Data Cache Analysis by Counting Integer Points

Pascal Sotin, Quentin Vermande, Hugues Cassé

Data Cache Analysis by Counting Integer Points

Pascal Sotin
IRIT – UT2J
Toulouse, France
pascal.sotin@irit.fr

Quentin Vermande
École Normale Supérieure
Paris, France
quentin.vermande@ens.fr

Hugues Cassé
IRIT – UPS
Toulouse, France
hugues.casse@irit.fr

ABSTRACT

The scheduling of reliable real-time systems require a precise and sound analysis of the execution times of their tasks. Part of these execution times is spent fetching data from the main memory to the cache memories. These fetch events occur on cache misses, but cache misses are hard to predict when the program accesses an array. An imprecise cache miss analysis can lead to an imprecise but still sound Worst-Case Execution Time (WCET) analysis. In this article we present a framework for deriving an upper bound to the number of times a data-accessing instruction triggers a cache miss. Backed by this framework, we present an analysis that produces numeric or symbolic bounds, by reasoning on a short history of accesses and by counting the number of integer points in a volume with an existing tool (Barvinok). Using such bounds will improve the precision of the estimations delivered by the WCET analyses.

KEYWORDS

static analysis, data cache analysis, formal methods, worst-case execution time analysis, precondition calculus

ACM Reference Format:
Pascal Sotin, Quentin Vermande, and Hugues Cassé. 2021. Data Cache Analysis by Counting Integer Points. In *29th International Conference on Real-Time Networks and Systems(RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3453417.3453424

1 INTRODUCTION

A *cache memory* is a hardware placed between the processor and the main memory. It manages copies of blocks of the main memory in order to speed up the execution of programs by giving fast answers to the processor memory accesses. The memory can be read or written and can contain instructions or data. An access treated by the cache without fetching data from the main memory is called a *hit*, and a *miss* otherwise.

Cache memories are primordial for the timing performances of the processors. Disabling them would slow down the execution of programs by one or two order of magnitudes. The reason why cache memories perform so well is that they are much faster than main memory and that they benefit from two patterns omnipresent in programs: *temporal and spatial locality* of the memory accesses. This means that programs tend to access repeatedly the same addresses or close addresses, possibly interleaved with other accesses.

Cache Analysis. The dynamic behaviour of caches makes that predicting the outcome of an access is non-trivial. However, a precise characterization of the hits and misses is required for computing a precise Worst-Case Execution Time (WCET) estimation. Therefore, cache analyses were developed to derive that information.

Instruction cache analysis is a well-studied issue. The standard approach is to categorize the accesses to the memory as *always hit*, *always miss*, *persistent*[1] or *not classified* [6–8].

For data cache analysis no fully satisfying solution was found (see related work in Section 10). The classification performed for the instruction cache do no apply straightforward due to fact that the data accessed by an instruction depend on a computed address.

```
int sum = 0;
for (int i = 0; i < N; i++) {
  sum += t[i];      // can read(t[i]) be a miss?
}
```

Program 1: Summation of an array `t` of integers.

Consider the Program 1 and consider the read access to `t[i]` nested within the loop (line 3). Can this access be a miss?

- The answer is *no* if the whole array `t` is already in the cache. This property is not trivial to establish but it brings down the miss bound from `N` to `0`.
- The answer is also *no* as long as we stay in a memory block in cache. This property would reduce the miss bound from `N` to a fraction of `N`. Finding such property is a *counting problem* rather than a classification problem.

Counting Integer Points. In [17], Verdoolaege et al. present a tool, called Barvinok, that can count the integer points contained in certain volumes, called parametric polytopes. Roughly speaking, it performs the transformation:

$$\sharp\left\{\vec{n}\in\mathbb{Z}^k \;\middle|\; \phi(\vec{n},\vec{x})\right\} \quad\rightsquigarrow\quad \mathcal{N}(\vec{x}) \tag{1}$$

where $\sharp S$ denotes the cardinality of the set S, ϕ is a (restricted) logical formula, k is the dimension of the polytope, $\vec{x}$ denotes the polytope parameters, and $\mathcal{N}$ is a count function that evaluates to a natural number in constant time. This work might seem unrelated to our problem, but the authors of [17] motivate their work with several counting questions occurring in program analyses and optimizations, notably this one:

How many cache misses does a loop generate?

As we will see, the Barvinok tool can be used to solve data cache analysis problems, but its application is not straightforward.

[1]Persistent means hit or miss at the first iteration of the enclosing loop, then only hits [5, 16].

Proposal. In this article we present a static analysis that computes a symbolic *upper-bound* for the number of data cache misses triggered by *one* given memory access located inside a loop. Roughly speaking, we allow the following transformation:

$$(C, \mathtt{P}, acc) \rightsquigarrow \sharp \left\{ n \in \mathbb{N} \,\middle|\, Init \,;\, Body^{\langle n \rangle} \,;\, \Diamond Miss \right\} \rightsquigarrow \mathcal{U} \quad (2)$$

where:

- C is a cache model, $\mathtt{P}$ is a program with no nested loops and acc designates *one* memory access within a loop body;
- $(Init \,;\, Body^{\langle n \rangle} \,;\, \Diamond Miss)$ is a predicate denoting the existence of an execution of program $\mathtt{P}$ on an architecture equiped with a cache of model C that successively: reaches the loop head, performs n complete iterations of the loop, and eventually *may*[2] trigger a cache miss at access acc;
- $\mathcal{U}$ is an upper-bound on the number of misses represented by a count function parametrised by the initial program state (e.g. initial value of a variable, address of an array).

Note that we do not consider the computation of $Init$ and $Body^{\langle n \rangle}$ in this article.

Contributions. We claim that:

- The upper bounds computed by our analysis reflects well the cache hits due to the *spatial and temporal locality* of the accesses inside the loop. We illustrate our analysis and its results on Program 1 (Section 3).
- The *soundness* of our analysis is ensured by its formal derivation from a concrete semantics that exposes the event to count (Sections 2 and 4).
- The predicate $\Diamond Miss$ can be *computed* using a Weakest Precondition Calculus according to the nature of the cache and the instructions preceding the memory access (Section 5).
- The predicate characterising a possible miss at iteration n is not shaped like a polytope but it can be *turned* into a suitable input for Barvinok, soundly and mechanically (Section 6).
- Our analysis can be adapted to handle several sources of cache misses (Section 7.1) and nested loops (Section 7.2).

Our analysis is not fully automated but the calculus of Section 5 and the tranformation of Section 6 have been developed. We present these implementations and some performance indications in Section 8. We discuss scalability in Section 9 and present our positioning with respect to existing proposals for data cache analysis in Section 10. We conclude in Section 11.

2 TERMINOLOGY AND MATHEMATICAL NOTATION

In this section, we introduce the concepts, terms and notations used throughout the article.

Programs and Events. A *program* is a text in a given programming language. A *semantics* characterises formally the executions of a program. An *event* designates a relevant instant in the execution of a program, or a family thereof (e.g. a cache miss). We rely on a program semantics related to a given event, represented by a set of traces (detailed in Section 4). A *trace* is a non-empty sequence of program states. A program *state* reflects the logical state of the machine that executes the program (e.g. values of the variables, content of the cache). Let Σ be the set of all program states.

[2]The $\Diamond$ symbol is borrowed from modal/temporal logic. Read it: "possible that…".

Sets and Relations. We write $\wp(S)$ for the set of subsets of S (powerset) ; $A \times B$ for the cartesian product of A and B ; $\{x \in S \mid \phi(x)\}$ for the subset of elements of S satisfying the formula ϕ ; and $\sharp A$ for the number of elements of A (cardinality).

We represent *sets of states*, in $\wp(\Sigma)$, using logical formulas. For example $\mathtt{i} = 0$ denotes the set of states $\left\{\sigma \in \Sigma \,\middle|\, [\![\mathtt{i}]\!]\sigma = 0\right\}$ where $[\![\mathtt{i}]\!]\sigma$ is the *evaluation* of the program variable $\mathtt{i}$ within program state σ. Throughout the article, we use *typewriter font* as a hint indicating program expressions.

