Warping Cache Simulation of Polyhedral Programs

Anonymous Author(s)

Abstract

Techniques to evaluate a program's cache performance fall into two camps: 1. Traditional trace-based cache simulators precisely account for sophisticated real-world cache models and support arbitrary workloads, but their runtime is proportional to the number of memory accesses performed by the program under analysis. 2. Relying on implicit workload characterizations such as the polyhedral model, analytical approaches often achieve problem-size-independent runtimes, but so far have been limited to idealized cache models.

We introduce a hybrid approach, warping cache simulation, that aims to achieve applicability to real-world cache models and problem-size-independent runtimes. As prior analytical approaches, we focus on programs in the polyhedral model, which allows to reason about the sequence of memory accesses analytically. Combining this analytical reasoning with information about the cache behavior obtained from explicit cache simulation allows us to soundly fast-forward the simulation. By this process of warping, we accelerate the simulation so that its cost is often independent of the number of memory accesses.

1 Introduction

Traditionally, the efficiency of an algorithm has been determined by evaluating its time complexity. Today, evaluating an algorithm's cache performance has become equally important. Over the past thirty years, the increasing processor-memory gap has led to the introduction of complex memory hierarchies consisting, in particular, of multiple cache levels. As a consequence, a program's runtime on modern hardware heavily depends on how well it exploits the underlying memory hierarchy. However, unlike time complexity, cache performance cannot easily be gauged in a compositional manner from a program's parts, i.e., the composition of two cache-efficient parts may be cache inefficient, and vice versa.

This calls for automatic methods to evaluate a program's cache performance, to inform programmers and compilers so that they can make informed choices about data-locality transformations. Cache performance analysis has already received considerable attention. Prior work can roughly be divided into two camps:

1. *Traditional cache simulators*, such as Dinero IV [18] or CASPER [34], simulate a program's cache behavior by explicitly iterating over the trace of memory accesses generated by the program. The advantage of this approach is that it is applicable to arbitrary workloads and it is possible to precisely model modern memory hierarchies, including sophisticated cache replacement policies, such as Pseudo-LRU [3] or Quad-age LRU [35, 36] found in real-world microarchitectures [2, 53]. The main drawback of traditional simulators is that their runtime is *proportional to the number of memory accesses* a program performs. As a consequence, the simulation of programs operating on large amounts of data may take weeks or more.

2. *Analytical cache models* [7, 11, 14, 15, 23, 24, 32, 47, 48], on the other hand, e.g. PolyCache [7] or HayStack [32], aim to achieve analysis times that are *independent of the number of memory accessed* performed by the program under analysis. To this end, they rely on program representations that implicitly represent a program's memory accesses. A prominent such program representation is the *polyhedral model* [10, 19], which, loosely speaking, captures a program's memory accesses as polyhedra. For such programs, the number of cache misses can be obtained analytically by applying a sequence of algebraic operations on the program representation and by applying symbolic counting techniques. One main drawback of these analytical models is that they are limited to simplified cache models: HayStack [32] applies to inclusive hierarchies of fully-associative caches with least-recently-used (LRU) replacement; PolyCache [7] applies to hierarchies of set-associative caches but is also limited to LRU replacement and can handle non-write allocate caches only approximately.

In this paper, we introduce a new approach called *warping cache simulation* that aims to combine the strengths of traditional cache simulators and analytical cache models. Warping cache simulation is applicable to realistic models of modern memory hierarchies, supporting hierarchical caches with various write policies and arbitrary replacement policies, and its runtime is often independent of the number of memory accesses of a program.

At its core, warping exploits the following data-independence property of caches: Assume c_1 and c_2 are two cache states that are equal up to a renaming of the addresses of the cached memory blocks, i.e., there is a bijection π mapping the memory blocks of c_1 to those of c_2, such that $\pi(c_1) = c_2$. Then, an access to c_1 under block b is a cache hit if and only if $\pi(b)$ hits in state c_2. Similarly, the resulting cache states c_1', c_2' under accesses to b and $\pi(b)$ are guaranteed to be related to each other under the same bijection π.

