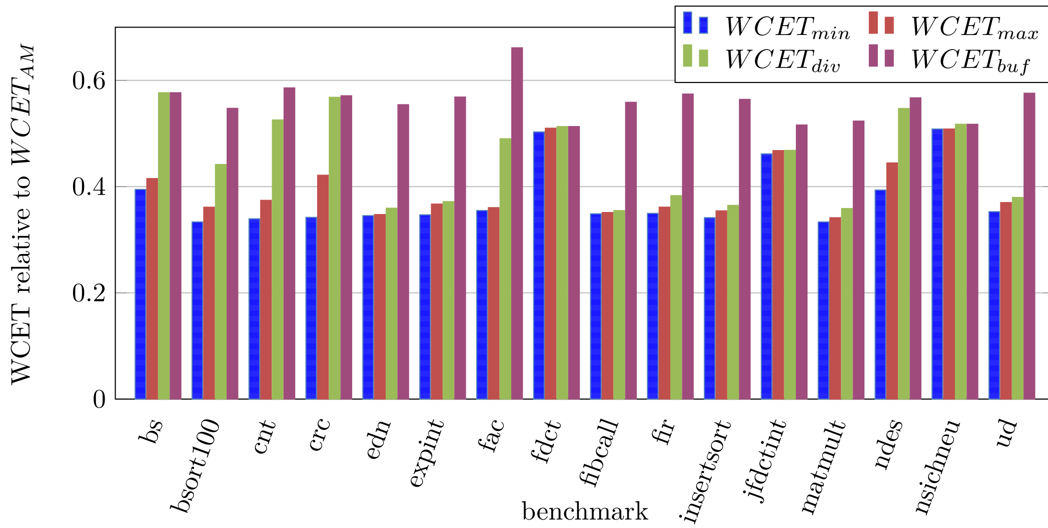



**Figure 5** Comparison of WCET estimates using 2-associative cache with 16 Bytes per line and 16 sets.

to $WCET_{max}$ . The results show that basic-block diversity produces the highest estimates (avg. 13.8%), while the segment-level diversity causes very low additional pessimism (avg. 1.2%). Function-level diversity averages at 6.0%.

Figure 7 shows the computation time compared to that of the original analysis. We measured the total execution time of each of the above experiments. The diagram shows the average factor by which the computation time exceeds that of the original analysis, in dependence of associativity A and the diversity type. We have expected an increase of computation time, as the ACS is a multiple of the size of the original analysis, because it contains a memory representation per offset. However, to deliver a sound upper bound, the original analysis would have to be executed for each possible version, which would take considerably longer depending on the number of fragments. The results for $A = 1$ show a factor eight in consumed time, while the number of offsets is $K = 4$. The time also increases with associativity A, but not proportionally. We have observed that it took more iterations for the fixpoint algorithm to terminate, and this increased with higher associativity. It can partly be explained with the fact that the cache sets are not independent in our analysis. The comparison of execution times of different diversity types delivers somewhat surprising results: The analysis time increase for segment-level diversity is the highest, while function level diversity performs best.

Note that in theory the performance analysis of OTAWA still has the bug referenced by Cullmann [10], which causes restrictions on the use of indirect branches. Our approach does not fix that, however, we think the demonstration of our approach using the proposed persistence analysis is representative as the error has a low impact, in particular on instruction caches. However, we plan to apply our solution to other persistence analyses as well.

To summarize our results, we can conclude that diversity is not prohibitive for WCET cache analysis. On the contrary, in many experiments our analysis results are very tight, and in all experiments they were far lower than the WCET without analyzing the cache.

