Precise timing analysis for direct-mapped caches

Sidharta Andalam, Roopak Sinha, Partha Roop, Alain Girault, Jan Reineke

Precise Timing Analysis for Direct-Mapped Caches

Sidharta Andalam
TUM CREATE, Singapore
sidharta.andalam@tum-create.edu.sg

Roopak Sinha, Partha Roop
University of Auckland, New Zealand
{r.sinha,p.roop}@auckland.ac.nz

Alain Girault
INRIA - Grenoble, France
alain.girault@inria.fr

Jan Reineke
Saarland University- Saarbrücken, Germany
reineke@cs.uni-saarland.de

ABSTRACT

Safety-critical systems require guarantees on their worst-case execution times. This requires modelling of speculative hardware features such as caches that are tailored to improve the average-case performance, while ignoring the worst case, which complicates the Worst Case Execution Time (WCET) analysis problem. Existing approaches that precisely compute WCET suffer from state-space explosion. In this paper, we present a novel cache analysis technique for direct-mapped instruction caches with the same precision as the most precise techniques, while improving analysis time by up to 240 times. This improvement is achieved by analysing individual control points separately, and carrying out optimisations that are not possible with existing techniques.

Categories and Subject Descriptors: B.3.3 [Performance Analysis and Design Aids]: Worst-case analysis
General Terms: Verification, Algorithms
Keywords: Instruction, Direct-Mapped, Cache Analysis.

1. INTRODUCTION

Hard real-time systems require accurate guarantees on the functionality as well as the timing characteristics of programs. Traditional speculative architectural features such as multi-level caches and deep pipelines render the worst-case execution. Two types of memory architectures are used in real-time systems: specialized compiler assisted caches, called *scratch-pads* [2], and (widely available) conventional caches [3, 9, 11]. This article focuses on the static analysis [12] for predictable direct-mapped instruction caches [9, 11], where locations in main memory are mapped to unique cache lines.

Cache analysis involves computing the number of cache misses that can happen in the instruction cache at specific control points in a program. The program is usually translated to a control flow graph (CFG) [9], which contains control points as its nodes. Analytically, the cache analysis problem boils down to statically determining all possible cache states (and therefore the number of misses in the worst case) at each node using a suitable fixed point computation.

While a number of cache analysis approaches exist [11, 9], some are not scalable (but more precise) while others overestimate cache misses (but are more scalable). In *concrete* techniques like [9], all possible cache states are enumerated explicitly at each control point, and very precise results can be obtained. However, these techniques suffer from state-explosion, and do not scale for large programs. In [8], a probabilistic approach for modelling cache behaviour is presented, which is used for design-space exploration to reduce overall analysis time by exploiting the structural similarities among related cache configurations. In [7], the idea of cache conflict graphs is introduced where cache lines are analysed one at a time. This approach is more scalable than concrete approaches, but loses precision as any relations between cache lines are abstracted out. On the other hand, *abstract* techniques like [10, 11] collapse multiple possible cache states at every control point into a single abstract cache state. This abstraction allows the algorithm to reach a fixed point much faster, and hence larger programs can be analysed, but at the cost of sacrificing precision.

In this article, we present a novel cache analysis technique for direct-mapped caches that maintains the same precision as the concrete techniques while significantly improving scalability (for large benchmarks, analysis time is less than 2 minutes, instead of 12 hours as in [9]). This improvement is achieved by analysing individual control points at once, capturing and aggregating instructions in a *relative* manner, and carrying out certain optimisations that are not possible in other techniques. The key contribution of this paper is a new algorithm for static analysis that offers the same precision as the most precise algorithms while being extremely scalable in comparison. It also compares very favourably with abstraction-based analysis techniques with respect to analysis time.

This paper is organised as follows. The cache analysis problem is formalized in Sec. 2. In Sec. 3 we present the proposed approach. Qualitative and quantitative comparison with the concrete and abstract approaches is presented in Sections 4 and 5 respectively. Finally, conclusions are presented in Sec. 6. The Appendix further illustrates key aspects of the proposed algorithm with additional experimental results to demonstrate its advantages. Also, in Appendix E, we discuss how the proposed approach for direct-mapped caches can be extended to set-associative caches, which are also widely used in predictable systems [3].

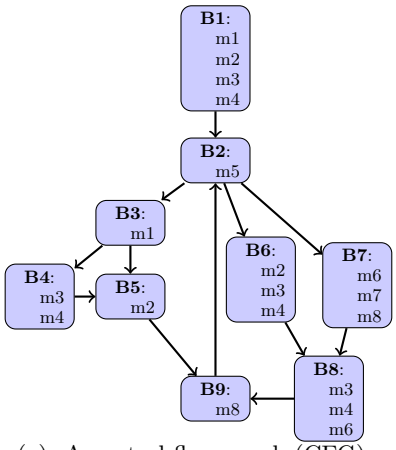
2. THE CACHE ANALYSIS PROBLEM

Our analytical *cache model* is based on the model of [9], and is defined below. To improve readability, we represent the terms "basic blocks" and "memory blocks" as "blocks" and "instructions", respectively.

DEFINITION 1 (CACHE MODEL). *The cache model for a given program is defined as a tuple* $CM = \langle I, C, CI, G, BI \rangle$, *where* I *is a finite set of instructions in the program,* $C = \{c_0, c_1, \ldots, c_{N-1}\}$ *is an ordered set of cache lines with* $N = |C|$ *as the total number of cache lines, and* $CI : C \rightarrow 2^I$ *is the direct-mapping function.*

Also, G is a directed graph $G = \langle B, b_{init}, E\rangle$ where B is a finite set of basic blocks with $b_{init} \in B$ as the initial block, and $E \subseteq B \times B$ is the set of edges.

Finally, $BI : B \to (I \cup \{\mp\})^N$ is the block to instruction mapping function, where $\mp$ represents no instruction.



(a) A control flow graph (CFG)

Cache line	Inst.
c_0	m1,m5
c_1	m2,m6
c_2	m3,m7
c_3	m4,m8

(b) Mapping of instructions to cache lines

Figure 1: An example cache model

Figure 1 shows the cache model of a sample program. It presents a CFG in Figure 1(a), which contains 9 blocks ($B = \{B1, \ldots, B9\}$) with $B1$ as the initial block. Each block executes one or more instructions. The CFG has 8 instructions ($I = \{m1, \ldots, m8\}$) that are mapped to 4 cache lines ($C = \{c_0, c_1, c_2, c_3\}$), as presented in Figure 1(b). Each cache line has two unique instructions mapped to it (e.g., $CI(c_0) = \{m1, m5\}$). An instruction can be mapped only to a unique cache line, representing a direct-mapped cache. A cache line, at any time, can contain only one of the instructions that are mapped to it (or is empty).

The edges describe program flow (sequential and branching) between the control points of the given program. We use the short-hand $B1 \to B2$ to describe a CFG edges such as $(B1, B2) \in E$.. Each block also contains instructions (from I) that it executes using the mapping function BI. For example, $B1$ contains four instructions $m1$, $m2$, $m3$ and $m4$. We restrict the model such that blocks can contain at most one instruction mapped to any cache line (as per CI), which allows us to represent the block to instruction mapping using a vector indexed by C. E.g., $BI(B1) = [m1, m2, m3, m4]$, with $BI(B1)[0] = m1$ describing the instruction mapped to cache line c_0. If a block b does not contain an instruction mapped to a cache line c_i, then $BI(b)[i] = \mp$ (e.g., $BI(B2)[1] = \mp$).

The contents of the cache at any time during the program execution is called a *cache state*, and is represented as a vector $cs = [inst_0, \ldots, inst_{N-1}]$ where each $inst_i$ represents the instruction contained in cache line $c_i \in C$. When execution begins, we assume that there is no instruction (represented by $\top$) in each cache line. The cache is assumed to be empty. For the example presented of Fig. 1, the empty cache state is represented by $cs_\top = [\top, \top, \top, \top]$. During execution, or traversal of the CFG, instructions are loaded into the cache as the basic blocks are executed (starting from the initial block). E.g., after executing the instructions in $B1$ the cache state is $cs_1 = [m1, m2, m3, m4]$. In this example, $cs_\top$ is a *reaching cache state* of $B1$, while cs_1 is a *leaving cache state* of $B1$. Since all 4 instructions needed by $B1$ were not present in $cs_\top$, we say that there were 4 cache *misses*.

Given any reaching cache state cs of a block b, we can compute the number of cache misses by comparing the instructions in cs and $BI(b)$. E.g., given a reaching cache state $cs = [m1, m2, m3, m4]$ of block $B2$ (with $BI(B2) = [m5, \mp, \mp, \mp]$), there is a single miss on cache line c_0 because the instruction $m5$ needed by the block (as per $BI(b)[0] = m5$) is not present in the cache state ($cs[0] = m1$).