Let us illustrate how we can exploit this data-independence property in warping cache simulation at the hand of the 1D stencil computation in Figure 1, which will serve as a running example throughout the paper. In the example, we assume a

PL'18, January 01–03, 2018, New York, NY, USA
2018.

```
for (int i = 1; i < 999; i++)
    B[i-1] = A[i-1] + A[i];
```

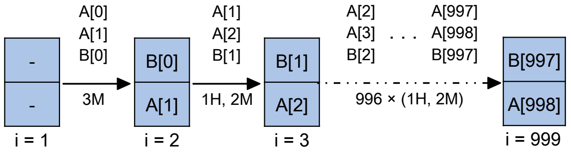

Figure 1. 1D stencil computation and its warping simulation.

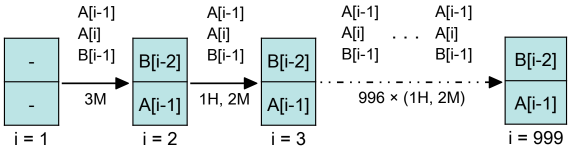

Figure 2. Symbolic warping cache simulation.

small fully-associative cache of size two with least-recently-used (LRU) replacement, but the approach equally applies to more complex real-world caches. In each loop iteration, the program accesses $A[i]$, $A[i-1]$ and $B[i-1]$. Thus, after the first loop iteration, which results in three misses, $B[0]$ and $A[1]$ are cached. All subsequent iterations will hit on the access to $A[i-1]$ because it was cached in the previous iteration. Iteration i results in a cache state containing $B[i-1]$ and $A[i]$. Thus cache states in consecutive iterations are related under the simple bijection that maps memory block i to memory block $i+1$. Warping cache simulation detects this relation. Then, it checks whether the future memory accesses relate to the past memory accesses in the same way that the matching cache states relate to each other. If that is the case, the simulation may fast forward, potentially all the way to the end of the loop, determining the resulting number of cache misses and the final cache state analytically. In our example, warping simulation fast forwards through the entire loop after explicitly simulating the loop for two iterations.

To make this basic idea a reality, we introduce symbolic cache simulation to efficiently determine whether two cache states encountered during concrete simulation are related under a bijection. Figure 2 illustrates the symbolic cache simulation of the 1D stencil computation. In our example, symbolic cache simulation determines that the cache states obtained after the first and the second iteration both contain $A[i]$ and $B[i-1]$ for different values of the loop iterator i. Hashing the symbolic cache states obtained in different iterations allows to efficiently detect such a match, also across several iterations. Further, we employ polyhedral techniques to check whether future memory accesses satisfy the warping conditions implied by matching cache states obtained during simulation. We have implemented our approach and applied it to the PolyBench [41] benchmark suite. Our experiments show that warping cache simulation may outperform traditional cache simulation by several orders of magnitude.

To summarize, we make the following contributions:

- We introduce *warping cache simulation*, the first approach that is both applicable to real-world cache architectures and may achieve simulation times that are independent of the number of memory accesses performed by the program.
- We implement warping cache simulation and experimentally evaluate its performance, demonstrating that warping cache simulation may outperform traditional cache simulation by several orders of magnitude.

2 Caches and Data Independence

Caches are fast but small memories that buffer parts of the large but slow main memory in order to bridge the speed gap between the processor and main memory. Caches operate at the granularity of memory blocks $b \in Block$, which are stored in the cache in *cache lines* of the same size. In order to facilitate an efficient cache lookup, the cache is organized into *sets* such that each memory block maps to a unique cache set. The size k of a cache set is called the *associativity* of the cache. If an accessed block resides in the cache, the access *hits* the cache. Upon a cache *miss*, the block is loaded from main memory, and, if the corresponding cache set is full, another memory block is evicted to make place for the newly loaded block. The block to evict is determined by the *replacement policy*.