We use *relations on states*, in $\wp(\Sigma \times \Sigma)$, to denote the effect of program statements. We also represent relations using logical formulas. For example $\mathtt{i}' = \mathtt{i} + 1$ denotes the relation:

$$\left\{\langle \sigma, \sigma' \rangle \in \Sigma \times \Sigma \,\middle|\, [\![\mathtt{i}]\!]\sigma' = [\![\mathtt{i}]\!]\sigma + 1\right\}$$

The relation links the value of the program variables before (unprimed) and after (primed). Relations R_1 and R_2 can be composed in a relation $R_1 \,;\, R_2$ defined by:

$$\langle \sigma, \sigma'' \rangle \in (R_1 \,;\, R_2) \iff \exists \sigma', \langle \sigma, \sigma' \rangle \in R_1 \wedge \langle \sigma', \sigma'' \rangle \in R_2$$

A relation R and a state S can be composed in a state $R \,;\, S$ defined by:

$$\sigma \in (R \,;\, S) \iff \exists \sigma', \langle \sigma, \sigma' \rangle \in R \wedge \sigma' \in S$$

We call *sound* approximation of a set A any set B such that $A \subseteq B$. This also applies if A and B are relations. These approximations of sets and relations coincide with the ordering of powersets used in Abstract Interpretation [2].

Counting. The property we look for is defined by counting the number of elements of a given set. We have the properties:

$$A \cap B = \emptyset \implies \sharp(A \cup B) = \sharp A + \sharp B \quad (3)$$

$$A \subseteq B \implies \sharp A \leq \sharp B \quad (4)$$

Necessary and Sufficient Conditions. In the formula $\phi_A \implies \phi_B$, the formula ϕ_A is called a sufficient condition and ϕ_B a necessary condition. This formula is equivalent to $\neg\phi_B \implies \neg\phi_A$.

Throughout the article, in order to be sound, we often look for formulas entailed by the program behaviour (necessary conditions, over-approximation) or, playing with negations, formulas entailing the absence of some program behaviours (sufficient conditions, under-approximation).

3 ANALYSIS PRINCIPLE

In this section, we illustrate the principle of our analysis by applying it to Program 1. We wish to bound the number of cache misses triggered by the read accesses to the cells of the array `t`.

We assume that the compiler decides to put the variables `sum` and `i` in registers and to align `t` on an integer boundary. We assume that an integer occupies 4 bytes. We assume that the cache:

- contains blocks of 32 bytes, hence 8 integers,
- ensures that the latest block fetched remains in the cache (very weak assumption).

In Section 5, we will show how our analysis takes into account more complex assumptions on the cache.

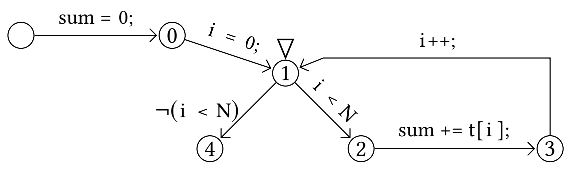

Figure 1: Control flow graph of the loop of Program 1.

3.1 Analysis stages

We perform the analysis on a Control Flow Graph (CFG) of Program 1, shown on Figure 1. Contrarily to CFG used in compilers and disassemblers, the instructions are carried by the edges, not by the nodes. The symbol ∇ marks the loop head. The instructions followed by a semi-column are assignments; the others are guards. The test of the `for` loop gives birth to two guards.

Loop Analysis. The relation B denotes the effect of a loop iteration, from the loop head, back to the loop head[3]. The relation $B^{\langle n\rangle}$ denotes an over-approximation of B^n, the effect of $n \in \mathbb{N}$ loop iterations. In this article, we do not consider how such a relation can be derived. For the loop of Program 1, we have:

$$B^{\langle n\rangle} \iff \mathtt{i}' = \mathtt{i} + n \ \wedge\ \mathtt{N}' = \mathtt{N} \ \wedge\ \&\mathtt{t}' = \&\mathtt{t} \tag{5}$$

Equation (5) state that the execution of n loop iterations increases the initial value of variable `i` by n but does not alter the value of `N` nor the address of the array `t` (written &`t`). It says nothing about `sum`, the content of `t` or the cache state. As mentioned in Section 2, Equation (5) is a lightweight notation for:

$$\forall \sigma, \sigma' \in \Sigma, \quad \langle \sigma, \sigma' \rangle \in B^{\langle n\rangle} \iff \begin{pmatrix} [\![\mathtt{i}]\!]\sigma' = [\![\mathtt{i}]\!]\sigma + n \\ \wedge\ [\![\mathtt{N}]\!]\sigma' = [\![\mathtt{N}]\!]\sigma \\ \wedge\ [\![\&\mathtt{t}]\!]\sigma' = [\![\&\mathtt{t}]\!]\sigma \end{pmatrix} \tag{6}$$

Miss Condition Analysis. Based on the hypotheses on the cache, we construct a formula denoting a *necessary* condition for triggering the cache miss in the current iteration. In our example:

$$\Diamond Miss \quad \Rightarrow \quad \Diamond \left(\mathrm{loc} = \underline{2} \ \wedge\ \widehat{age}(\mathrm{bk}(\mathtt{t}, \mathtt{i})) \neq 0\right) \tag{7}$$

where:

- the symbol ◇ denotes the *possibility* of what follows;
- loc is the current control point (a node of Figure 1);
- $\mathrm{bk}(\mathtt{t}, \mathtt{i})$ is the memory block identifier containing the integer `t[i]`, defined as $\left\lfloor \frac{\&\mathtt{t}+4\mathtt{i}}{32} \right\rfloor$;
- and $\widehat{age}$ is a function representing the current cache, binding a *maximal age* to every memory block. By convention, the most recently accessed block has age 0.

We perform a *backward computation* to transform the condition into a predicate suitable and useful for composition after $B^{\langle n\rangle}$. This computation creates a family $\bar{\mathcal{M}}$ of conditions *sufficient* for *avoiding* the considered cache miss. The conditions computed for Program 1 are shown in Table 1. The function $\mathrm{wp}\,(\mathtt{stmt}, Q)$ returns a sufficient condition for establishing Q after `stmt`.

[3]Hence $\langle \sigma, \sigma' \rangle \in B$ means that if the current program state is σ, then entering the loop and returning to the head through a back-edge *can* end up in program state σ'.

Condition	Definition	Value
$\bar{M}_{\underline{2},0}$	$\neg(\widehat{age}(\mathrm{bk}(\mathtt{t},\mathtt{i})) \neq 0)$	$\widehat{age}(\mathrm{bk}(\mathtt{t},\mathtt{i})) = 0$
$\bar{M}_{\underline{1},0}$	$\mathrm{wp}\left(\mathtt{i} < \mathtt{N}, \bar{M}_{\underline{2},0}\right)$ $\wedge\ \mathrm{wp}\,(\neg(\mathtt{i} < \mathtt{N}), \mathit{true})$	$\widehat{age}(\mathrm{bk}(\mathtt{t},\mathtt{i})) = 0$ $\vee\ \mathtt{i} \geq \mathtt{N}$
$\bar{M}_{\underline{3},+1}$	$\mathrm{wp}\left(\mathrm{i{+}{+}}, \bar{M}_{\underline{1},0}\right)$	$\widehat{age}(\mathrm{bk}(\mathtt{t},\mathtt{i}+1)) = 0$ $\vee\ \mathtt{i}+1 \geq \mathtt{N}$
$\bar{M}_{\underline{2},+1}$	$\mathrm{wp}\left(\mathrm{access}\ \mathtt{t}[\mathtt{i}], \bar{M}_{\underline{3},+1}\right)$	$\mathrm{bk}(\mathtt{t},\mathtt{i}+1) = \mathrm{bk}(\mathtt{t},\mathtt{i})$ $\vee\ \mathtt{i}+1 \geq \mathtt{N}$
$\bar{M}_{\underline{1},+1}$	$\mathrm{wp}\left(\mathtt{i} < \mathtt{N}, \bar{M}_{\underline{2},+1}\right)$ $\wedge\ \mathrm{wp}\,(\neg(\mathtt{i} < \mathtt{N}), \mathit{true})$	$\mathrm{bk}(\mathtt{t},\mathtt{i}+1) = \mathrm{bk}(\mathtt{t},\mathtt{i})$ $\vee\ \mathtt{i}+1 \geq \mathtt{N}$

Table 1: Sufficient conditions for avoiding the miss.