It is generally possible for a block to have multiple reaching cache state. Here, we compute the *worst-case* miss count wmc_b as the maximum number of cache misses possible, as defined below.

Definition 2 (The Cache Analysis Problem). *Given a cache model CM, the cache analysis problem is the computation of the number of worst case cache misses (wmc_b), for all basic blocks $b \in B$, along all possible executions.*

3. PROPOSED APPROACH

Our approach for the static analysis of direct-mapped caches is based on the intuition that analysing a single basic block b_{ref} of the CFG at a time allows us to (a) reduce the number of blocks needed to compute the worst and best case miss counts for b_{ref}, and (more importantly), (b) we can abstract cache states computed during the fixed point algorithm w.r.t. the instructions executed by b_{ref}. This may significantly reduce the number of possible cache states and consequently reduce analysis time. We use Fig. 1 to illustrate these benefits.

First principle: During the analysis of block $b_{ref} = B8$, since $B8$ does not execute any instruction on cache line c_0 ($BI(B8)[0] = \mp$), we can ignore block $B2$ as the execution of B2 can only affect the cache line c_0. We call $B2$ a *vacuous* block because it does not interfere with any of the cache lines used by $B8$, and it can be removed when analysing $B8$.

Second principle: During the analysis of block $b_{ref} = B8$, the instructions contained in another block, say $B1$, can be abstracted such that they only refer to their *effect* on the analysis of $B8$. Given that $BI(B8) = [\mp, m6, m3, m4]$ and $BI(B1) = [m1, m2, m3, m4]$, the instructions in $B1$ can be abstracted as the vector $[\times, 1, 0, 0]$. Here, the first element '$\times$' means that the instruction is *not of interest* as the reference block does not use this cache line ($BI(B8)[0] = \mp$). The second element '1' means that for cache line c_1, the instruction in $B1$ is *different* from the instruction in $B8$ ($BI(B1)[1] = m2 \neq m6 = BI(B8)[1]$). Finally, for the third and the fourth elements '0' means that for cache line c_2, the instruction in $B1$ is the *same* as the instruction in $B8$ ($BI(B8)[2] = m3 = BI(B1)[2]$). Also, when there is *no instruction* on cache line c_i in a block the abstract representation is '$\mp$'. The ability of reducing the number of instructions ($|I|$) in the CFG to just four *relative instructions* ($\times, \mp, 0, 1$) significantly reduces the memory foot print and analysis time, without sacrificing precision. This is a key optimization that enables us to propose a scalable analysis technique without sacrificing precision.

Alg. 1 presents an overview of our approach. Each block in the CFG is analysed individually (described using the for-loop on lines 1–5). For each reference block b_{ref} in B, on line 2, we first reduce the CFG by removing the vacuous blocks and then compute the relative instructions w.r.t. b_{ref}. Next, on line 3, a fixed point algorithm is used to compute all possible cache states of the reference block. Finally, on line 4, the number of cache misses in the worst case are computed. We now present the details of each of these steps.

Algorithm 1 Overview of the proposed approach

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$.
Output: Compute the worst miss count (wmc) for all basic blocks.

1: **for each** $b_{ref} \in B$ **do**
2: {Reduced CFG, compute relative instructions (Sec. 3.1)} $(G^r, BI^r) = Reduce(CM, b_{ref})$
3: {Compute reaching relative cache states (Section 3.2.3)} $RCS^r_{b_{ref}} = FP(CM, G^r, BI^r)$
4: {Compute cache misses in the worst-case (Section 3.3)} $wmc_{b_{ref}} = MAXmc(RCS^r_{b_{ref}})$
5: **end for**
6: **return** wmc_b for all blocks b in B {Solution for the cache analysis problem (Definition 2)}

3.1 CFG Reduction

Alg. 2 presents the pseudocode of the *Reduce* algorithm that returns a reduced graph G^r from a given CFG G. Given G and a reference block b_{ref}, we first represent the instructions in each block w.r.t. the reference block, and then remove any vacuous blocks. Line 1 initializes G^r as a copy of G. Then, for each block b in G^r, and each cache line c_i, a relative block to instructions mapping BI^r is created (lines 2–7). Depending on whether the instruction originally contained in b (in G) is of no-interest to the reference block (line 3), different (line 4) or identical (line 5) to the instruction contained in the reference blocks, or is equal to $\mp$ (line 6).

Algorithm 2 *Reduce*: Reduce the CFG and abstract inst.

Input: Cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$ and a reference block $b_{ref} \in B$.
Output: $G^r = \langle B^r, b_{init}, E^r\rangle$ and $BI^r : B^r \rightarrow (\{\times, \mp, 1, 0\})^N$
1: Initialize G^r as a copy of G with $BI^r(b) = \emptyset$ for all $b \in B^r$.
2: **for each** $b \in B^r$ and for each $c_i \in C$ **do**
3: $BI^r(b)[i] = \times$, if $(BI(b_{ref})[i] = \mp)$ {not of interest}
4: $BI^r(b)[i] = 1$, if $(BI(b_{ref})[i] \neq BI(b)[i])$ {different}
5: $BI^r(b)[i] = 0$, if $(BI(b_{ref})[i] = BI(b)[i])$ {identical}
6: $BI^r(b)[i] = \mp$, if $(BI^r(b)[i] \neq \times \wedge BI(b)[i] = \mp)$ {no inst.}
7: **end for**
8: **for each** $b \in B^r$ **do**
9: **if** $(BI(b) \in (\{\times, \mp\})^N) \wedge (b \neq b_{init})$ {check for all vacuous blocks, excluding initial block} **then**
10: Remove b from B^r, and adjust E^r
11: **end if**
12: **end for**

Next, on lines 8–11, blocks for which the mapping function BI^r returns $\times$ or $\mp$ for every element of the vector $BI^r(b)$ are declared as vacuous and are removed from the graph. The removal of a vacuous block b_r involves adjusting the edges E^r of the graph G^r such that each predecessor of b_r now has a direct edge to each of the successors of b_r. More details and the full *Reduce* Algorithm appear in Appendix A.

Fig. 2 shows the reduced CFG returned by the algorithm *Reduce* when $b_{ref} = B8$. Note that the vacuous block $B2$ is removed from G^r. Also, every block now contains the relative instructions.

3.2 Fixed point Computation

3.2.1 Relative cache states

Since block to instructions mapping is described using relative instructions, we also compute cache states in a relative fashion. A *relative cache state* is defined as follows.

Definition 3 (Relative cache state). *A relative cache state cs^r is a vector $[inst^r_0, \ldots, inst^r_{N-1}]$, where each element $inst^r_i \in \{1, 0, \top, \bot, \times\}$. The set of all possible relative cache states is denoted as CS^r.*

Each relative instruction $inst^r_i$ of a relative cache state $cs^r = [inst^r_0, \ldots, inst^r_{N-1}]$ is described w.r.t. the instruction $(BI(b_{ref})[i]$, $i \in [0, |C|])$ in the reference block b_{ref} for the cache line $c_i \in C$. $inst^r_i = 1$ or $inst^r_i = 0$ means the instruction in the cache is *different* or *identical* respectively to the instruction executed in the reference block b_{ref}. E.g., given $BI(b_{ref})[i] = m1$, $inst^r_i = 1$ or $inst^r_i = 0$ means the instruction on cache c_i is not $m1$ or $m1$ respectively. $inst^r_i = \top$ means that cache line c_i is *empty*, whereas $inst^r_i = \bot$ means that the cache has an *unknown* instruction. Finally, $inst^r_i = \times$ means that the instruction on this cache line is *not of interest* during the analysis of b_{ref}. Also, a relative cache state before executing block b is know as a *reaching relative cache state* of b. Similarly, a relative cache state after executing block b is know as a *leaving relative cache state* of b. More details about relative cache states are presented in Appendix B.

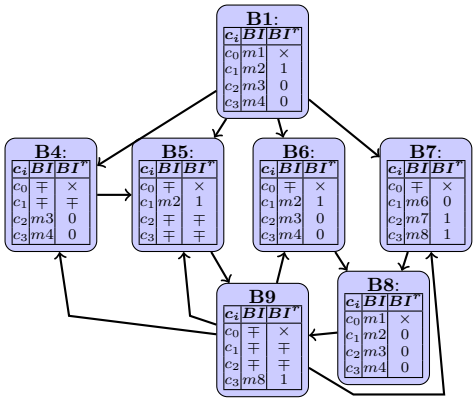
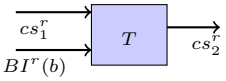


Figure 2: The reduced graph G^r obtained from the *Reduce* algorithm, with the reference block $b_{ref} = B8$.