To ease the formal development, we first formalize the behavior of individual cache sets, which is already sufficient to capture fully-associative caches; and then generalize this formalization to set-associative caches.

2.1 Cache Sets

The state of an individual cache set in a cache of associativity k is a pair

$$s \in SetState = (Line \to (Block \cup \{\epsilon\})) \times PolicyState,$$

where $Line = \{1, \ldots, k\}$, and thus the first component of the pair captures the memory blocks stored in the k cache lines of the cache set. Empty lines are represented by ϵ. The state of many replacement policies can be fully captured by the order in which memory blocks occupy a set's cache lines. Examples of such policies are least-recently-used (LRU), first-in first-out (FIFO), and Pseudo-LRU (PLRU) [1]. For such policies, the *PolicyState* component of cache states can be omitted. E.g. under LRU, cache lines are ordered from most- to least-recently-used; under FIFO they are ordered from last- to first-in. To model more complex policies, such as Quad-age LRU [35, 36], the *PolicyState* component is used to capture additional state of the replacement policy. Given a set state $s = (m, ps)$, we refer to the mapping of s by $s.m$ and to the policy state of s by $s.ps$.

Note that this model does not include the data stored in the cache set as this is not relevant to determine whether a memory access results in a hit or a miss. For simplicity, we also do not differentiate between reads and writes, which may make a difference depending on the write policy. Thus the formalization applies to write-allocate caches, but our implementation also supports no write-allocate caches.

We may model the effect of a memory access on the cache state using the two functions $UpSet : SetState \times Block \to SetState$ and $ClSet : SetState \times Block \to \mathbb{B}$, which take as input a set state and the accessed memory block and return the updated set state and the access's classification as a hit or a miss, respectively.

The definition of *UpSet* depends on the particular replacement policy. For LRU, e.g., it is defined as follows:

$$UpSet_{LRU}(s, b) := \lambda l \in Line. \begin{cases} b & : \mathit{if}\ l = 1 \\ s(l) & : \mathit{if}\ \exists l' < l : s(l') = b \\ s(l-1) & : \mathit{otherwise} \end{cases}$$

Our approach is applicable to *any* replacement policy as long as it satisfies the data-independence property we will define shortly. In contrast, *ClSet* can be defined generically by inspecting the contents of the cache lines:

$$ClSet(s, b) := \begin{cases} \mathit{true} & : \mathit{if}\ \exists i : s.m(i) = b \\ \mathit{false} & : \mathit{otherwise} \end{cases} \tag{1}$$

Let $\Pi \subset Block \to Block$ be the set of bijections from memory blocks to memory blocks. A bijection $\pi \in \Pi$ can be applied to a set state as follows:

$$\pi(s) := (\lambda l.\pi(s.m(l)), s.ps),$$

where we define $\pi(\epsilon) = \epsilon$. In other words, the bijection is applied to the contents of each cache line, mapping empty lines to empty lines.

Property 1 (Data independence of cache sets). *Let $s \in SetState$, $b \in Block$, and $\pi \in \Pi$. Then:*

$$\pi(UpSet(s, b)) = UpSet(\pi(s), \pi(b)) \tag{2}$$

In other words, the cache update is independent of the particular memory blocks stored in a cache set. To simplify the following statements, we do not restate Property 1 in the remainder of the paper, but implicitly assume it holds.

All cache architectures we are aware of satisfy Property 1. Recent measurement-based approaches [1, 2, 53] to automatically derive cache models are also naturally limited to models satisfying data independence. Our warping cache simulator supports LRU, FIFO, PLRU [3], and Quad-age LRU [35, 36], which allows to model the L1 and L2 caches of most recent Intel microarchitectures [2, 53]. Other policies can be added as long as they satisfy data independence.

2.2 Set-associative Caches

Set-associative caches can be seen as the composition of multiple cache sets. Typically the number of cache sets s is a power of two, so that the cache set that a memory block maps to is determined by a subset of its address, which is commonly referred to as the cache index.