The formula $\bar{M}_{\underline{1},+1}$ states that when the execution is at the loop head (i.e. $\underline{1}$) we can ensure that no miss will occur in the iteration *after this one* (i.e. +1) when either:

- the current iteration will access the same memory block than the following one i.e. $\mathrm{bk}(\mathtt{t},\mathtt{i}+1) = \mathrm{bk}(\mathtt{t},\mathtt{i})$,
- we exit the loop beforehand i.e. $\mathtt{i}+1 \geq \mathtt{N}$.

Note that the formula $\bar{M}_{\underline{1},0}$ is meaningful but its dependence on $\widehat{age}$ would give a disappointing composition with $B^{\langle n\rangle}$ given that $B^{\langle n\rangle}$ carries no information on the cache.

Miss Count. The number of cache misses is *upper bounded* by the expression:

$$\mathcal{U}_{\mathrm{miss}} = 1 + \sharp \left\{ n \in \mathbb{N} \,\middle|\, n \geq 1 \wedge \left(\mathit{Init}\,; B^{\langle n-1\rangle}\,; \neg\bar{M}_{\underline{1},+1} \right) \right\} \tag{8}$$

This expression is a sum because we separated the first access[4] from the subsequent ones. After expansion, Expression (8) gives:

$$\begin{aligned} \mathcal{U}_{\mathrm{miss}} &= 1 \\ &+ \sharp \left\{ n \in \mathbb{N} \,\middle|\, \begin{array}{l} n \geq 1 \wedge \exists \mathtt{i}, \mathtt{i}', \mathtt{N}', \mathtt{t}' \\ \quad \&\mathtt{t} \bmod 4 = 0 \wedge \mathtt{i} = 0 \\ \quad \wedge\ \mathtt{i} + n - 1 = \mathtt{i}' \wedge \mathtt{N} = \mathtt{N}' \wedge \&\mathtt{t} = \&\mathtt{t}' \\ \quad \wedge\ \mathtt{i}' + 1 < \mathtt{N}' \wedge \mathrm{bk}(\mathtt{t}', \mathtt{i}'+1) \neq \mathrm{bk}(\mathtt{t}', \mathtt{i}') \end{array} \right\} \end{aligned} \tag{9}$$

After elimination of the existential quantifiers we get:

$$\mathcal{U}_{\mathrm{miss}} = 1 + \sharp \left\{ n \in \mathbb{N} \,\middle|\, \begin{array}{l} 1 \leq n < \mathtt{N} \ \wedge\ \&\mathtt{t} \bmod 4 = 0 \\ \wedge \left\lfloor \dfrac{\&\mathtt{t} + 4n}{32} \right\rfloor \neq \left\lfloor \dfrac{\&\mathtt{t} + 4(n-1)}{32} \right\rfloor \end{array} \right\} \tag{10}$$

Using a process described in Section 6, we turn Expression (10) into the following expression, that is (almost) suitable for processing by the Barvinok tool:

$$\mathcal{U}_{\mathrm{miss}} = 1 + \sharp \left\{ n \in \mathbb{Z} \,\middle|\, \begin{array}{l} \exists q_a, q_b, q_c \in \mathbb{Z} \\ \quad 1 \leq n < \mathtt{N} \\ \quad \&\mathtt{t} = 4q_a \\ \quad 0 \leq \&\mathtt{t} + 4n - 32q_b < 32 \\ \quad 0 \leq \&\mathtt{t} + 4n - 4 - 32q_c < 32 \\ \quad q_b \neq q_c \end{array} \right\} \tag{11}$$

[4]In order to keep the presentation simple, we treat the first access as a possible miss, regardless of the history, but the framework allows a more precise treatment using $\sharp \left\{ n \in \mathbb{N} \,\middle|\, n = 0 \wedge \left(\mathit{Init}\,; B^{\langle 0\rangle}\,; \neg\bar{M}_{\underline{1},0} \right) \right\}$ as first term of the sum.

Our transformation added three existentially quantified integer variables representing the *quotient* of some divisions. The variable q_a is used to encode the modulo expressing the alignment constraint for the array `t`. The variables q_b and q_c were introduced to cope with the floored divisions. Note that all the transformations presented on that example are exact.

Fed with Equation (11), Barvinok gives us in return the bound:

$$\mathcal{U}_{\text{miss}} = \begin{cases} \left\lfloor \frac{7}{8} + \frac{\texttt{\&t}}{32} + \frac{\texttt{N}}{8} \right\rfloor - \left\lfloor \frac{\texttt{\&t}}{32} \right\rfloor & \text{if } \texttt{N} \geq 2 \\ 1 & \text{otherwise} \end{cases} \tag{12}$$

3.2 Analysis Results

Comments on Equation (12). The bound computed by Barvinok is parametric both in the size of the array, `N`, and the address of the array, `&t`. Table 2 shows the evaluation of $\mathcal{U}_{\text{miss}}$ for some variations on `&t` and `N`.

`&t` \ `N`	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3
4	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3
8	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3
12	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
16	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3
20	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3
24	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
28	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4
32	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3

Table 2: Some evaluations of $\mathcal{U}_{\text{miss}}$

Unsurprisingly, the number of cache misses grows slowly as `N` grows, i.e. as the number of accesses grows, due to spatial locality. The ratio is approximately one miss for eight accesses. The address of the array `t` has a marginal influence on the number of misses. The best case occur when the array is aligned on a cache block boundary. Note that when $\texttt{N} = 9$, the alignment has no influence.

Note that as soon as $\texttt{N} \geq 1$, our bound is the *most precise* bound one can have under the expressed hypotheses. Note also that it is still an *over-approximation* because the number of misses could be zero if the array is entirely in cache when the loop begins.

Variations. If we give additional invariants to Barvinok, we can get simpler formula, or even just numbers. Table 3 shows the formula we get using a variety of additional constraints.

4 SEMANTIC DERIVATION

In this section, we present a framework for counting occurrences of an event during the execution of a program. This framework makes explicit the hypotheses under which expressions like Expression (8) truly reflect what needs to be counted. This section can be skipped at first read.

4.1 Trace Semantics

Let $\mathcal{T}(\Sigma, L)$ be the set of traces of states in Σ labelled by labels in L. A labelled trace $u \in \mathcal{T}(\Sigma, L)$ is of the form:

$$\sigma_0 \xrightarrow{l_0} \sigma_1 \xrightarrow{l_1} \sigma_2 \cdots \sigma_n$$

where the σ_i are in Σ and the l_i are in L. We write:

- u_i for the *state* at index i (hence σ_i);
- $u_{i\rightarrow}$ for the *label* following the state at index i (hence l_i).

Let `P` denote a program and `e` denote a kind of event. The trace semantics of `P` with respect to `e` is written $\mathcal{T}[\![\texttt{P}, \texttt{e}]\!]$ and is a subset of $\mathcal{T}(\Sigma, \mathbb{N})$ where Σ denotes the possible states of Program `P`. This setting is very general and we immediately restrain it with the two following hypotheses.

Hypothesis 1 (One Occurrence Per Transition). *We make the hypothesis that $\mathcal{T}[\![\texttt{P}, \texttt{e}]\!]$ is such that each program transition triggers at most one occurrence of the event. Formally, we have:*

$$\mathcal{T}[\![\texttt{P}, \texttt{e}]\!] \subseteq \mathcal{T}(\Sigma, \{0, 1\}) \tag{13}$$

In the following we interpret 0 and 1 respectively as *false* (i.e. the event did not occur) and *true* (i.e. the event occurred). We write $\mathbb{B}$ instead of $\{0, 1\}$.

Hypothesis 2 (Marking Function). *We make the hypothesis that we have a marking function that can* instrument *traces in $\mathcal{T}[\![\texttt{P}, \texttt{e}]\!]$ with markers such that in each trace the markers identify the events. Formally, we have:*

$$\begin{aligned} &\mathit{instr} : \mathcal{T}[\![\texttt{P}, \texttt{e}]\!] \to \mathcal{T}(M \times \Sigma, \mathbb{N}) \\ &\text{st} : M \times \Sigma \to \Sigma \\ &\text{mark} : M \times \Sigma \to M \end{aligned}$$

such that $v = \mathit{instr}(u)$ implies:

$$\begin{cases} \forall i, \text{st}(v_i) = u_i \\ \forall i, v_{i\rightarrow} = u_{i\rightarrow} \\ \forall i, j, i \neq j \wedge \text{mark}(v_i) = \text{mark}(v_j) \implies v_{i\rightarrow} = 0 \vee v_{j\rightarrow} = 0 \end{cases}$$

This hypothesis will allow us to reason on the set of marks for which the event occur. In Section 5.1, we will use an instrumentation based on a loop counter.