3.2.2 The Transfer Function

An important operation in the fixed point computation is the transformation of a reaching relative cache state into a leaving relative cache state. This transformation, called the *transfer* function $T : CS^r \times B \rightarrow CS^r$, is illustrated as follows.



$$\begin{array}{rl} cs^r_1 & =[\times, 1, 0, 1] \\ BI^r(b) & =[\times, 0, \mp, 1] \\ \hline T(cs^r_1, b) = cs^r_2 & =[\times, 0, 0, 1] \end{array}$$

For any cache line c_i, the instruction $cs^r_2[i]$ in the leaving relative cache state is equal to $BI^r(b)[i]$ only if there is an instruction in block b ($BI^r(b)[i] = 1$ or $BI^r(b)[i] = 0$). Otherwise, the instruction $cs^r_2[i]$ is the same as the instruction $cs^r_1[i]$ in the reaching cache state. For the above example, given $BI^r(b) = [\times, 0, \mp, 1]$ and $cs^r_1 = [\times, 1, 0, 1]$, after execution of block b, $cs^r_2 = [\times, 0, 0, 1]$. For cache lines c_1 and c_3, block b executes instructions that relate to the reference block, and for cache lines c_0 and c_2, its does not execute any relevant instructions. Therefore, the contents of the leaving cache states are updated to be the same as the instructions executed by the block on cache lines c_1 and c_3, and remain the same as the reaching cache state for cache lines c_0 and c_2.

3.2.3 Fixed point computation

Alg. 3 shows the fixed point algorithm FP used to compute all possible reaching relative cache states for the referenced block in the reduced graph G^r. We illustrate the fixed point computation using Tab. 1, which shows the possible reaching and leaving relative cache states of each block in the reduced graph (Fig. 2) in every iteration of FP.

During initialization, the reaching cache state of the initial block is set to $cs^r_\top$ (line 2) because the cache is considered

empty initially. For every other block, on line 2, the initial reaching cache state is unknown ($cs^r_\perp$). As shown in Tab. 1, The initial reaching relative cache state of the initial block $B1$ is set to $cs^r_\top = \{[\times, \top, \top, \top]\}$, while for every other block, the initial reaching relative cache state is $cs^r_\perp = \{[\times, \perp, \perp, \perp]\}$ (iteration 1, column 3, Tab. 1).

Algorithm 3 FP: Fixed point computation

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$, reduced graph $G^r = \langle B^r, b_{init}, E^r\rangle$ and $BI^r : B^r \rightarrow (\{\times, \mp, 1, 0\})^N$.
Output: Reaching relative cache states of block b_{ref} ($RCS^r_{b_{ref}}$).

1: Create initial cache state $cs^r_\top$ where for every $c_i \in C$, $cs^r_\top[i] = \times$ if $BI^r(b_{ref}) = \times$, or $cs^r_\top[i] = \top$ otherwise.
2: $RCS^{r^1}_{b_{init}} = \{cs^r_\top\}$, and $RCS^{r^1}_b = \{cs^r_\perp\}$ for all other $b \in B^r$.
3: $i = 1$
4: **repeat**
5: **for each** $b \in B^r$ **do**
6: $LCS^{r^i}_b = T(RCS^{r^i}_b, b)$
7: **end for**
8: $i = i + 1$; {Next iteration}
9: **for each** $b \in B^r$ **do**
10: **if** $b = b_{init}$ **then**
11: $RCS^{r^i}_{b_{init}} = \{cs^r_\top\} \cup (\bigcup LCS^{r^i}_{b'} \mid (b', b) \in E^r)$
12: **else**
13: $RCS^{r^i}_b = \bigcup LCS^{r^i}_{b'} \mid (b', b) \in E^r$
14: **end if**
15: **end for**
16: **until** $\forall b \in B^r$, $RCS^{r^i}_b = RCS^{r^{i-1}}_b$ {Termination condition}
17: **return** $RCS^r_{b_{ref}}$

Table 1: Computing all possible reaching relative cache states of the reference block b_{ref} (B8).

Itr. (i)	Block (b)	Reaching Relative Cache States (RCS^i_b)	Leaving Relative Cache States (LCS^i_b)
1	B1	{[×, ⊤, ⊤, ⊤]}	{[×, 1, 0, 0]}
	B4	{[×, ⊥, ⊥, ⊥]}	{[×, ⊥, 0, 0]}
	B5	{[×, ⊥, ⊥, ⊥]}	{[×, 1, ⊥, ⊥]}
	B6	{[×, ⊥, ⊥, ⊥]}	{[×, 1, 0, 0]}
	B7	{[×, ⊥, ⊥, ⊥]}	{[×, 0, 1, 1]}
	B8	{[×, ⊥, ⊥, ⊥]}	{[×, 0, 0, 0]}
	B9	{[×, ⊥, ⊥, ⊥]}	{[×, ⊥, ⊥, 1]}
2	B1	{[×, ⊤, ⊤, ⊤]}	{[×, 1, 0, 0]}
	B4	{[×, 1, 0, 0], [×, ⊥, ⊥, 1]}	{[×, 1, 0, 0], [×, ⊥, 0, 0]}
	B5	{[×, 1, 0, 0], [×, ⊥, ⊥, 1]}	{[×, 1, 0, 0], [×, 1, ⊥, 1]}
	B6	{[×, 1, 0, 0], [×, ⊥, ⊥, 1]}	{[×, 1, 0, 0]}
	B7	{[×, 1, 0, 0], [×, ⊥, ⊥, 0]}	{[×, 0, 1, 1]}
	B8	{[×, 1, 0, 0], [×, 0, 1, 1]}	{[×, 0, 0, 0]}
	B9	{[×, 1, ⊥, ⊥], [×, 0, 0, 0]}	{[×, 1, ⊥, 1], [×, 0, 0, 1]}
3 4	... ...	... ...	... ...
5	B1	{[×, ⊤, ⊤, ⊤]}	
	B4	{[×, 1, 0, 0], [×, 1, 0, 1], [×, 0, 0, 1]}	
	B5	{[×, 1, 0, 0], [×, 1, 0, 1], [×, 0, 0, 1], [×, 0, 0, 0]}	
	B6	{[×, 1, 0, 0], [×, 1, 0, 1], [×, 0, 0, 1]}	
	B7	{[×, 1, 0, 0], [×, 1, 0, 1], [×, 0, 0, 1]}	
	B8	**{[×, 1, 0, 0], [×, 0, 1, 1]}**	
	B9	{[×, 1, 0, 0], [×, 0, 0, 0], [×, 1, 0, 1]}	

The repeat-until loop (lines 4–16) is the fixed point iteration. In each iteration i, each reaching relative cache state contained in the set $RCS^{r^i}_b$ for every block b is transformed into a leaving relative cache state (in set $LCS^{r^i}_b$) by applying the transfer function (lines 5–7). E.g., for block $B1$ in iteration 1, given the only reaching relative cache state $[\times, \top, \top, \top]$, the corresponding leaving relative cache state is $T([\times, \top, \top, \top], B1) = \{[\times, 1, 0, 0]\}$ (see Tab. 1).

Then we compute, the reaching relative cache states of each block for the next iteration. For each block, the set of reaching relative cache states is the union of the sets of the leaving relative cache states of all of its predecessors (line 13). For b_{init}, the additional reaching cache state $cs^r_\top$ is also added to this set (line 11). E.g., the predecessors of $B4$ are $B1$ and $B9$ (see Fig. 2). Hence, their sets of leaving cache states (resp. $\{[\times, 1, 0, 0]\}$ and $\{[\times, \perp, \perp, 1]\}$) for iteration 1 are aggregated together to form the reaching cache state of $B4$ in iteration 2.

The iterations continue until the fixed point is reached, i.e., when two consecutive iterations yield the same sets of reaching relative cache states for all blocks (line 35). For the reduced CFG shown in Fig. 2, the fixed point is reached in the 5^{th} iteration. Also during the fixed point, as an optimisation, a relative cache state cs^r_k is dropped if there exists cs^r_j such that, if for all cache lines (c_i), when the relative instruction in cs^r_j is the same as the instruction in cs^r_k ($cs^r_j[i] = cs^r_k[i]$) or, the relative instruction in cs^r_j captures a cache miss ($cs^r_j[i] = 1$). E.g., given four possible reaching cache states $\{[0, 0, 0, 1], [0, 1, 0, 1], [0, 1, 1, 1], [1, 0, 0, 1]\}$, we can safely ignore the first two cache states, because the third state captures the worst case behaviour. However, we must still carry the last state, resulting in the reduced set $\{[0, 1, 1, 1], [1, 0, 0, 1]\}$.