In the following, we model the mapping from memory blocks into cache sets using the function $index : Block \to Set$, where $Set = \{0, \ldots, s-1\}$. Most real-world caches employ a modulo mapping of blocks to cache sets, i.e., $index(b) = b \bmod s$. Then, the state of a set-associative cache can be captured simply as a mapping from cache sets to their states: $c \in CacheState = Set \to SetState$.

A memory access results in a hit if it hits in the cache set that it maps to:

$$ClCache(c, b) := ClSet(c(index(b)), b) \tag{3}$$

Cache states are updated by updating the cache set the block maps to:

$$UpCache(c, b) := c[index(b) \mapsto UpSet(c(index(b)), b)] \tag{4}$$

Thus, cache sets are updated independently of each other, which implies another source of symmetry we seek to exploit. To this end, let $\Pi_{index_=}$ be the set of bijections on blocks that preserve the partition of blocks into cache sets:

$$\begin{aligned} \Pi_{index_=} := \{\pi \in \Pi \mid \forall b, b' \in Block : (index(b) = index(b')) \\ \Leftrightarrow (index(\pi(b)) = index(\pi(b')))\} \end{aligned}$$

A bijection $\pi \in \Pi_{index_=}$ induces a bijection π_{Set} on cache sets:

$$\pi_{Set} := \{(index(b), index(\pi(b))) \mid b \in Block\}.$$

This allows to apply bijections from $\Pi_{index_=}$ to cache states:

$$\pi(c) := \lambda s.\pi(c(\pi_{Set}^{-1}(s))) \tag{5}$$

Assuming Property 1 holds on the underlying cache sets, it also holds for the resulting set-associative cache:

Theorem 1 (Data independence of caches). *Let $c \in CacheState$, $b \in Block$, and $\pi \in \Pi_{index_=}$. Then:*

$$\pi(UpCache(c, b)) = UpCache(\pi(c), \pi(b)), \tag{6}$$

$$ClCache(c, b) = ClCache(\pi(c), \pi(b)). \tag{7}$$

The proof of this theorem and all other proofs are given in the supplementary material. There, we also show that data independence also applies to two-level cache hierarchies.

Example 2.1. Let us illustrate Theorem 1 at the hand of the 1D stencil code from Figure 1. Assume a set-associative cache consisting of four cache sets of associativity two with LRU replacement. Further assume, as in the previous examples, that each array cell occupies one full cache line and that the index of both $A[0]$ and $B[0]$ is zero. Then, the execution reaches cache state c_5 in Figure 3 at the start of loop iteration 5. In the figure, the cache lines within each cache set are

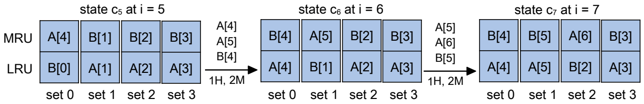

Figure 3. Example illustrating the data independence of set-associative caches.

ordered from most-recently-used (MRU) to least-recently-used (LRU). Performing the accesses of iteration 5 yields cache state c_6, with $c_6 = \pi(c_5)$, where $\pi(i) = i + 1$, and thus $\pi_{Set}(s) = (s+1) \bmod 4$. As the accesses in iteration 6 relate to those of the iteration 5 under the same bijection π, following Theorem 1, the next state can be obtained as $c_7 = \pi(c_6)$.

3 Polyhedral Program Representation

The polyhedral model [10, 19, 20] is a mathematical framework to succinctly describe and manipulate programs' control flow and data-access patterns using Presburger arithmetic [33]. In this section, we introduce a simple program representation resembling abstract syntax trees that is tailored to cache simulation.

3.1 Presburger Sets and Maps

To manipulate integer sets and to represent programs in the polyhedral model, we make use of *isl*, the *integer set library* [49]. Here, we introduce how integer sets can be defined and some important operations on integer sets provided by *isl*. Our presentation loosely follows the tutorial by Verdoolaege [50]. More details can be found there.