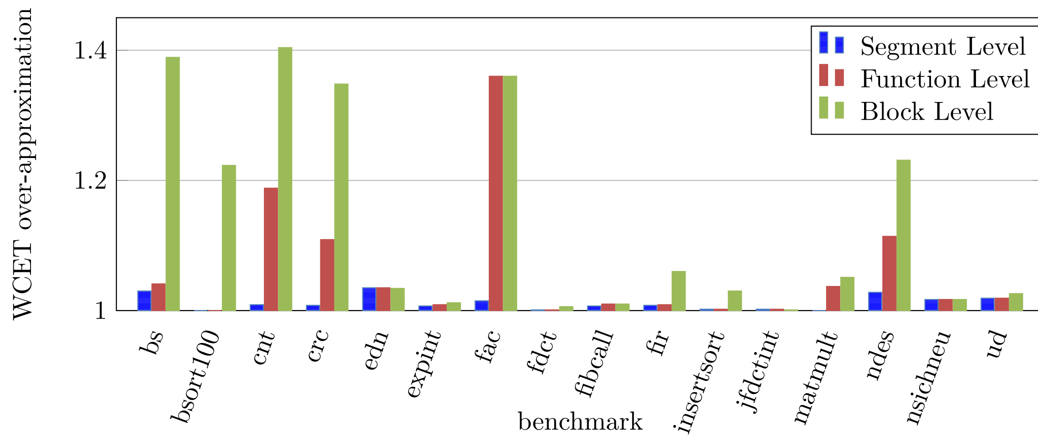



**Figure 6** Diverse WCET pessimism with different types of diversity (number of fragments).

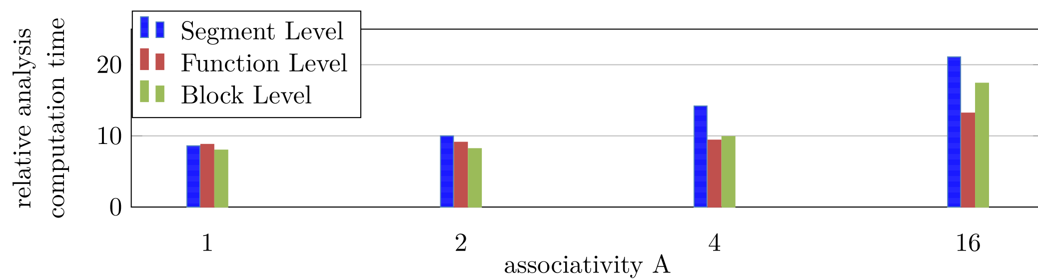



**Figure 7** Relative WCET Computation time increase.

## 7 Related Work

To our knowledge, there is no approach of static WCET cache analysis that supports artificial software diversity. However, many of the existing approaches serve as a basis for our approach or may be complemented with our ideas.

### 7.1 WCET Analysis

Our approach is based on Ferdinand's abstract interpretation-based analyses [15, 16], as well as Ballabriga's extension, the multi-level persistence [1]. Other approaches improved the persistence analysis as well. Cullmann [10] proposed two analyses, one based on conflict counting and one that combines a regular may analysis with one that collects the oldest age. Huynh [21] collects a set of conflicting blocks that may be younger than a block $m$ at its execution. If there are less than $A$ blocks in this set, $m$ is persistent. These analyses are based on abstract interpretation, thus we believe that our ideas are applicable there as well.

Apart from abstract interpretation there are other static cache analysis techniques, comprehensively surveyed in [27]. [24] introduces cache conflict graphs, explicitly modeling all concrete cache states. [26] uses model checking, which also, implicitly, analyzes all cache states. These approaches suffer from scalability issues, which would be worsened considerably by extending the state space with the uncertainty of diversity.

In [19] also investigates relative addressing of cache contents. However, they concentrate on data flow analyses for data caches, rather than implicit relations between instructions.

WCET-aware compiler optimizations such as code positioning [13] [25] are used to improve the WCET by varying compiler decisions similar to diversity. However, these approaches are based on heuristics and are not able to find a guaranteed upper or lower bound.

## 7.2 Artificial Diversity

Out of the many artificial software diversity approaches [23], we chose those that limit their transformations to reordering and relocating of fragments, for the reason that we need to be able to predict the WCET of all possible versions. This would be possible as well for narrow-scope in-place code transformations [31]. However, it would require a more detailed analysis of the instruction semantics and would further complicate the ACS construction.

There are also basic-block level approaches that propose splitting the code dynamically into blocks at random positions [11,37] for each variant. This is not supported as they change the CFG and thus the WCET impact over all variants would be unpredictable statically.

Instruction level diversity [20] uses higher fragmentation than block-level diversity. We expect its caching behavior to be very bad, and the analysis is futile as the relative distances of all instructions are unknown.

# 8 Conclusion

The instruction cache analysis we propose fills an important gap in the research of static WCET analysis, as it is the first to support artificially diversified programs. The key idea is to precisely represent the uncertainties in relative and absolute positioning, which are introduced by diversity, in the analysis, while still exploiting all relative information that is still available. Our analysis supports all artificial diversity approaches where diversification is achieved by reordering and relocation of code fragments. We have discussed that diversity has an impact on caching and its analysis, which is confirmed by our experimental results.

Our evaluation shows that our analysis delivers a safe upper bound for all versions, and that it is a major improvement over assuming *all miss* for the worst case, or not enabling the cache at all. In many cases, the estimations are even very close to the highest WCET that the non-diverse analysis may find. As we chose very small caches for being able to better observe the different effects, we can even expect better results for more common cache sizes.

In addition to systems using artificial software diversity, the analysis is also applicable to other areas, where code fragment positions are, at least partially, unavailable, such as dynamic libraries, or resources linked together from independent teams or vendors.