3.3 Computing the number of cache misses

The final step in the cache analysis algorithm is the computation of the number of cache misses in the worst case. This is done by analysing the relative cache states of the reference block b_{ref} as computed by the fixed point algorithm. For each reaching cache state $cs^r = [inst^r_1, \ldots, inst^r_{N-1}]$, and for every cache line $c_i \in C$, $inst_i = 1$ represents a cache miss on c_i. The total number of misses when the reaching cache state cs^r is the number of 1's contained in cs^r. The reaching cache states with the highest number of 1's correspond to the worst-case miss counts of b_{ref} respectively. E.g., as shown in Tab. 1, the two reaching cache states of the reference block $B8$ as computed by the fixed point algorithm FB are $\{[\times, 1, 0, 0]$ and $[\times, 0, 1, 1]\}$. The first reaching cache state has one occurrence of '1' while the second one has two occurrences of '1'. Thus, the maximum miss count for $B8$ is 2, i.e., $wmc_{B8} = 2$.

4. QUALITATIVE COMPARISON

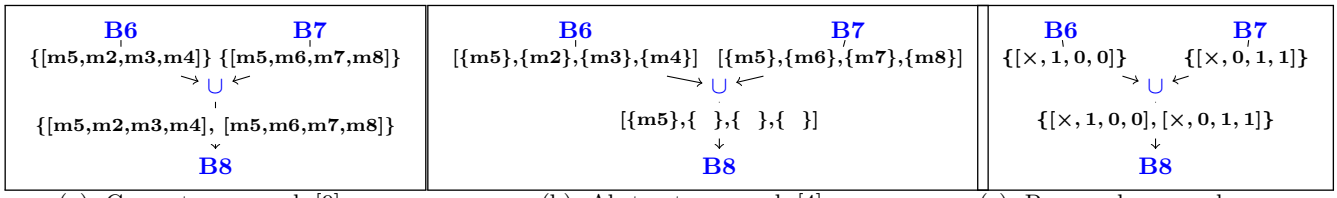
Tab. 2 provides a qualitative comparison between the concrete [9], abstract [4], and the proposed approaches. Figure 3 illustrates the basic block $B8$ of the CFG shown in Fig. 2 with its two predecessors $B6$ and $B7$. We use this example to show the differences in the ways the three approaches represent and aggregate cache states.

Table 2: Qualitative comparison of the three approaches.

App.	Fixed point	Precision	Time	Optimisation	Max no. of cs at each program point
Conc.	all blocks	high	slow	none	$(I/N)^N$
Abs.	all blocks	low	fast	merge cache states	constant
Pro.	one block at a time	high	med.	(1) reduce graph (2) reduce cache lines (3) relative instructions	$(3)^N$

In the **concrete** approach [9], a cache state (cs) is described as a vector $[inst_0, \ldots, inst_{N-1}]$ where each $inst_i$ represents a single instruction contained in cache line $c_i \in C$ or is empty ($\top$), i.e., $inst_i \in CI(c_i) \cup \{\top\}$. E.g., the sets of leaving cache states for $B6$ and $B7$ are $\{[m5, m2, m3, m4]\}$ and $\{[m5, m6, m7, m8]\}$ respectively (see Fig. 3(a)). The set of reaching cache states of $B8$ is the *union* of these sets, as shown in Fig. 3(a). This set represents the fact there are only two states in which the cache can be before $B8$ is executed.

In the **abstract** approach [4], a cache state is described as a vector $[set_0, \ldots, set_{N-1}]$ where each set_i represents a set of instructions that must be contained in cache line $c_i \in C$. That is, $set_i \subseteq CI(c_i) \cup \{\top\}$.. E.g., the abstract leaving cache states for $B6$ and $B7$ are $[\{m5\}, \{m2\}, \{m3\}, \{m4\}]$ and $[\{m5\}, \{m6\}, \{m7\},\ \{m8\}]$ respectively (see Fig. 3(b)). For



(a) Concrete approach [9] (b) Abstract approach [4] (c) Proposed approach

Figure 3: Illustration of the combining cache states.

each of these abstract cache states, each cache line contains precisely one instruction. Hence, these two abstract cache states are equivalent to the concrete cache states $[m5, m2, m3, m4]$ and $[m5, m6, m7, m8]$ shown in Fig. 3(a). Next, the abstract reaching cache state of $B8$ is the *pair-wise intersection* over the vector elements of the above abstract cache states, giving $[\{m5\}, \{\ \}, \{\ \}, \{\ \}]$ (see Fig. 3(b)). A cache line c_i, with an empty set represents *any instruction* (that is mapped to c_i, $CI(c_i)$) or *no instruction*. E.g., the empty set on cache line c_1, represents three possible states $\{m2, m6, \top\}$. Similarly, $[\{m5\}, \{\ \}, \{\ \}, \{\ \}]$ represents $[\{m5\}, \{m2, m6, \top\}, \{m3, m7, \top\}, \{m4, m8, \top\}]$. This abstract reaching cache state represents 27 possible concrete cache states, which are computed using cross product as:

$$\begin{aligned}&\{m5\} \times \{m2, m6, \top\} \times \{m3, m7, \top\} \times \{m4, m8, \top\}\\ =&\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m2, m7, m4],\\ &[m5, m2, m7, m8], ...\}\end{aligned}$$

While two of these possible concrete states $[m5, m2, m3, m4]$ and $[m5, m6, m7, m8]$ are valid (as discussed above), many others, such as $[m5, m6, m3, m4]$, are not reachable because instructions $m6$ and $m3$ cannot be loaded into the cache by any one of the predecessor blocks of $B8$. As the above example shows, the **abstract** approach may lose precision due to joins, which may introduce non-reachable cache states. Yet, the fixed point algorithm converges faster for the same reason.

Finally, in the **proposed approach**, cache states are relative to the instructions of the reference block. As shown in Fig. 3(c), the reaching relative cache states of $B8$ are computed by taking a *union* of the predecessor's leaving relative cache states, $\{[\times, 1, 0, 0]\}, \{[\times, 0, 1, 1]\}\}$, as shown in Fig. 3(c). The relative cache state representation allows us to optimize performance without sacrificing precision. E.g., as shown in Fig. 3(c), one of the reaching relative cache state of $B8$ is $\{[\times, 1, 0, 0]\}$. It represents 6 possible concrete cache states, which are computed using cross product as:

$$\begin{aligned}\{[\times, 1, 0, 0]\} =&\{\top, m1, m5\} \times \{\top, m2\} \times \{m3\} \times \{m4\}\\ =&\{[\top, \top, m3, m4], [\top, m2, m3, m4], [m1, \top, m3, m4],\\ &[m1, m2, m3, m4], [m5, \top, m3, m4], [m5, m2, m3, m4]\}\end{aligned}$$

Thus, the translation results in 6 concrete cache states. However, the extra states, unlike in the abstract approach, do not effect the precision of the analysis.

Table 3: Comparing the cache states and WCET estimates between the three approaches as we analyse block $B8$.

App.	cache state	wmc_{B8} (Worst)
Concrete	$\{[m5, m2, m3, m4], [m5, m6, m3, m4]\}$	2
Abstract	$[\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m4, m8\}]$	3
Proposed	$\{[\times 100], [\times 011]\}$	2

Comparison of the precision: By analysing the reaching cache states of $B8$, we can compute wmc_{B8} and compare the precision among the three approaches. We present this comparison using Table 3. For block $B8$ ($BI(B8) = [\mp, m6, m3, m4]$), given the set of possible reaching cache states for each approach (in column 2), the WCET estimate (wmc) is presented in column 3. We note that the **concrete** approach has higher precision (smaller WCET), compared the **abstract** approach. Also, the **proposed** approach maintains this high precision. The optimised use of relative cache states is only possible in the **proposed** approach because it analyses one block at once. Both the concrete and abstract approaches analyse all blocks in the CFG together.

Comparison of the complexity: The time complexity of each of the three approaches depends on the number of blocks in the CFG, the complexity of the merging and equivalence operations, and the maximum number of cache states that can be created (or the height of the lattice). The number of blocks of the CFG is fixed at $|B|$, while we assume that the merging and equivalence operations between cache states can be performed in $O(N)$ times (N is the number of cache lines). Therefore, the complexities are of the form $O(|B| \times N \times HeightOfLattice)$. To compute the height of the lattice, we assume that each cache line has an equal share $\lceil |I|/N \rceil$ (I is the set of instructions) of instructions mapped to it, and that $|I|$ and N are sufficiently large so that we can ignore the presence of $\bot$ and $\top$ in the computed cache states.