A *Presburger set* $S = \{(i_1, \ldots, i_n) \mid c\}$ is an integer set, i.e., a set of integer tuples $(i_1, \ldots, i_n) = \vec{i} \in \mathbb{Z}^n$, whose elements satisfy the Presburger formula c. The only free variables allowed in c are $i_1, \ldots, i_n$, so that the set S corresponds to the satisfying assignments of c.

Presburger formulas are first-order formulas that are limited to the Presburger language, which allows for addition $+$, subtraction $-$, integer constants d, floored division by integer constants $\lfloor \cdot / d \rfloor$ and the binary predicate $\leq$.

Example 3.1. The set $E = \{(i, j) \mid \exists k : \lfloor i/7 \rfloor = k+k \wedge i \leq j\}$ consists of all pairs of integers (i, j), s.t. the floored division of i by seven is even and for which $i \leq j$.

To ease notation, several other operations are supported as syntactic sugar, in particular multiplication by constants, modulo with a constant divisor, and comparison of integer tuples by lexicographic ordering $\preceq$. It is also convenient to refer to previously defined Presburger sets within the definition of a new set: $F = \{(i, j) \mid \exists k : (i, k) \in E \wedge j+j = k\}$.

A *Presburger relation* $R = \{(i_1, \ldots, i_n) \to (j_1, \ldots, j_m) \mid c\}$ relates integer tuples $\vec{i} \in \mathbb{Z}^n$ to integer tuples $\vec{j} \in \mathbb{Z}^m$, where the constraint c has the same restrictions as in the case of Presburger sets. For a Presburger relation R, R_{dom} denotes its domain, i.e., $R_{dom} = \{\vec{i} \mid \exists \vec{j} : \vec{i} \to \vec{j} \in R\}$. Dually, R_{ran} denotes its range, i.e., $R_{ran} = \{\vec{j} \mid \exists \vec{i} : \vec{i} \to \vec{j} \in R\}$. We express the lexicographic minimum and maximum of sets as $lexmin(S) = \{\vec{i} \mid \vec{i} \in S \wedge \nexists \vec{j} \in S : \vec{j} \prec \vec{i}\}$ and $lexmax(S) = \{\vec{i} \mid \vec{i} \in S \wedge \nexists \vec{j} \in S : \vec{i} \prec \vec{j}\}$, respectively.

Given the definition of a set or relation we may use *isl* to check whether the set if empty, and, if not, to extract an element of this set or relation.

```
for (int i = 0; i < 100; i++) {
  c[i] = 0;
  for (int j = i; j < 100; j++) {
    c[i] = c[i] + A[i][j] * x[j];
  }
}
```

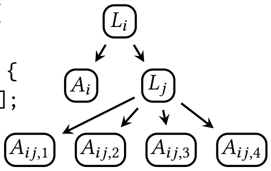

Figure 4. Computation of the product of an upper triangular matrix with a vector and its tree representation.

3.2 Static Control Parts

Warping cache simulation applies to *static control parts* (SCoPs) of programs. SCoPs are loop nests whose control flow and memory-access behavior is determined statically, and thus independent of the program's inputs. Further restrictions apply to the loop bounds and the array index expressions, which are limited to affine expressions. We also refer to such loops as *polyhedral programs*.

For the purpose of cache simulation, it is sufficient to capture a SCoP's memory-access behavior. Thus, we can safely abstract from the computations performed by a SCoP. To this end, we introduce a tree-structured representation for SCoPs, which resembles the programs abstract syntax tree. This tree representation consists of two types of nodes:

1. *Loop nodes* correspond to loops in the source program.
2. *Access nodes* form the leaves of the tree and correspond to the array accesses[1] performed by the program.