Additional constraints	Miss bound
$\texttt{N} \geq 1 \wedge \texttt{\&t} \equiv 0 \pmod{32}$	$\left\lfloor \frac{7}{8} + \frac{\texttt{N}}{8} \right\rfloor$
$\texttt{N} \geq 1 \wedge \texttt{\&t} \equiv 4 \pmod{32}$	$1 + \left\lfloor \frac{\texttt{N}}{8} \right\rfloor$
$\texttt{N} \geq 1 \wedge \texttt{\&t} \equiv 28 \pmod{32}$	$1 + \left\lfloor \frac{3}{4} + \frac{\texttt{N}}{8} \right\rfloor$
$\texttt{N} = 32$	$4 + \left\lfloor \frac{7}{8} + \frac{\texttt{\&t}}{32} \right\rfloor - \left\lfloor \frac{\texttt{\&t}}{32} \right\rfloor$
$\texttt{N} = 33$	5
$\texttt{N} = 100 \wedge \texttt{\&t} = 512$	13

Table 3: Variations on the miss bound

4.2 Maximal Number of Event Occurrence

Property of Interest. The property that we look for is the *maximal* number of occurrences of the event e during any execution of the program P, given by bound($\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]$), with:

$$\text{bound}(T) = \sup_{u \in T} \left(\sum_i u_{i \to} \right) \tag{14}$$

where sup is the supremum operator, delivering the maximal value if it exits or $+\infty$ otherwise. We consider an analysis to be *sound* if it delivers an *over-approximation* of this number.

Property Transformation. Under Hypothesis 1 we have:

$$\text{bound}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]) = \sup_{u \in \mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]} \sharp \{ i \mid u_{i\to} \} \tag{15}$$

Equation (15) states that under Hypothesis 1, counting in each trace the states triggering the event is equivalent to counting the events themselves.

Under Hypotheses 1 and 2 we have:

$$\begin{aligned} &\text{bound}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]) \\ &= \sup_{v \in \dot{instr}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!])} \sharp \{ m \mid \exists i, \text{mark}(v_i) = m \wedge v_{i\to} \} \end{aligned} \tag{16}$$

where $\dot{instr}$ is the instrumentation function lifted to sets of traces. Equation (16) states that under Hypotheses 1 and 2, counting in each instrumented trace the marks of the states triggering the event is equivalent to counting the events themselves.

Mark Set Approximation. We use the properties of sup and $\sharp$ to turn Equation (16) into the following inequation:

$$\text{bound}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]) \leq \sharp \left\{ m \,\middle|\, \begin{array}{l} \exists v \in \dot{instr}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]), \\ \quad \exists i, \text{mark}(v_i) = m \wedge v_{i\to} \end{array} \right\} \tag{17}$$

Note that the tranformation giving Equation (17) can induce an over-approximation since it does not compute the supremum of the number of marks triggering the event *in each trace* but count the number of marks triggering the event *in some trace.*

4.3 Relational Semantics

We define an approximation of the instrumented trace semantics using the following Galois connection:

$$\begin{aligned} &\mathcal{T}(\Sigma, \mathbb{B}) \underset{\alpha_{\text{rel}}}{\overset{\gamma_{\text{rel}}}{\leftrightarrows}} I{:}\wp(\Sigma) \times R{:}\wp(\Sigma \times \Sigma) \times E{:}\wp(\Sigma) \\ &\alpha_{\text{rel}}(T) = \langle\, I{:}\{u_0 \mid u \in T\}, \\ &\qquad\qquad R{:}\{\langle u_i, u_{i+1}\rangle \mid u \in T\}, \\ &\qquad\qquad E{:}\{u_i \mid u \in T \wedge u_{i\to}\} \,\rangle \end{aligned} \tag{18}$$

For clarity, we tagged our sets with I for *initial*, R for *relation* and E for *event*. As usual, the lattices are ordred by inclusion.

Using this abstraction, we can define a new approximation for our property:

$$\text{bound}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!]) \leq \sharp \left\{ m \,\middle|\, \begin{array}{l} \exists v \in \mathcal{T}(M \times \Sigma), \\ \quad v_0 \in I \\ \quad \forall i, \langle v_i, v_{i+1} \rangle \in R \\ \quad \exists i, v_i \in E \wedge \text{mark}(v_i) = m \end{array} \right\} \tag{19}$$

with $\langle I, R, E\rangle = \alpha_{\text{rel}}(\mathcal{T}[\![\mathsf{P},\mathsf{e}]\!])$. Note that the marks on the traces are no longer considered since E carries that information. It is likely that we can derive precise values for I, R and E from the considered program P and event e.

Precondition Calculus. We can show that performing backward computation using a precondition calculus is sound.

Lemma 4.1. *For all predicates P and Q such that:*

$$\forall \langle \sigma, \sigma' \rangle \in R, \quad P(\sigma) \Rightarrow Q(\sigma') \tag{20}$$

we have:

$$\left(\begin{array}{l} \forall v \in \mathcal{T}(M \times \Sigma), \\ \quad v_0 \in I \\ \quad \forall i, \langle v_i, v_{i+1} \rangle \in R \\ \quad \Rightarrow Q(v_0) \wedge \forall i, P(v_i) \end{array} \right) \Rightarrow \left(\begin{array}{l} \forall v \in \mathcal{T}(M \times \Sigma), \\ \quad v_0 \in I \\ \quad \forall i, \langle v_i, v_{i+1} \rangle \in R \\ \quad \Rightarrow \forall i, Q(v_i) \end{array} \right) \tag{21}$$

Note that the formulas used on each side of Equation 21 are the negated form of the condition found in Equation (19).

Lemma 4.1 allows us to compute sound approximations of our bound by:

- Starting the analysis with a formula denoting:
$$\text{mark}(v_i) = m \Rightarrow v_i \notin E$$
- Computing new formula that is a sufficient precondition for the current formula by the relation R,
- Keeping the formula as it is for the states in I,
- Using the negation of the result formula as $\Diamond Miss$.

5 HANDLING COMPETITION FOR THE CACHE

In this section, we present how the non-miss condition is built, according to the nature of the cache and to the history of memory accesses.

5.1 Loop and Instrumentation

The program we consider in this section is restrained to a loop and the event we track is the triggering of a cache miss related to a specific memory access that:

- can occur at a given control point $\underline{acc}$ within the loop body;
- is not nested within an inner loop.

The array accesses in Programs 1 and 2 satisfy these conditions. It is also the case in Programs 3 and 4 (page 8) but not in Program 5 because of the nested loops.

We instrument the loop with a counter n that is:

- initialized to zero;
- incremented when a back-edge of the loop is taken.

This instrumentation and the fact that the considered access occurs at most once per loop iteration enforces Hypothesis 2: in an execution, every occurrence of the considered miss is identified by a distinct value of n.

5.2 Cache Modeling

We model the cache with a function $\widehat{age} : \mathbb{N} \to \mathbb{N}$ that maps each block identifier to an *upper bound* of the age of that block. This information coincides with the *must* analysis of Ferdinand et al. [7]. A block having a maximal age greater than the cache associativity might be absent from the cache.

Trivial Cache. The formula $\bar{M}_{\underline{1},+1}$ derived in Section 3 holds for any type of cache, since the only hypothesis we made was that the last memory location accessed remained in the cache. The functions determining respectively if an access to a block may be a miss and how evolve the maximal ages[5] after an access were:

$$may\_miss(\widehat{age}, b_{\text{acc}}) \iff \widehat{age}(b_{\text{acc}}) \neq 0$$

$$update(\widehat{age}, b_{\text{acc}}) = \lambda b_{\text{up}}.\begin{cases} 0 & \text{if } b_{\text{up}} = b_{\text{acc}} \\ \infty & \text{otherwise} \end{cases}$$

This approximation was sufficient to prove that the read access in the summation loop only triggers a cache miss every 8 iterations. However, in the presence of other accesses, like in the loop of Program 2, that will not be sufficient to prove that the write access does not evict the cache block loaded by the read access.

```
for (int i = 0; i < N; i++) {
  t[i] = u[i];  // reads u[i] then writes t[i]
}
```

Program 2: Copy loop.