For the **concrete** approach, there are $(\lceil |I|/N \rceil)^N$ possible reaching cache states. Hence the complexity of the approach is $O(|B| \times N \times (\lceil |I|/N \rceil)^N)$. For the **abstract** approach, because of the point-wise intersection operation used to merge cache states for every cache line, the height of the lattice is constant at 1 for a single cache line, or N for all cache lines. The complexity of this approach is therefore $O(|B| \times N^2)$. Finally, for the **proposed** approach, each cache line has precisely 5 relative instructions $\{\times, 0, 1, \bot, \top\}$ mapped to it. However, recall that if a cache line has $\times$ assigned to it, it means the cache line is not used (and hence cannot contain any other instruction during the analysis). Similarly, $\top$ is used only for the initial node. The number of cache states possible (over N cache lines) in our approach is therefore 3^N. The complexity of our approach is therefore $O(|B| \times N \times 3^N)$ for *one* call to the fixed point algorithm. Given that we repeat this process $|B|$ times (once for every block), and that the CFG reduction operation is linear to the size of the CFG (again, $|B|$), the complexity of our approach is therefore $O(|B|^3 \times N \times 3^N)$.

As can be seen above, the complexity of the abstract approach is significantly lower than the concrete approach as well as our approach. Also, the complexity of the proposed approach does not depend on the number of instructions in the program, which helps us achieve much better performance than the concrete approach, as discussed in the next section.

5. BENCHMARKING AND RESULTS

We compare the precision and analysis time of the concrete, abstract, and proposed approaches over a set of benchmarks consisting of five control applications from [6]. In addition, we created two examples: a BubbleSort program and a Synthetic example. The benchmarking results are presented in Tab. 4. The first column presents the examples followed by their description (column 2). The number of C lines of the program and its memory footprint is presented in columns 3 and 4 respectively. The largest examples are CruiseController

and RailRoadCrossing, with more than 4000 lines of code.

Each program is compiled to execute on the MicroBlaze (MB) processor [1]. We choose MB due to the availability of timing analysis tools [5]. The CFG for each example is extracted automatically from its compiled binary, and each loop of the CFG is unrolled once for more precise cache analysis [4]. For MicroBlaze with 64 MB main memory, the size of the cache can be configured from 128 bytes to 64 KB [1]. Using these proportions, in our experiments we explore cache sizes between 0.1% and 1% of the program's size.

Table 4: Benchmark programs and their characteristics.

Example		Description	LOC	Size
BubbleSort	(BS)	Bubble sort algorithm	128	2KB
Synthetic	(SY)	Branching and loops	180	4KB
Flasher	(FL)	Distributed lights	384	9KB
DrillStation	(DS)	Drilling station	1800	62KB
ConvBelt	(CB)	Airport conveyor belt	1280	44KB
RailRoadCros.	(RR)	Rail road cnt.	4613	163KB
CruiseCntroler	(CC)	Cruise control model	4194	146KB

For the first two examples (BS, SY), the WCET estimates from the proposed approach was identical to that of the concrete approach (see Table 5 in Appendix D). However, for the rest of the examples, the concrete approach failed to terminate (analysis time is more than 12 hours, represented using "T.O" in Table 5). Thus, we only focus on comparing between the proposed and the abstract approaches.

For a cache size of 1% of the program size, normalised WCET estimates w.r.t. the results from the abstract approach are presented in Figure 4(a). Across all the benchmarks, we observe that the WCET computed by the proposed approach is always less than or equal to the estimates from the abstract approach. On average, the WCET estimate from the proposed approach is 15% smaller than the abstract approach. For a 0.1% relative cache size, the WCET analysis results are presented in Figure 4(b). Here, the proposed approach does not gain extra precision (w.r.t. the abstract approach), because the cache size is too small.

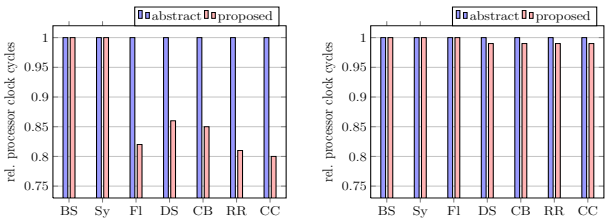
In terms of analysis time, the proposed approach always takes less than 3 minutes for each example, which is significantly faster than the concrete approach. However, the abstract approach is even much faster (always takes less than 4 seconds) than the proposed approach. For the largest example (RR), the proposed approach takes 142 seconds compared to the 4 seconds taken by the abstract approach, but the WCET estimate is tighter by 19% ($1 - 250725/308505$).

Finally, Fig. 5 shows the WCET vs analysis time for the control applications (last 5 benchmarks). Since the concrete approach failed to terminate, its WCET is represented by the WCET of the proposed approach (since both yield identical results). On average, compared to the abstract approach, the WCET from the proposed approach is 16% tighter. For the analysis time, the proposed approach always completes in less than 3 minutes, compared to the timeout after 12 hours for the concrete approach.

In Appendix D, we explore eight other cache sizes between 0.2% and 0.9%. For the RailRoadCrossing example, on average across the eight cache sizes, the WCET of proposed approach gives 16 % much tighter results on average, and up to 19 % tighter result than the abstract approach.

6. CONCLUSIONS

We proposed a new cache analysis approach for precise analysis of instructions in direct-mapped caches. The proposed approach presents a new abstraction, and compared to the concrete approach it significantly reduces the analysis time without sacrificing the precision. This is unlike the concrete



(a) 1% relative cache size (b) 0.1% relative cache size

Figure 4: Comparing the WCET estimates (the smaller the better) of the abstract and proposed approaches

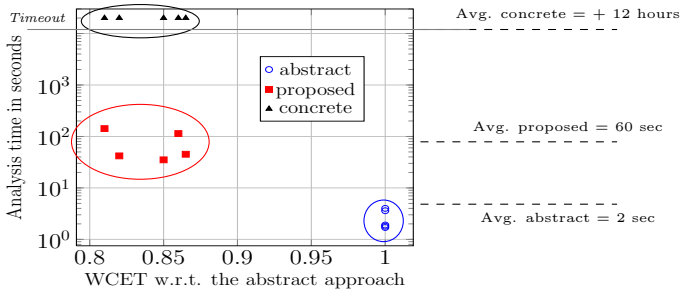


Figure 5: Comparing WCET and analysis time for the last five examples (1% relative cache size)

or the abstract approaches, where scalability or precision must be sacrificed. Overall, the proposed approach enables precise and efficient WCET analysis even for larger programs. In the future, we will be extending our approach for analysing set associative caches and design space exploration of caches.

7. REFERENCES

[1] *MicroBlaze Processor Reference Guide*. `www.xilinx.com` (Last accessed: 01/10/2012).

[2] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Proc. of DAC'11*, pages 274–279, San Diego, California, June 2011.

[3] H. Ding, Y. Liang, and T. Mitra. WCET-centric partial instruction cache locking. In *Proc. of DAC'12*, pages 412–420, San Francisco, California, June 2012.

[4] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 35:163–189, November 1999.

[5] M. Kuo, R. Sinha, and P. S. Roop. Efficient WCRT Analysis of Synchronous Programs using Reachability. In *Proc. of DAC'11*, San Diego, USA, June 2011.

[6] M. Kuo, L. H. Yoong, S. Andalam, and P. Roop. Determining the worst-case reaction time of IEC 61499 function blocks. In *Proc. INDIN'10*, pages 1104 –1109, July 2010.

[7] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3):257–279, July 1999.

[8] Y. Liang and T. Mitra. Static analysis for fast and accurate design space exploration of caches. In *Proc. of CODES+ISSS'08*, pages 103–108, Atlanta, GA, USA, 2008.

[9] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate Estimation of Cache-Related Preemption Delay. In *Proc. of CODES+ISSS'03*, pages 201–206, CA, USA, 2003.

[10] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007.

[11] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18:157–179, 1999.

[12] R. Wilhelm et al. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

APPENDIX

A. REDUCING THE CFG AND COMPUTING RELATIVE INSTRUCTIONS

Given a cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$ and the reference block $b_{ref} \in B$, the objective of the algorithm is: (1) to compute a new reduced graph $G^r = \langle B^r, b_{init}, E^r\rangle$ which contains only these blocks that are relevant for the analysis of block b_{ref} (described earlier in Example 1) and, (2) to compute the function BI^r which describes the relative instructions executed by the blocks in B^r w.r.t. b_{ref}, referred as the *relative instruction mapping* function (described earlier in Example 2). The algorithm contains the following three steps.

Step 1: Initialise (lines 2 to 4) On line 2, we initialise G^r to have the same content as G, i.e., same set of blocks ($B^r = B$), initial block (b_{init}), edges ($E^r = E$). On lines 3 to 4, we initialise the function BI^r such that it does not contain any relative instructions for any block. For the example CFG (presented in Figure 1(a)), the graph G^r is presented in Figure 6(a).