Each loop node L has the following attributes:

- An *iteration domain* $L.dom$ that captures the values of loop iterators for which the loop is executed. The dimensionality of $L.dom$ depends on the nesting level of the loop node: The root node's domain is one dimensional, and each nesting level adds one dimension, corresponding to the loop iterator of the loop node.
- A list of children $L.children$. Children are either loop nodes or access nodes. Their order defines the order in which the children are to be visited during simulation.

By construction, the iteration domain of each loop is traversed in lexicographic order.

Each access node A has the following attributes:

- An *iteration domain* $A.dom$ that captures the loop iterations in which the access should be performed. This is required to model memory accesses that are guarded by a conditional within a loop.
- An *access function* $A.access$ that determines the accessed memory block for each access instance.

[1] If necessary, scalar variables can be modeled as zero-dimensional arrays.

Algorithm 1 Non-warping cache simulation

```
procedure LoopNode::Simulate($\vec{j}$)
    $\vec{i} \leftarrow this.initial(\vec{j})$
    $\vec{final} \leftarrow this.final(\vec{j})$
    while $\vec{i} \preceq \vec{final}$ do
        if $\vec{i} \in this.dom$ then
            for all $child \in this.children$ do
                $c \leftarrow child.\textsc{Simulate}(\vec{i})$
        $\vec{i} \leftarrow \vec{i} + this.\vec{stride}$
procedure AccessNode::Simulate($\vec{j}$)
    if $\vec{j} \in this.dom$ then
        $c \leftarrow UpCache(c, this.access(\vec{j}))$
        $m \leftarrow m + ClCache(c, this.access(\vec{j}))$
```

Consider the example program implementing a matrix-vector product computation and its tree representation in Figure 4. Child nodes are sorted from left to right, corresponding to the execution order.

In our example, the iteration domains are

$$
\begin{aligned}
L_i.dom = A_i.dom &= \{(i) \in \mathbb{Z}^1 \mid 0 \le i < 100\},\\
L_j.dom &= \{(i,j) \in \mathbb{Z}^2 \mid 0 \le i < 100 \wedge i \le j < 100\}\\
&= A_{ij,1}.dom = A_{ij,2}.dom = A_{ij,3}.dom = A_{ij,4}.dom.
\end{aligned}
$$

As none of the access nodes are guarded by a conditional, their iteration domains are equal to those of their enclosing loops. The access functions are

$$
\begin{aligned}
A_i.access &= \{(i) \rightarrow block(linearize(c[i]))\},\\
A_{ij,1}.access &= \{(i,j) \rightarrow block(linearize(c[i]))\},\\
A_{ij,2}.access &= \{(i,j) \rightarrow block(linearize(A[i][j]))\},\\
A_{ij,3}.access &= \{(i,j) \rightarrow block(linearize(x[j]))\},\\
A_{ij,4}.access &= \{(i,j) \rightarrow block(linearize(c[i]))\},
\end{aligned}
$$

where $linearize(\cdot)$ converts an array expression into an expression capturing the accessed memory address. E.g. assuming a row-major layout and an array $A[23][42]$ of 4-byte integers, $linearize(A[i][j]) = start_A + 42 \cdot 4i + 4j$. As caches operate at the granularity of memory blocks, *block* translates the accessed address into the corresponding memory block, i.e., $block(x) := \lfloor x/64 \rfloor$ assuming a block size of 64 bytes.

We use *pet*, the *Polyhedral Extraction Tool* [51] to obtain polyhedral representations of SCoPs, which we subsequently transform into the tree representation introduced above. For convenience during simulation, we also define the following helper functions, which can be implemented using *isl*: $L.initial(\vec{j}) := lexmin(L.dom \cap (\{\vec{j}\} \times \mathbb{Z}))$, and $L.final(\vec{j}) := lexmax(L.dom \cap (\{\vec{j}\} \times \mathbb{Z}))$, which provide the smallest and the largest elements of the iteration domain of L for a fixed assignment of the first $n-1$ dimensions of the iteration domain. In addition, $interval(\vec{i}, \vec{j}) := \{\vec{k} \mid \vec{i} \preceq \vec{k} \prec \vec{j}\}$ captures the set of integers in the interval between $\vec{i}$ and $\vec{j}$. Similarly, we may extract the *stride* $L.\vec{stride}$ of a loop node, which is the increment of the iteration variable of loop node L.