LRU Cache. We now consider a Least Recently Used (LRU) cache made of blocks of B bytes spread among S cache sets, each having an associativity of A. The cache size is thus $B \times S \times A$ bytes. Using an over-approximation of the age, the most precise miss and update functions are:

$$may\_miss(\widehat{age}, b_{\text{acc}}) \iff \widehat{age}(b_{\text{acc}}) \geq A$$

$$update(\widehat{age}, b_{\text{acc}}) = \lambda b_{\text{up}}.$$
$$\begin{cases} 0 & \text{if } b_{\text{up}} = b_{\text{acc}} \\ \widehat{age}(b_{\text{up}}) & \text{if } b_{\text{up}} \neq b_{\text{acc}} \wedge \\ & \quad \begin{pmatrix} b_{\text{up}} \not\equiv b_{\text{acc}} \pmod S \\ \vee\ \widehat{age}(b_{\text{acc}}) = 0 \end{pmatrix} \\ \widehat{age}(b_{\text{up}}) + 1 & \text{otherwise} \end{cases}$$

where $b_1 \not\equiv b_2 \pmod S$ means that the blocks b_1 and b_2 belong to distinct cache sets.

5.3 Sufficient Precondition Analysis

Condition of Interest. We use the may_miss function to express a *necessary* condition $\Diamond Miss_n$ for a miss at iteration n:

$$\Diamond Miss_n(\sigma) \Rightarrow \\ \big(\text{mark}(\sigma) = n \wedge \text{loc}(\sigma) = \underline{acc} \wedge may\_miss(\widehat{age}, b_{\text{acc}})\big) \tag{22}$$

Note that $\Diamond Miss_n$ is not a *sufficient* condition due to the fact that the function $\widehat{age}$ reflect a *maximal* age: an access can still be a hit even if may_miss is true. The negation of $\Diamond Miss_n$ condition is a sufficient condition for the absence of miss at iteration n:

$$\big(\text{mark}(\sigma) = n \wedge \text{loc}(\sigma) = \underline{acc} \quad \Rightarrow \quad \neg may\_miss(\widehat{age}, b_{\text{acc}})\big) \\ \Rightarrow \neg\Diamond Miss_n(\sigma) \tag{23}$$

We compute sufficient preconditions of the form:

$$\text{mark}(\sigma) = n + k \wedge \text{loc}(\sigma) = \underline{l} \quad \Rightarrow \quad \bar{M}_{\underline{l},+k}(\sigma) \tag{24}$$

[5]The result of the $update$ function is a function binding block identifiers to ages. We describe it using a lambda expression ($\lambda x.f(x)$).

We seek for a sufficient precondition such that both:

- The control point is the loop head;
- The mark is n and the condition no longer depend on the $\widehat{age}$ function, or the mark is $n-1$.

These halting conditions make that we need to move backward for one partial iteration, and potentially one more complete iteration.

Computation. The computation is done following the principles of Dijkstra's Weakest Precondition Calculus [4]. We expect that if $P = \text{wp}\,(\texttt{stmt}, Q)$ then $\{P\}\texttt{stmt}\{Q\}$ is a Hoare triple. Guards and assignments are treated as expected in such calculus:

$$\text{wp}\,(\texttt{C} \text{ is true}, \phi) = \texttt{C} \Rightarrow \phi$$
$$\text{wp}\,(\texttt{x} := \texttt{E}, \phi) = \phi[\texttt{x}/\texttt{E}]$$

However, the treatment of memory accesses is not straightforward because it involves updating a *function*. This can be done using parallel assignments:

$$\text{wp}\,(\text{access } \texttt{x}, \phi)$$
$$= \text{wp}\,(\widehat{age} := update(\widehat{age}, \text{bk}(x)), \phi) \tag{25}$$
$$= \text{wp}\,(\|_b\ \widehat{age}(b) := update(\widehat{age}, \text{bk}(x))(b), \phi) \tag{26}$$

Equation (26) might look like an intractable computation.

- On the one hand, this impression is false given that the only assignments having an effect on the result are the ones where $\widehat{age}(b)$ appears in ϕ.
- On the other hand, this computation potentially leads to a disjunction with $(D+1).U^D$ cases where D is the number of blocks to update and U the number of cases generated by each update. However, the good news are:
  - Some cases boil down to *false*. In the tool presented in Section 8.1, we rely on SMT-solving for disproving these spurious cases.
  - These cases give birth to formulas of the form $\neg A \wedge \neg B \wedge \ldots$ where A and B are conjunctions. Since we look for a sufficient condition, we can replace $\neg A \wedge \neg B$ by $\neg C$ provided that $A \vee B \implies C$. Conjunctions are well represented in usual static analysis abstract domains and the computation of C is provided by the join operation of the domain. We did not explore this possibility so far, but the presence of disequation in our conjunctions could be an issue.
  - In each case of the form $\neg(A_1 \wedge A_2 \wedge \ldots)$ we can soundly drop any conjunct A_i. We did not explore the potential of this possibility.

The treatment of the $\widehat{age}$ function update leads us to a ad-hoc theoretical development that we present in the following subsection.

5.4 Sufficient Precondition Calculus on Functions

Our representation of the $\widehat{age}$ function as a logical formula is inspired by the shape analyses based on separation logic, in particular by the work of Rival et al. [11]. Shape analyses target the representation of the memory i.e. a mapping from addresses to values. We adapt their approach to our $\widehat{age}$ function i.e. a mapping from block identifiers to ages. Note that the shape analysis problem is more complex due to the fact that values may be themselves addresses.

Function Representation. We represent the constraints on a function f by a pair $\langle \vec{\mathtt{E}}, \phi \rangle$ interpreted as:

$$\exists \vec{x}, \quad \mathrm{alias}(\vec{x}, \vec{\mathtt{E}}) \land \phi(f, \vec{x}) \tag{27}$$

where the formula ϕ constrains the $\vec{x}_i$ and the $f(\vec{x}_i)$ and where alias is defined by:

$$\begin{aligned} \mathrm{alias}(\vec{x}, \vec{\mathtt{E}}) \quad \iff \quad & \forall i, j, i \neq j \Rightarrow \vec{x}_i \neq \vec{x}_j \\ & \land \forall i, \bigwedge_{\mathtt{e} \in \vec{\mathtt{E}}_i} \vec{x}_i = [\![\mathtt{e}]\!] \end{aligned} \tag{28}$$

The first conjunct of Equation (28) state that all $\vec{x}_i$ are distinct values. The dictionary $\vec{\mathtt{E}}$ stores for each $\vec{x}_i$ a set of alias expressions.

Operations. We list here a set of operations sufficient to perform the precondition calculus of Section 5.3. We do not provide here their algorithms but an implementation has been developed (see Section 8.1). The operations are:

- Yielding a pair $\langle \vec{\mathtt{E}}, \phi \rangle$ equivalent to *true*.
- Exposing an element $\mathtt{a}$ of the function domain. This operation can produce from 1 to $length(\vec{x}) + 1$ cases depending on whether $\mathtt{a}$ can be aliased/unaliased with an existing dimension of $\vec{x}$.
- Simplifying the formula by eliminating the quantified x such that $f(x)$ is not in ϕ.
- Adding a conjunct to ϕ (for treating guards).
- Performing a substitution (for treating scalar assignment).
- Compution a sufficient precondition for a backward update of the function. This operation explores the *combination* of possible updates for each of the quantified variables.

5.5 Results for the Analysis of Program 2

We look for bounds for the number of miss triggered by the read accesses to the array $\mathtt{u}$ in Program 2. A similar analysis, with similar results, could be performed for the write accesses to $\mathtt{t}$.

LRU Cache. We assume a LRU cache of associativity A, containing S sets, with 32 bytes in each block. We use an update function pessimistic with respect to sets. The backward analysis gives us the following necessary condition for a miss at iteration n:

$$\Diamond Miss_n \Rightarrow \left(\begin{array}{l} n = \mathrm{mark} + 1 \land \mathrm{loc} = \underline{h} \land n < \mathtt{N} \\ \land\ \mathrm{bk}(\mathtt{u}, n) \neq \mathrm{bk}(\mathtt{t}, n-1) \\ \land \left(\begin{array}{l} (\mathrm{bk}(\mathtt{u}, n) = \mathrm{bk}(\mathtt{u}, n-1) \land A \leq 1) \\ \lor\ \mathrm{bk}(\mathtt{u}, n) \neq \mathrm{bk}(\mathtt{u}, n-1) \end{array} \right) \end{array} \right) \tag{29}$$

The novelties with respect to Program 1 are the fact that the cache associative enough will surely avoid the miss and that the miss can also be avoided if the access to $\mathtt{t}$ loads the next block.