Algorithm 4 *Reduce*: Reduce the CFG and compute relative instructions

Input: Cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$ and a reference block $b_{ref} \in B$.
Output: $G^r = \langle B^r, b_{init}, E^r\rangle$ and $BI^r : B^r \rightarrow (\{\times, \mp, 1, 0\})^N$

1: {Step 1: Initialise}
2: $B^r = B,\ E^r = E,\ G^r = \langle B^r, b_{init}, E^r\rangle$ {Copy all blocks and edges}
3: **for each** $b_1 \in B^r$ **do**
4: $BI^r(b_1) = \emptyset$ {Initialise the vector $BI^r(b_1)$}
5: **end for**
6: {Step 2: Relative instruction mapping}
7: **for each** $b_1 \in B^r$ **do**
8: **for each** $c_i \in C$ **do**
9: **if** $(BI(b_{ref})[i] = \mp)$ {not of interest} **then**
10: $BI^r(b_1)[i] = \times$
11: **end if**
12: **if** $(BI(b_{ref})[i] \neq BI(b_1)[i])$ {different instruction} **then**
13: $BI^r(b_1)[i] = 1$
14: **end if**
15: **if** $(BI(b_{ref})[i] = BI(b_1)[i])$ {same instruction} **then**
16: $BI^r(b_1)[i] = 0$
17: **end if**
18: **if** $(BI(b_1)[i] = \mp)$ {no instruction} **then**
19: $BI^r(b_1)[i] = \mp$
20: **end if**
21: **end for**
22: **end for**
23: {Step 3: Remove vacuous blocks and update edges}
24: **for each** $b_1 \in B^r$ **do**
25: **if** $(BI(b_1) \in (\{\times, \mp\})^N) \wedge (b_1 \neq b_{init})$ {check for all vacuous blocks, excluding initial block} **then**
26: {Compute predecessors and successors of b_1}
27: $Preds = \{b_2 | b_2 \rightarrow b_1 \in E^r\}$
28: $Succ = \{b_2 | b_1 \rightarrow b_2 \in E^r\}$
29: {Remove incoming and outgoing edges of b_1}
30: **for each** $b_2 \in Preds$ **do**
31: $E^r = E^r \setminus \{b_2 \rightarrow b_1'\}$
32: **end for**
33: **for each** $b_2 \in Succ$ **do**
34: $E^r = E^r \setminus \{b_2 \rightarrow b_1'\}$
35: **end for**
36: {Add new edge from each predecessor to each successor }
37: **for each** $b_p' \in Preds$ **do**
38: **for each** $b_s' \in Succ$ **do**
39: $E^r = E^r \cup \{b_p' \rightarrow b_s'\}$
40: **end for**
41: **end for**
42: $B^r = B^r \setminus \{b_1\}$ {Remove block b_1}
43: **end if**
44: **end for**

Step 2: Relative instruction mapping (lines 7 to 22) Given a reference block $b_{ref} \in B$ and a cache line c_i, the instruction of any blocks $b_1 \in B^r$ can be expressed as *different* (1) when $BI(b_1)[i] \neq BI(b_{ref})[i]$ (checked on line 12), or *same* (0) when $BI(b_1)[i] = BI(b_{ref})[i]$ (checked on line 15), or *no instruction* ($\mp$) when there is no instruction in b_1 on cache line c_i, $BI(b_1)[i] = \mp$ (checked on line 18), or *not of interest* ($\times$) when there is no instruction in b_{ref} on cache line c_i, $BI(b_{ref})[i] = \mp$ (checked on line 9). This relation is captured using the relative instruction mapping $BI^r(b_1)$. For illustration, refer to Example 2 in Section 3.

For the graph G^r in Figure 6(a), the relative mapping (w.r.t. $B8$) for every block is shown in Figure 6(b). Note that for all blocks b in B^r, if the reference block (b_{ref}) does not have an instruction on cache line c_i ($BI(b_{ref})[i] = \mp$), then for cache line c_i, the relative instruction for all blocks is $\times$ ($BI^r(b)[i] = \times$). This shows that during the analysis of b_{ref}, the cache line c_i is not considered. For example, in Figure 6(b), the reference block $B8$ does not have an instruction on cache line c_0 ($BI(B8)[0] = \mp$). Thus, for cache line c_0, the relative instruction for all blocks is $\times$ ($BI^r(b)[i] = \times$).

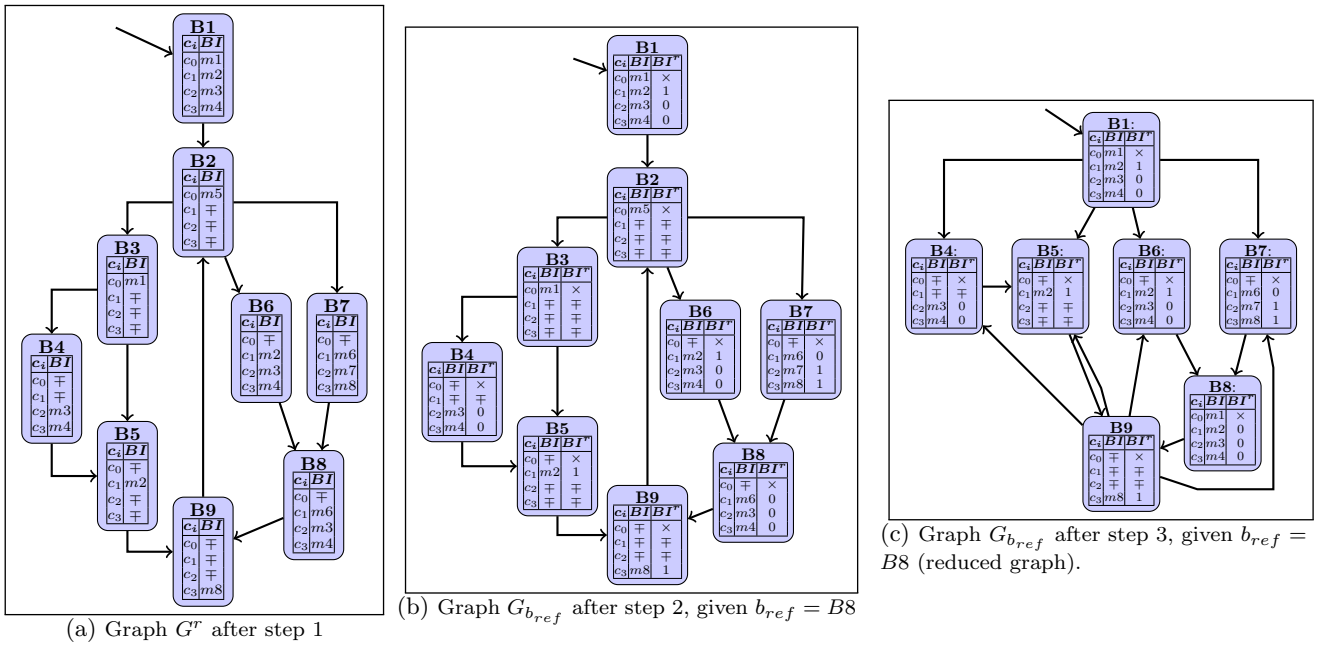
Step 3: Remove vacuous blocks and update edges (lines 24 to 12). As described in Example 1, we can remove vacuous blocks which do not affect the precision of the reference block b_{ref}. We define a block $b_1 \in B^r$ to be vacuous, if $BI^r(b_1) \in (\{\times, \mp\})^N$. This check is done on line 25. It is possible for the initial block (b_{init}) to be vacuous. However, if the initial block has more than one successors, removing the initial block may result in multiple initial blocks. Thus, to simplify the analysis, we do not remove the initial block even when it is vacuous (line 25). If a block b_1 is vacuous, we first compute the predecessors (line 27) and the successors (line 28) of b_1. Secondly, we remove the incoming edges to b_1 from each predecessor block (on lines 30 to 32) and, we remove the outgoing edges from b_1 to each successor block (on lines 33 to 35). Thirdly, we create a transition from each predecessor to each successor of b_1. This is achieved using the nested loop on lines 37 to 41. Finally, on line 42, the vacuous block b_1 is removed. For the graph shown in Figure 6(b), blocks $B2$ and $B3$ do not contain any relative instructions ($BI^r(B2) = [\times, \mp, \mp, \mp]$ and $BI^r(B3) == [\times, \mp, \mp, \mp]$, thus, they are removed for the graph and the updated graph G^r is presented in Figure 6(c). This is the *reduced graph* G^r (w.r.t. to $b_{ref} = B8$).