4 Non-Warping Cache Simulation of Polyhedral Programs

Algorithm 1 shows how to perform non-warping cache simulation on top of the tree representation introduced in the previous section. The algorithm uses two global variables, c and m, the current cache state and the current cache miss count.

To analyze a SCoP, the simulation is initiated by invoking the Simulate procedure of the root node of the tree. The first parameter, $\vec{j}$, is the state of those loop iterators that are defined in ancestors of a node. Thus, at the top level, the zero-dimensional tuple $\vec{j} = ()$ can be passed to the procedure.

For a loop node, the simulator steps through the iteration domain from the initial state $this.initial(\vec{j})$ to the final state $this.final(\vec{j})$. At each point in the iteration domain the simulation of all child nodes is triggered.

Memory accesses are simulated at access nodes. If the current iterator state $\vec{j}$ is in the access's domain, the cache state is updated and the cache miss count is incremented based on the classification of the memory access associated with the current iterator state $\vec{j}$.

As the SCoP cache simulation may be initiated with any cache state and any cache miss count, the SCoP simulation could be integrated into more general simulation frameworks that apply to non-static control parts of a program. This also applies to the warping cache simulation that we introduce in the following section. However, experimental evaluation of such an integration is outside of the scope of this paper due to the significant required required engineering effort.

5 Warping Cache Simulation of Polyhedral Programs

We now show how to exploit the data independence of caches in order to speed up cache simulation. The basic idea is to identify recurring patterns of cache states and memory accesses during the cache simulation and to "warp" across these. In Section 5.1 we introduce the warping theorem that formalizes the above idea. To efficiently determine candidates for warping, we introduce symbolic cache simulation and a corresponding symbolic warping theorem in Section 5.2. Based on these foundations we finally introduce a warping symbolic cache simulation algorithm in Section 5.3.

5.1 Concrete Cache Warping

Warping is based on the following theorem, which follows from Theorem 1:

Theorem 2 (Cache warping). *Let $c_0, c_1 \in CacheState$, $s_0, \ldots, s_n \in Block^*$, and $\pi \in \Pi_{index_=}$, s.t.*

$$c_1 = UpCache(c_0, s_0) = \pi(c_0), \tag{8}$$

$$\forall i, 0 \le i < n : s_{i+1} = \pi(s_i). \tag{9}$$

Then:

$$\mathit{UpCache}(c_1, s_1 \circ \cdots \circ s_n) = \pi^n(c_1), \tag{10}$$

$$\mathit{ClCache}(c_1, s_1 \circ \cdots \circ s_n) = n \cdot \mathit{ClCache}(c_0, s_0), \tag{11}$$

where $\circ$ *denotes the concatenation operator on access sequences.*

In other words, if, during the simulation we arrive at cache state c_1 that is equal up to a bijection on the cache contents to an earlier cache state c_0, i.e., $c_1 = \pi(c_0)$, and the subsequent memory accesses correspond to those observed between c_0 and c_1 under the same bijection π, then warping can be applied, and the final cache state can be computed directly from c_1 solely based on the bijection π. Depending on the structure of π, π^n can be computed efficiently, e.g. if π corresponds to shifting all addresses by a constant.

Example 5.1. Consider again our running example from Figure 1 and its concrete simulation on a set-associative cache in Figure 3. After reaching cache state c_6 and observing that $c_6 = \pi(c_5)$ with $\pi(i) = i+1$, we can apply Theorem 2 to obtain cache state c_{999} at the end of the stencil computation as $c_{999} = \pi^{993}(c_6)$. Also, the number of misses on the remaining 993 iterations of the loop can be determined as $2 \cdot 993$.