In order to show a readable formula we add the constraint $\mathtt{N} = 100$ and $A \geq 2$, then Barvinok finds the following bound:

$$\mathcal{U}_{\mathrm{miss}} = \begin{cases} 1 & \text{if } 0 \leq \&\mathtt{u} - \&\mathtt{t} \leq 36 \\ 13 + \left\lfloor \dfrac{3}{8} + \dfrac{\&\mathtt{u}}{32} \right\rfloor - \left\lfloor \dfrac{\&\mathtt{u}}{32} \right\rfloor \end{cases} \tag{30}$$

Direct Mapped Cache. We assume a cache with no associativity ($A = 1$) but 16 sets and still 32 bytes by cache block. The backward analysis gives us a necessary condition for a miss at iteration n:

$$\Diamond Miss_n \Rightarrow \left(\begin{array}{l} n = \mathrm{mark} + 1 \land \mathrm{loc} = \underline{h} \land n < \mathtt{N} \\ \land\ \mathrm{bk}(\mathtt{u}, n) \neq \mathrm{bk}(\mathtt{t}, n-1) \\ \land \left(\begin{array}{l} \left(\begin{array}{l} \mathrm{bk}(\mathtt{u}, n) = \mathrm{bk}(\mathtt{u}, n-1) \\ \land \mathrm{bk}(\mathtt{u}, n) \not\equiv \mathrm{bk}(\mathtt{t}, n-1) \pmod{16} \end{array} \right) \\ \lor\ \mathrm{bk}(\mathtt{u}, n) \equiv \mathrm{bk}(\mathtt{t}, n-1) \pmod{16} \end{array} \right) \end{array} \right) \tag{31}$$

The novelty here is the congruence inspection that could allow a hit if the arrays $\mathtt{t}$ and $\mathtt{u}$ are aligned correctly.

Without placing $\mathtt{t}$ and $\mathtt{u}$, the tool Barvinok produces a huge formula (~90 kB). By placing $\mathtt{t}$ and $\mathtt{u}$, it gives numbers. Some configurations are shown below:

&t

&u	480	484	488		504	508	512	516	520		524	540	544	548
1024	13	13	25	··	74	85	100	100	88	··	76	26	13	13

6 COUNTING USING BARVINOK DECOMPOSITION

In this section, we present the transformation of our formulas encoding the loop summary composed with miss conditions into problems that can be processed by the tool Barvinok.

Back-End Capacities. The tool Barvinok [17] can turn, in polynomial time, expressions of the form:

$$\lambda \vec{x} \in \mathbb{Z}^p. \quad \sharp \left\{ \vec{n} \in \mathbb{Z}^k \;\middle|\; \exists \vec{e} \in \mathbb{Z}^q, \phi(\vec{x}, \vec{n}, \vec{e}) \right\} \tag{32}$$

where ϕ is a conjunction of linear inequalities into expressions of the form:

$$\lambda \vec{x} \in \mathbb{Z}^p. \quad \sum_c \mathcal{I}_{d_c(\vec{x})} \times f_c(\vec{x}) \tag{33}$$

where each term of the sum denotes a *chamber*, each d_c is a predicate denoting a *domain of validity* and each f_c is a count function represented by a *Ehrart polynomial*. We refer to $\vec{x}$ as the parameters, to $\vec{n}$ as the variables and to $\vec{e}$ as the quantified variables.

6.1 Transformations

We present a serie of formula transformation that can be used to reduce the formula characterising a possible miss into a formula supported by Barvinok. These transformations are expressed as term substitutions, with $\phi[s/t]$ being the formula ϕ where all occurrences of the term s have been replaced by the term t. In this article, we do not present how this transformations are chained but we mention that we developed a tool that does the whole transformation (see Section 8.2).

6.1.1 Modulo Elimination. The modulo operation appears when the address of some program data is not fixed but its alignment is constrained. E.g. the array $\mathtt{t}$ is aligned on an integer boundary, the array $\mathtt{u}$ is aligned on a cache boundary. Congruence operations are also produced by the backward analysis when it considers the cache sets.

Lemma 6.1. *For all formula ϕ, for all terms s and t and for all fresh quantified variable q we have:*

$$\phi \iff \exists q \in \mathbb{Z}, \bigwedge \left| \begin{array}{l} \phi[s \bmod t/(s - q * t)] \\ \bigvee \left| \begin{array}{l} 0 \leq (s - q * t) < t \\ t < (s - q * t) \leq 0 \end{array} \right. \end{array} \right. \tag{34}$$

The elimination of a modulo introduces a new quantified variable in the formula and would also create a disjunction if the sign of the divisor t is not known. The latter will not occur in the usages mentioned above. Note that the result of the operator is always of the sign of the divisor[6]. This ensures a continuity in the treatment of the memory even if addresses can go below zero.

6.1.2 Floored Division Elimination. The floored division operation appears in each block identifier computation. This operation is linked with the modulo used above in the sense that:

$$y\left\lfloor\frac{x}{y}\right\rfloor + (x \bmod y) = x \tag{35}$$

Lemma 6.2. *For all formula ϕ, for all terms s and t and for all fresh quantified variable q we have:*

$$\phi \iff \exists q \in \mathbb{Z}, \bigwedge \left| \begin{array}{l} \phi[\left\lfloor\frac{s}{t}\right\rfloor/q] \\ \bigvee \left| \begin{array}{l} 0 \le (s - q * t) < t \\ t < (s - q * t) \le 0 \end{array}\right. \end{array}\right. \tag{36}$$

Here also, the transformation introduce a quantified variable and could create a disjunction if the sign of the divisor is not known.

6.1.3 Non-Linear Term Extraction.

Non-Linear Terms on Parameters. Due to some loop entering condition of the program, the formula might contain conditions like: $n \le \mathtt{M}\times\mathtt{N}$. This inequality contain the non-linear terms $\mathtt{M}\times\mathtt{N}$ but since it contains only parameters, its value does not depend on the number n of iterations. This product, or any non-linear expression can be seen as a parameter.

Lemma 6.3. *For all formula ϕ, for all term t containing only parameters, for all term s, for all relation $\precsim \in \{=, <, \le\}$ and for all fresh parameter p we have:*

$$\phi \iff \phi[s \precsim t/s \precsim p] \wedge p = \lfloor t \rfloor \tag{37}$$

This transformation introduces new parameters and set aside their definitions. Once Barvinok returns a count function, these artificial parameters are replaced by their original definition.

Non-Linear Terms Containing a Variable. In the case of loop update based on integer shifts, the formula might contain relations like: $2^n \le \mathtt{N}$. The transformation presented above cannot be applied because n is not a parameter. However, we can see 2^n as a function $f(n)$ and since this function is invertible we can rewrite the equation into $n \le \log_2(\mathtt{N})$. Eventually, $\log_2(\mathtt{N})$ can be extracted as seen previously.

Lemma 6.4. *For all formula ϕ, for all functions f and f^{-1} such that $f^{-1} \circ f$ is the identity function, for all variable x and term t, we have:*

$$\phi \iff \phi[f(x) = t/x = f^{-1}(t)] \tag{38}$$

Moreover, if $\phi \Rightarrow t \in \text{dom } f^{-1}$ and f^{-1} is monotone, resp. antitone, then for all $\precsim \in \{<, \le\}$ Equation (39), resp. Equation (40), holds:

$$\phi \Rightarrow \phi[f(x) \precsim t/x \precsim f^{-1}(t)] \tag{39}$$

$$\phi \Rightarrow \phi[f(x) \precsim t/x \succsim f^{-1}(t)] \tag{40}$$

Note that this transformation is useless if the term t contains a variable, since it would only transfer the non-linearity.

[6]Like in Python, but unlike Java.

6.1.4 Disjunction Management. We eventually put the formula in Disjunctive Normal Form. Each of the disjunct forms a sub-problem that is submitted to Barvinok. The tool answers are then summed to give the final formula.

It is important that the disjuncts are exclusive from one to the other, otherwise, the sum of their results could be greater than the potential number of misses.

7 ANALYSIS EXTENSIONS

As stated in Section 1, the analysis presented in this article targets *one* memory access that should appear in the body of *non-nested* loop. In this section, we discuss informally how to go beyond this two limitations, using examples.