B. RELATIVE CACHE STATES

We describe the contents of a cache using the notion of *relative cache states*. It is described as a vector $[inst_0^r, inst_1^r, \ldots, inst_{N-1}^r]$, where each element $inst_i^r$ is described w.r.t. the instruction ($BI(b_{ref})[i]$) in the block b_{ref}.

Definition 4 (Relative cache state). *Given a reference block b_{ref}, a relative cache state $cs^r \in (\{1, 0, \top, \bot, \times\})^N$, is a vector $[inst_0^r, inst_1^r, \ldots, inst_{N-1}^r]$, where each element $inst_i^r \in \{1, 0, \top, \bot, \times\}$. Also, the set of all possible relative cache states (w.r.t b_{ref}) is denoted as CS^r.*

Before we illustrate relative cache states, we introduce two key terms essential to cache analysis. For any basic block b, the *reaching relative cache states* represent the set of relative cache states prior to the execution of a basic block and the *relative leaving cache states* represent the set of cache states



(a) Graph G^r after step 1

(b) Graph $G_{b_{ref}}$ after step 2, given $b_{ref} = B8$

(c) Graph $G_{b_{ref}}$ after step 3, given $b_{ref} = B8$ (reduced graph).

Figure 6: Illustration of Algorithm 4 with $b_{ref} = B8$.

after the execution of the basic block. We denote the reaching relative cache states of a basic block b as RCS_b^r and the relative leaving cache states of a basic block b as LCS_b^r.

B.1 Illustration

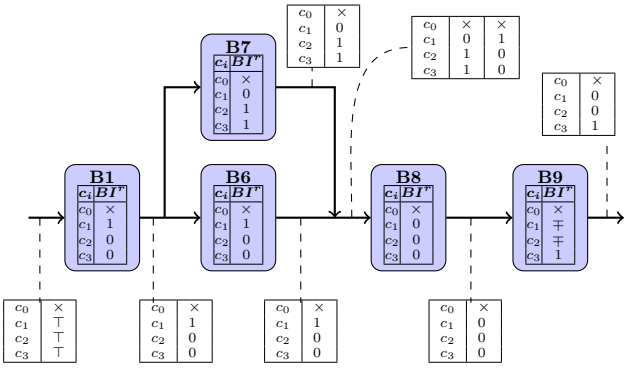


Figure 7: Illustration of the relative cache states.

A fragment of the reduced graph (Figure 6(c)) is presented in Figure 7. Using this graph we illustrate relative reaching and leaving cache states of a block. Given the reference block $b_{ref} = B8$ with $BI(b_{ref}) = [\mp, m6, m3, m4]$ and the relative instruction mapping of each block (described by the function BI^r), we illustrate the relative cache state as we start executing the initial block $B1$. Initially, the cache is empty and the reference block does not have an instruction on cache line c_0 ($BI^r(b_{ref})[0] = \mp$). Thus the instruction on cache line c_0 is *not of interest* ($\times$) resulting in the initial cache state of $[\times, \top, \top, \top]$, denoted as $cs_\top^r$. Note this initial cache state is unlike the concrete approach, where the initial state is represented using the vector $[\top, \top, \top, \top]$.

In our approach, given the reference block b_{ref} (in this case $b_{ref} = B8$), we are only interested in the cache line c_1, c_2, c_3. Thus, we ignore the cache state w.r.t cache line c_0, as it does not affect the precision of the reference block. Also, by ignoring c_0, allows for a memory efficient implementation. In this case, the relative cache state can be represented using a vector with only three elements, instead of four elements, saving 25% of memory.

After execution of block $B1$, where the relative instructions are described by $BI^r(B1) = [\times, 1, 0, 0]$, cache lines c_1, c_2, c_3 will contain relative instructions 1, 0, 0. Hence after $B1$ executes, the relative cache state $cs^r = [\times, 1, 0, 0]$. Here, $cs^r[0] = \times$ represents the instruction on cache line c_0 is *not of interest*. $cs^r[1] = 1$ represents that the instruction in the cache is not $m6$, because $BI(b_{ref})[1] = m6$. $cs^r[2] = 0$ represents that the instruction in the cache is $m3$, because $BI(b_{ref})[2] = m3$. Similarly, $cs^r[3] = 0$ represents the instruction in the cache is $m4$ ($BI(b_{ref})[3] = m4$).

The relative cache state $cs_\top^r$ is the state of the cache prior to the execution of the basic block $B1$. Thus, $RCS_{B1}^r = \{[\times, \top, \top, \top]\}$ is the set of reaching relative cache states of block $B1$. Similarly, $LCS_{B1}^r = \{[\times, 1, 0, 0]\}$ is the set of relative leaving cache states of block $B1$.

Now, the control reaches a branch due to which, it is possible to execute either block $B6$ or block $B7$. In this case, $RCS_{B6}^r = LCS_{B1}^r = RCS_{B7}^r$. After executing blocks $B6$ (or $B7$), the state of the cache is $[\times, 1, 0, 0]$ (or $[\times, 0, 1, 1]$).

Block $B8$ has two incoming edges: from blocks $B6$ and $B7$. To compute RCS_{B8}^r, we need to *join* LCS_{B6}^r and LCS_{B7}^r. In this case, the join function is a union over the set of relative cache states. Thus, $RCS_{B8}^r = LCS_{B6}^r \cup LCS_{B7}^r$, resulting in $RCS_{B8}^r = \{[\times, 0, 1, 1], [\times, 1, 0, 0]\}$.

C. COMPUTING ALL POSSIBLE REACHING RELATIVE CACHE STATES OF THE REFERENCE BLOCK

The first step for cache analysis involves the computation of all possible reaching relative cache states of b_{ref} ($RCS_{b_{ref}}^r$), using the fixed point computation algorithm presented in Algorithm 5.

As illustrated in Figure 7, the initial state of the cache is empty($cs_\top^r$), and is based on the instructions of the reference block. Similarly, like the concrete and the abstract ap-

proaches, we must also introduce the unknown cache state ($cs^r_\perp$), which is explained later during this algorithm. In general, the empty/unknown cache state is different for each $b_{ref} \in B$. Thus, for a given reference block b_{ref}, we first compute the empty cache state $cs^r_\top$, and unknown cache state $cs^r_\perp$ on lines 2 to 8.

For each cache line c_i, if the reference block b_{ref} does not have an instruction (in this case, $BI^r(b_{ref}) = \times$), then the cache state on c_i is *not of interest* during the analysis of the reference block b_{ref}. Thus, the relative instruction on cache line c_i of $cs^r_\top$ and $cs^r_\perp$ is set to $\times$ (line 4). Otherwise ($BI^r(b_{ref}) \neq \times$), on line 6, for cache line c_i, the empty cache state is set to be $\top$ ($cs^r_\top[i] = \top$), and the unknown cache state is set to be $\perp$ ($cs^r_\perp[i] = \perp$).

Using relative cache states $cs^r_\top$ and $cs^r_\perp$, we initialise the reaching relative cache states for all blocks (lines 11 to 17). Since we assume that initially the state of the cache is empty, on line 13 for the initial block b_{init} we set its reaching as $RCS^{r^1}_{b_{init}} = \{cs^r_\top\}$. Here, the notation $RCS^{r^i}_b$ represents the reaching relative cache states of block b in iteration i. E.g., $RCS^{r^1}_{b_{init}}$ represents the reaching relative cache states of block b_{init} for iteration 1. For rest of the blocks, the initial state of the cache is unknown. Thus, on line 15, we set their reaching cache states as $\{cs^r_\perp\}$. After initialisation, we compute the relative leaving cache states of each block, on lines 19 to 22. We apply the transfer function(T) to every block and its corresponding reaching relative cache states.

The iteration index (i) is incremented (line 23) to signal the start of the next iteration. Next, on lines 25 to 34, the reaching relative cache states of each block are computed. For the initial block b_{init}, we know that the reaching relative cache state is always empty. Thus, on line 27, we always set its reaching as $RCS^{r^i}_{b_{init}} = \{cs^r_\top\}$. For rest of the blocks, we first initialise the reaching relative cache state as empty set (line 29), and on lines 30 to 32, the reaching cache states are computed by looking at the relative leaving cache states of the predecessors (b') of the block b and using the union operation.

The iterative process, repeat-until loop on lines 18 to 35, is repeated until a fixed point is reached, i.e., if two consecutive iterations have the same sets of reaching relative cache states for all blocks (line 35).

Algorithm 5 FP: Fixed point computation for the proposed approach

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$, reduced graph $G^r = \langle B^r, b_{init}, E^r\rangle$ and $BI^r : B^r \rightarrow (\{\times, \mp, 1, 0\})^N$.
Output: Reaching relative cache states of block b_{ref} ($RCS^r_{b_{ref}}$).