There are interesting aspects of the approach that deserve further attention. Applying the worst case of cache accesses per basic block is pessimistic considering that several successive basic blocks of the same fragment might not contain any conflicts. We will look into extending this scope to larger regions or sub-paths. We did also not investigate the actual structure of the code, with aspects of fragmented loops, sub-functions within loops and such. Note that we have also presented a WCET aware diversification approach in [14]. We plan to use our cache analysis to optimize fragmentation in this approach with respect to caching behavior. We also plan to consider other hardware features, such as certain kinds of branch prediction, multi-core, multi-level caches etc.

By enabling diversity in the instruction cache analysis of static WCET analysis, our approach delivers an important contribution to make artificial software diversity applicable to more systems. Thus, a critical group of CPS can be protected against code-reuse-attacks, making the systems they are controlling considerably more secure.

## References


1. C. Ballabriga and H. Casse. Improving the first-miss computation in set-associative instruction caches. In *2008 Euromicro Conference on Real-Time Systems*, pages 341–350, July 2008. doi:10.1109/ECRTS.2008.34.
2. Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.
3. Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.
4. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011. doi:10.1145/1966913.1966919.
5. Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the Fourth ACM Conference on Wireless Network Security*, WiSec '11, pages 127–138, New York, NY, USA, 2011. ACM. doi:10.1145/1998412.1998434.
6. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Computer and Communications Security*, pages 27–38, 2008. doi:10.1145/1455770.1455776.
7. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *ACM Conf. on Computer and Communications Security*, pages 559–572, 2010. doi:10.1145/1866307.1866370.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi:10.1145/512950.512973.
10. Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):40, 2013.
11. Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 299–310, 2013. doi:10.1145/2484313.2484351.
12. embedded.com EE Times. 2017 embedded market survey, 2017. URL: http://m.eet.com/media/1246048/2017-embedded-market-study.pdf.
13. Heiko Falk and Helena Kotthaus. Wcet-driven cache-aware code positioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 145–154, Taipei, Taiwan, oct 2011.
14. J. Fellmuth, P. Herber, T. F. Pfeffer, and S. Glesner. Securing real-time cyber-physical systems using wcet-aware artificial diversity. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 454–461, Nov 2017. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.88.

15 Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

16 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, 1999.

17 S. Forrest, A. Somayaji, and D.H. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, pages 67–72, 1997. doi:10.1109/HOTOS.1997.595185.

18 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, pages 137–147, 2010.

19 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 102–111. IEEE, 2012.

20 Jason Hiser, Anh Nguyen-Tuong, Michele Co, Mathew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *Security and Privacy*, pages 571–585, 2012.

21 Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 203–212. IEEE, 2011.

22 C. Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC*, 2006. doi:10.1109/ACSAC.2006.9.

23 P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014. doi:10.1109/SP.2014.25.

24 Y. T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *17th IEEE Real-Time Systems Symposium*, pages 254–263, Dec 1996. doi:10.1109/REAL.1996.563722.

25 P. Lokuciejewski, H. Falk, and P. Marwedel. Wcet-driven cache-based procedure positioning optimizations. In *2008 Euromicro Conference on Real-Time Systems*, pages 321–330, 2008. doi:10.1109/ECRTS.2008.20.

26 Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu, and Wang Yi. Mcait-a timing analyzer for multicore real-time software. In *ATVA*, pages 414–417. Springer, 2011.

27 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

28 Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

29 Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016. URL: https://hal.archives-ouvertes.fr/hal-01271552.

30 Frank Mueller, David B Whalley, and Marion Harmon. Predicting instruction cache behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

31 V. Pappas, M. Polychronakis, and A.D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP)*, pages 601–615, 2012. doi:10.1109/SP.2012.41.

32 PaX Team. PaX address space layout randomization (ASLR), 2003.

33 Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. Security and privacy challenges in industrial internet of things. In *Proceedings of the 52Nd Annual Design*

*Automation Conference*, DAC '15, pages 54:1–54:6, New York, NY, USA, 2015. ACM. doi:10.1145/2744769.2747942.

**34** Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561, 2007.

**35** L. Szekeres, M. Payer, Tao Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symp. on Security and Privacy (SP)*, pages 48–62, 2013. doi:10.1109/SP.2013.13.

**36** Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000. doi:10.1023/A:1008141130870.

**37** Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Computer and communications security*, pages 157–168, 2012.

**38** Yves Younan, Wouter Joosen, and Frank Piessens. Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs. *ACM Comput. Surv.*, 44:17:1–17:28, 2012. doi:10.1145/2187671.2187679.

**39** Zhenkai Zhang and Xenofon Koutsoukos. Improving the precision of abstract interpretation based cache persistence analysis. *SIGPLAN Not.*, 50(5):10:1–10:10, 2015. doi:10.1145/2808704.2754967.