7.1 Handling Multiple Accesses

Simplest Approach. The naive approach to count the misses generated by several accesses is to perform the analysis several time. For example, in Program 3 that reverses order of the elements of an array `t`, we have potentially four sources of cache misses.

```
1 int low = 0, high = N - 1;
2 while (low < high) {
3   int tmp = t[low];
4   t[low] = t[high];
5   t[high] = tmp;
6   low++; high--;
7 }
```

Program 3: Array reversal

Applying the cache analysis several times will give good results. For a 2-way associative cache, 32 bytes per cache block, `t` aligned on a cache boundary and `N` = 100 we have:

Line	Access	Miss bound
3	read `t[low]`	7
4	read `t[high]`	7
4	write `t[low]`	0
5	write `t[high]`	0

Table 4: Miss bounds for the accesses of Program 3

This gives only 14 cache misses to be compared to the 200 potential cache misses. This is an important achievement for the WCET computation.

However, because the four analyses are independent, they might count a miss *multiple times*. It probably happens here because when `low` reaches `high` the accesses to `t[low]` and `t[high]` fall in the same cache block.

Program 4 shows a situation where summing two analyses will give disappointing results. We assume that `condition(i)` returns a boolean value that we cannot predict and that its execution has a bounded impact on the cache.

```
for (int i = 0; i < N; i++) {
  if (condition(i)) {
    t[i] = u[i];
  } else {
    t[i] = v[i];
  }
}
```

Program 4: Merging arrays

Incorporating Maximums. Given that the accesses occur on different paths within the loop body, it is sound to take the *maximum* of the misses triggered at line 3 and 5. Soundness is ensured by the fact that the backward analysis finds sufficient conditions for avoiding the miss. Thus the miss needs to be avoided whatever decision we make at line 2. For the accesses to `t`, things are alright because both paths ensures that its blocks will be found quite often in the cache. However, for `u` and `v`, things are disappointing: both can occur up to `N` times, while it is clear for the reader that some hits must happen.

Playing with Marks. We have a solution for the specific problem mentionned above. We did not explored how such solution can be generalized but we mention it anyhow. Since the accesses to `u` and `v` belong to distinct path of the body, we can consider them as a single event without breaking Hypothesis 2. We get the necessary condition below for a miss reading `u` or `v` at iteration n. We simplified this condition using the hypothesis that the arrays do not overlap.

$$\Diamond Miss_n \Rightarrow \begin{pmatrix} n = \text{mark} + 1 \wedge \text{loc} = \underline{h} \wedge n < \mathtt{N} \\ \wedge \begin{pmatrix} (\text{bk}(\mathtt{u}, n) = \text{bk}(\mathtt{u}, n-1)) \\ \vee\ (\text{bk}(\mathtt{v}, n) = \text{bk}(\mathtt{v}, n-1)) \end{pmatrix} \end{pmatrix} \tag{41}$$

If we add the constraint $\mathtt{N} = 100$ and place `u` and `v` in memory, then Barvinok finds a bound of 25 misses which is the best we could hope.

7.2 Handling Nested Accesses

Let consider now the case of nested loops. Program 5 shows a two nested loops operating on a matrix `m`. The issue is how to enforce Hypothesis 2: since the inner loop is executed several times by the outer loop, its iteration numbers cannot be considered as unique miss identifiers.

```
int m[N][N];
...
double sum = 0;
for (int row = 0; row < N; row++) {
  for (int col = row; col < N; col++) {
    sum += m[row][col];
  }
}
```

Program 5: Upper triangle sum

Our framework offers two solutions for this problem.

Inner Loop Scaling. The first solution is to perform the analysis on the inner loop in order to get a general bound. The bound we find depends on the parameters `row` and `N`. The presence of `N` is fine since it is constant for the whole program but the presence of `row` is an issue because it varies from one iteration of the outer loop to another.

- We can get rid of the parameter `row` by putting it in the existentially quantified variables of the problem together with the constraint that $\mathtt{row} \geq 0$ and run again Barvinok. It will deliver a miss bound depending only on `N` that we can multiply by the outer loop bound to get a general miss bound. This is sound but imprecise, since Program 5 does not sum the whole matrix.
- Or we can sum the miss bound parametrized by `row` for the successive values of row. This would be more precise but why not delegate this task to Barvinok and stay in the framework?

Nested Identifiers. The second solution is to adapt our instrumentation to the loop structure and identify a miss by a pair $\langle n_{\text{out}}, n_{\text{in}} \rangle$ and consider a problem of the following shape.

$$\sharp \left\{ \langle n_{\text{out}}, n_{\text{in}} \rangle \in \mathbb{N}^2 \;\middle|\; I_{\text{out}} \,; B_{\text{out}}^{\langle n_{\text{out}} \rangle} \,; I_{\text{in}} \,; B_{\text{in}}^{\langle n_{\text{in}} \rangle} \,; \Diamond Miss \right\} \tag{42}$$

The tool Barvinok is tailored for this kind of problems and solves them easily when we can turn the condition into a suitable input. However:

- We do not know how to perform this transformation when `N` is not a constant. This is due to the presence of some non-linearity that does not fall in the transformations presented in Section 6.
- The presence of the inner loop *would* also cause troubles to the backward analysis of Section 5 if the access was outside of the inner loop (which is not the case of Program 5). We would need to compute the precondition before the inner loop, which is non-trivial. This limitation make sense in a framework that pretends to exploit temporal and spatial locality.

Deriving by hand the miss formula and submitting it to Barvinok with $\&\mathtt{m} = 512$ and $\mathtt{N} = 100$ we get a bound of 700 which is only 7% of the accesses.

8 IMPLEMENTATION

In this section, we present a partial implementation of our approach. All the miss conditions $\Diamond Miss$ and miss bounds $\mathcal{U}_{\text{miss}}$ in this article have been computed using the two implementations presented here.

8.1 Backward Analysis

We developed the primitives of the Weakest Precondition Calculus presented in Section 5, in Python. This primitives are then used to write programs that compute preconditions for different control points of the program and distance to the miss, e.g. the formulas of Table 1. We do not parse from C nor from binary so far.

Module	Lines of code
Basic logic formulas	~500
Cache formulas & WPC	~500
Cache models	~100

Our code makes an intensive use of the SAT Modulo Theory solver Z3 [3] to simplify our formulas and to cut the exploration of configurations that cannot occur during the processing of memory accesses. Note that we only exploit the cases where Z3 ensures that a formula is unsatisfiable and do not use the model when the formula is satisfiable. The intuition behind this usage is that the existence of a model is necessary for the existence of a point in the polytope submitted to Barvinok.

8.2 Barvinok Custom Front-End

We developed the primitive for deriving an upper bound on the number of misses from the possible miss formula, using Barvinok and including the tranformations presented in Section 6 (among others). Our tool was developed in OCaml.

Module	Lines of code
Interface with Barvinok	∼ 250
Formula representation	∼ 450
Transformations	∼ 350
User interface	∼ 400

8.3 Preliminary Timing Performances

Even if our two implementations are far from being optimized, we give in Table 5 the measurements for the conditions and bounds shown or mentioned in this article.

Program	Cache	Condition (Sec. 8.1)		Bound (Sec. 8.2)	
		ϕ	Time (s)	$\mathcal{U}$	Time (s)
Prog. 1	trivial	Table 1	0.28	Eq. (11)	0.03
				Table 3 each line	0.01
Prog. 2	LRU	Eq. (29)	0.40	Eq. (30)	0.03
	Direct	Eq. (31)	0.41	huge, p. 7	0.76
Prog. 3	LRU[7]	read `t[low]`	0.33	Table 4 each line	0.01
		read `t[high]`	0.34		
		write `t[low]`	0.26		
		write `t[high]`	0.28		
Prog. 4	LRU	Eq. (41)	0.85	=25, p. 9	0.02
Prog. 5	LRU	read `m[r][c]`	by hand	=700, p. 9	0.01

Table 5: Preliminary performances evaluation

The measurements were conducted on computer equiped with an Intel Core i7-7600U CPU at 2.80GHz × 2, four cores and 32 GB of RAM, running a Debian 10 operating system. The time measurements are expressed in seconds. The times of the column *Condition* are gathered using the cPython profiler. The conditions marked `read` or `write` are not shown in the article.

9 ADDRESSING SCALABILITY

The timing measurements shown in Section 8.3 prove that the approach is feasible for small programs but let open the question of its scalability to larger programs. In this short section, we emphasize several elements concerning scalability.