1: {Initialise $cs^r_\top$ and $cs^r_\perp$}
2: **for each** $c_i \in C$ **do**
3: **if** $BI^r(b_{ref})[i] = \times$ **then**
4: $cs^r_\top[i] = \times$, $cs^r_\perp[i] = \times$ {When b_{ref} has no instruction (in this case, $BI^r(b_{ref}) = \times$) on cache line c_i, initialise empty/unknown cache states as *not of interest*.}
5: **else**
6: $cs^r_\top[i] = \top$, $cs^r_\perp[i] = \perp$ {When b_{ref} has an instruction on cache line c_i, initialise empty/unknown cache states as $\top/\perp$.}
7: **end if**
8: **end for**
9: $i = 1$ {iteration counter}
10: {Initialise RCS^r for all blocks }
11: **for each** $b \in B$ **do**
12: **if** $b = b_{init}$ **then**
13: $RCS^{r^1}_{b_{init}} = \{cs^r_\top\}$ {for the initial block, the initial state of the cache is always empty}
14: **else**
15: $RCS^{r^1}_b = \{cs^r_\perp\}$ {for rest of the blocks, the initial state of the cache is unknown}
16: **end if**
17: **end for**

18: **repeat**
19: {Compute LCS^r for all blocks}
20: **for each** $b \in B^r$ **do**
21: $LCS^{r^i}_b = T(RCS^{r^i}_b, b)$
22: **end for**

23: $i = i + 1$; {Next iteration}
24: {Compute RCS^r for the next iteration $i + 1$}
25: **for each** $b \in B^r$ **do**
26: **if** $b = b_{init}$ **then**
27: $RCS^{r^i}_{b_{init}} = \{cs^r_\top\}$
28: **else**
29: $RCS^{r^i}_b = \emptyset$
30: **for each** $LCS^{r^i}_{b'}$, where $(b', b) \in E^r$ **do**
31: $RCS^{r^i}_b = RCS^{r^i}_b \cup LCS^{r^i}_{b'}$
32: **end for**
33: **end if**
34: **end for**

35: **until** $\forall b \in B^r$, $RCS^{r^i}_b = RCS^{r^{i-1}}_b$ {Termination condition}
36: **return** $RCS^r_{b_{ref}}$

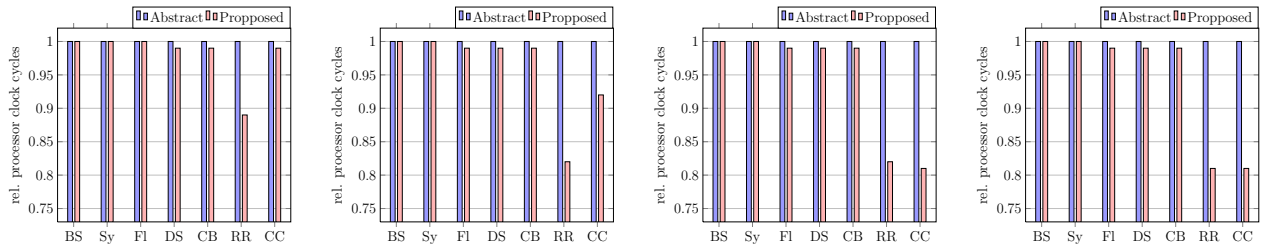
D. RESULTS

Example	abstract		proposed		concrete		Gain
	WCET (clks)	AT (sec)	WCET (clks)	AT (sec)	WCET (clks)	AT (sec)	(col5/ col2)
BubbleSort	2571	0.7	2571	36.6	2571	44	1
Synthetic	14134	1.4	14134	8.2	14134	311	1
Flasher	117508	1.8	95908	41.9	T.O	T.O	0.82
DrillStation	31881	1.9	27453	45.1	T.O	T.O	0.86
ConvBeltModel	21344	1.7	18104	35.1	T.O	T.O	0.85
RailRoadCrossing	308505	4.0	250725	142.0	T.O	T.O	0.81
CruiseController	357206	3.6	288374	113.7	T.O	T.O	0.80

Table 5: Quantitative comparison between the abstract, proposed and concrete approaches

The WCET and the analysis time for abstract is presented in columns 2 and 3 respectively of Table 5. Similarly, the following columns present results for the proposed and concrete approaches. Final column presents the WCET estimate of the proposed approach w.r.t. the abstract approach. The proposed approach is tighter by 13% on average, and up to 19% tighter than the abstract approach.

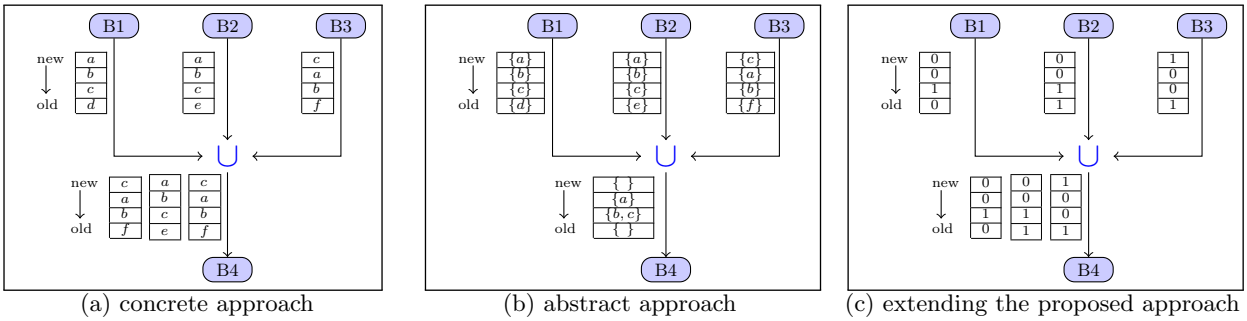
Using the cache size between 0.2% and 0.9% of the program size, for each example, the WCET analysis results are presented in Figures 8(a)–8(h). Across all the benchmarks, we observe that the WCET estimates from the proposed approach is always less than or equal to the estimates from the abstract approach. With 1% relative cache size, on average, the WCET of proposed approach gives 13 % much tighter results and upto 19% tighter result than the abstract approach.



(a) 0.2% relative cache size (b) 0.3% relative cache size (c) 0.4% relative cache size (d) 0.5% relative cache size

(e) 0.6% relative cache size (f) 0.7% relative cache size (g) 0.8% relative cache size (h) 0.9% relative cache size

Figure 8: Comparing the WCET of the abstract and proposed approaches between 0.2% and 0.9% relative cache sizes



(a) concrete approach (b) abstract approach (c) extending the proposed approach

Figure 9: Extending and comparing the proposed approach to set associative caches

E. EXTENDING OUR APPROACH TO SET ASSOCIATIVE CACHES

The proposed approach can easily be extended for timing analysis of set associative caches. For a 4 way set associative cache with the least recently used replacement (LRU) policy [3], a comparison of the *join operation* between the concrete and abstract approaches is presented in Figure 2 of [3]. Using this exact example, we reproduce Figures 9(a) and 9(b). Here, block $B4$ has three predecessor blocks (B1, B2 and B3). The stack(s) next to the transitions represent the state of the cache, for each cache line. The order represents the history, the recent instruction is on top of the stack, and the least recent instruction is at the bottom of the stack.

For the concrete approach (Fig. 9(a)), the reaching cache states of $B4$ are computed as the union of all the leaving cache states of the predecessor blocks, resulting in three cache states. This is very precise, but will not scale for large programs. In the case of the abstract approach (Fig. 9(b)), the instructions are combined based on the *upper bound of its age*. E.g., for the leaving cache states of $B1$, $B2$ and $B3$, instruction 'a' resides on *top* (first), *top* (first) and *second* positions, respectively. Since, the upper age bound (oldest) is the *second* position, the join function abstracts the age of instruction 'a' as *second* position. Similarly, instruction 'b' and 'c' are computed as *third* position. In contract, instruction 'd', only exists in the leaving cache state of $B1$, and not in $B2$ and $B3$. In this case, we cannot guarantee its presence, so it is removed from the stack. Similarly, instructions 'e' and 'f' are removed during the join operation. This abstraction, improves scalability, but lacks precision.

For the proposed approach (Fig. 9(c)), the instructions in the cache are presented w.r.t. the instructions of the reference block, 0 if *identical* , or 1 otherwise. E.g., let us assume that instructions a, b, d are abstracted as 1, and c, e, f are abstracted as 0. This abstraction is similar to the idea of *relative cache states* presented earlier in Sec. 3.2.1. Once again, the join operation performs the union over the reaching relative cache states, maintaining both precision and scalability.

Since the join operations of each approach is similar to their counter parts in direct-mapped analysis, we believe the complexity of the three approaches may be similar to Tab. 2. Also, the WCET and the analysis time may reflect the trends seen in Figures 4 and 5.