[7]In the preliminary experiments, the analysis was performed with a symbolic number of bytes per blocks (instead of using the 32 bytes hypothesis) that lead Z3 into long computations (up to 4 minutes for read `t[high]`). From these counter-performances, we should retain that: (a) the cost of genericity can be high and (b) the unpredictability of the SMT solvers behaviour can be a limitation for our approach.

Local Application. First of all, there is little need to target much larger programs because most of the gain we can get comes from the spatial behaviour of the innermost loops. Our typical local gain is decreasing the miss bound from N to $\frac{N}{k}$. This kind of gain persists even if the loop is activated several times.

Genericity Tradeoff. Our approach allows a high level of genericity in its results. A miss bound can depend on unknown characteristics of the cache (e.g. associativity) and on the value of some program variables. Providing detailled information on the cache or on the program values, especially on possible aliases, will specialise the bound and thus limit the possibilities explored by the backward analysis, as mentioned in Section 5.3.

10 RELATED WORK

There is a vast litterature about data cache analysis but no generic solutions has arised for now. Our proposal is quite different from what have been proposed so far:

(1) We have a formal foundation up to the trace semantics; some of the related work uses the framework of Abstract Interpretation but only at the state level.
(2) Our analysis goes backward; their analyses go forward.
(3) Our analysis can exploit the dynamic of the loop; the abstraction performed in the related work tends to reduce sequences of accesses to sets of accesses.
(4) Our analysis can find information for arbitrary size arrays; the analyses below will face either complexity issue or precision issue in such case.

Li et al. [13] propose a two-step strategy: first they determine the set of accessed addresses for each memory instruction (the way it is obtained is not clearly described), then they use Cache Conflict Graphs [12] to cope with memory blocks competing for the same line in the cache. Unfortunately, the complexity of the approach is exponential in the number of accessed memory blocks and in the cache associativity degree. In addition, considering the set of accessed blocks instead of their sequence creates artificial pressure on the cache lines and, as a consequence, overestimates the number of misses.

Ferdinand et al. [8] extend their (Abstract-Interpretation-based) analysis designed for instruction caches to data caches. For that purpose, they identify the set of memory blocks that can possibly be addressed by a memory instruction (a single block for scalar variables or a set of blocks for arrays) and consider that any execution of the memory instruction can target any of these blocks. The paper does not report any experimental results but we implemented it ourselves within our WCET toolset (OTAWA) and found out that it was very pessimistic and imprecise for programs that process large arrays.

White et al. [19] introduce a new category, $\mathit{Calculated}(n)$, to express the behaviour of accesses to arrays, where n stands for the number of expected cache misses. The algorithm to compute n is not clearly described but is based on information provided by the compiler in which their tool is embedded. This approach is limited to direct-mapped caches and to a particular compiler.

Rathijit and Srikant [14] use Circular Linear Progression analysis (CLP) [1, 15]. Accesses to the data cache are abstracted as sets which fails to fully capture the dynamic behaviour of accesses to arrays. To get valuable results, the authors need to unroll each loop several times which drastically increases the computation time for both the cache analysis and the WCET computation.

Huynh et al. [10] propose a completely different approach, the Scope-Aware Persistent analysis. Their analysis is split into three steps: (a) the address analysis, (b) the cache abstract state analysis and (c) the computation of the number cache misses. The address analysis is an adaptation of [20] to obtain memory addresses as symbolic expressions. These addresses are used to get the set of the accessed memory blocks and to refine the persistence analysis by considering temporal scopes (loop iterations). The main challenge here was to determine precisely which memory blocks compete for the same cache set at the same execution time. Experimental results are promising but this approach faces several issues: (1) the complexity depends on the size of the arrays and on the structure of the program (building temporal scopes requires enumerating all accesses); (2) the approach only supports only linear and regular accesses to arrays.

In [9], Hahn et al. use a congruence relation to identify the references that map to the same cache set or memory block. Their analysis, may be applied to symbolic addresses (providing more flexibility) and also to set of addresses (intervals, octagons, etc.), which cannot cope well with the dynamic behaviour of array accesses. In addition, the experimental needs to be extended to ensure the method applies to real applications.

Wegener [18] also uses a congruence relation to detect whether two references point to the same memory block and to infer a hit in the data cache. This approach is successful on scalar variables but requires loop unrolling to precisely support accesses to arrays, that, in turn, increases the computation time of the upcoming analyses.

11 CONCLUSION

In the article, we presented a framework for inferring an upper bound on the number of cache misses triggered by a given memory access during the execution of a loop. The framework was illustrated in Section 3 and a formal foundation can be found in Section 4. The framework uses the tool Barvinok as a backend for computing the bound, which can be numeric or symbolic.

The framework relies on finding a predicate that is a necessary condition for the considered miss or, conversely, finding a predicate that is a sufficient condition to avoid the considered miss. In Section 5 we show how such predicate can be computed using a precondition calculus, a model of the cache and formulas containing an function symbol denoting the maximal age of a block. This analysis produces formulas that characterise how the accesses obey to the temporal and spatial locality in the recent history (i.e. the current iteration or the previous one).

In Section 6, we present several transformations that help us to turn the formula resulting from this analysis into a suitable input for Barvinok. The latter produces upper bounds that range from the absence of miss to as many misses as loop iterations, passing by expressing a fraction of the number of loop iterations.

The key parts of the framework have been implemented into tools presented in Section 8 and all the formulas and bounds shown in this article have been computed using these tools.

We left several points as future work, including:

- a static analysis engine that mechanize the inference of bounds for the memory accesses of a given program,
- the treatment of the possible correlations between several cache miss bounds (sketched in Section 7.1),
- the treatment of the loop nesting (sketched in Section 7.2),
- the use of abstract domains to represent may-miss conditions (mentioned in Section 5.3).

REFERENCES

[1] G. Balakrishnan and T. Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*. Lecture Notes in Computer Science, Vol. 2985. Springer Berlin, 2732–2733.

[2] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340.

[4] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate Calculus and Program Semantics.* Springer. https://doi.org/10.1007/978-1-4612-3228-5

[5] C. Ferdinand. 2005. *A Fast and Efficient Cache Persistence Analysis.* Technical Report. Saarländische Universitäts- und Landesbibliothek / Naturwissenschaftlich-Technische Fakultät I, http://www.scientificcommons.org/2150484.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. 1997. Applying Compiler Techniques to Cache Behavior Prediction. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Embedded Systems.*

[7] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. 1999. Cache Behavior Prediction by Abstract Interpretation. *Sci. Comput. Program.* 35, 2 (1999), 163–189. https://doi.org/10.1016/S0167-6423(99)00010-6

[8] C. Ferdinand and R. Wilhelm. 1998. On predicting data cache behavior for real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Embedded Systems.*

[9] S. Hahn and D. Grund. 2012. Relational Cache Analysis for Static Timing Analysis. In *2012 24th Euromicro Conference on Real-Time Systems.*

[10] B. K. Huynh, L. Ju, and A. Roychoudhury. 2011. Scope-Aware Data Cache Analysis for WCET Estimation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 203–212.

[11] Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. 2010. Separating Shape Graphs. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 387–406. https://doi.org/10.1007/978-3-642-11957-6_21

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. 1995. Efficient microarchitecture modelling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium.*

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. 1997. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Embedded Systems.*

[14] Rathijit S. and Srikant Y. N. 2007. WCET estimation for executables in the presence of data caches. In *ACM & IEEE International Conf. on Embeded Software.*

[15] R. Sen and Y. N. Srikant. 2007. *Executable Analysis with Circular Linear Progressions.* Technical Report IISc-CSA-TR-2007-3. Computer Science and Automation Indian Institute of Science.

[16] Gregory Stock, Sebastian Hahn, and Jan Reineke. 2019. Cache Persistence Analysis: Finally Exact. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019.* IEEE, 481–494. https://doi.org/10.1109/RTSS46320.2019.00049

[17] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (2007), 37–66. https://doi.org/10.1007/s00453-006-1231-0

[18] S. Wegener. 2012. Computing Same Block Relations for Relational Cache Analysis. In *WCET'12*, Vol. 23.

[19] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. 1997. Timing Analysis for Data Caches and Set-Associative Caches. In *IEEE Real-Time and Embedded Technology and Applications Symposium.*

[20] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. 1999. Timing Analysis for Data and Wrap-Around Fill Caches. *Real-Time Systems* 17 (1999).