Thus at a high level, a warping-based simulation algorithm could proceed as follows: 1. Simulate cache accesses concretely, until a cache state is obtained that satisfies the conditions of Theorem 2. 2. Analyze the "future" memory accesses to determine up to which point Theorem 2 can be applied. 3. Continue at 1.

A naive implementation would compare each cache state to all cache states encountered before. This would be highly inefficient. To more efficiently determine matching cache states, our simulator instead operates on *symbolic cache states*. Symbolic cache states express the concrete cache state in terms of the iterator state. Whenever the simulation reaches a symbolic cache state that is the *equal* to a symbolic cache state encountered before, this implies that the corresponding concrete cache states are related by a bijection; and this bijection can be extracted efficiently from the symbolic cache states. Equality of symbolic cache states can be detected efficiently via hashing.

5.2 Symbolic Simulation and Symbolic Warping

To efficiently determine candidate pairs of matching states, we introduce *symbolic cache states*. In place of concrete memory blocks, symbolic cache sets and symbolic cache states associate cache lines with *symbolic memory blocks*:

$$\begin{aligned} \mathit{sym\text{-}s} \in \mathit{SymSetState} = (\mathit{Line} \longrightarrow (\mathit{SymBlock} \cup \{\epsilon\})) \\ \times \mathit{PolicyState}, \\ \mathit{sym\text{-}c} \in \mathit{SymCacheState} = \mathit{Set} \longrightarrow \mathit{SymSetState}. \end{aligned}$$

Symbolic memory blocks correspond to the access functions of access nodes in our SCoP representation. Thus symbolic memory blocks represent functions that map the state of the loop iterators to concrete memory blocks. Due to the restriction to the polyhedral model the expressions used to represent symbolic memory blocks are always of the form $\lfloor e/c \rfloor$, where c is a constant corresponding to the block size and e is an affine expression in the loop iterators. In the following we assume an interpretation function $[\![\cdot]\!]$ that maps symbolic memory blocks to the functions they represent.

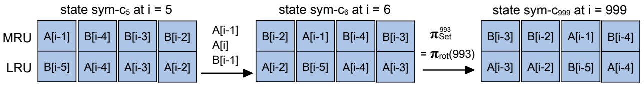

Figure 5. Example illustrating symbolic equivalence and symbolic cache warping.

Example 5.2. Consider an access $A[i][j]$. As discussed in Section 3.2, this access would be associated with an expression representing the function

$$\lambda(i, j).\mathit{block}(\mathit{linearize}(A[i][j])) = \lfloor (\mathit{start}_A + 42 \cdot 4i + 4j)/64 \rfloor.$$

Symbolic memory blocks, and by extension symbolic cache states, represent different concrete cache states depending on the state of the loop iterators. Symbolic cache simulation thus maintains a pair of the current symbolic cache state and the current loop iterator. Such a pair $(\mathit{sym\text{-}c}, \vec{i})$ then concretizes to a concrete cache state by replacing each symbolic memory block by the concrete block it represents under $\vec{i}$:

$$\begin{aligned} \gamma(\mathit{sym\text{-}c}, \vec{i}) &:= \lambda s.\gamma_{Set}(\mathit{sym\text{-}c}(s), \vec{i}), \\ \gamma_{Set}(\mathit{sym\text{-}s}, \vec{i}) &:= (\lambda l.[\![\mathit{sym\text{-}s.m}(l)]\!](\vec{i}), \mathit{sym\text{-}s.ps}). \end{aligned}$$

Symbolic cache states can be updated and accesses can be classified so that the following equalities hold:

$$\begin{aligned} &\gamma(\mathit{SymUpCache}((\mathit{sym\text{-}c}, \vec{i}), \mathit{sym\text{-}b})) = \\ &\qquad \mathit{UpCache}(\gamma(\mathit{sym\text{-}c}, \vec{i}), [\![\mathit{sym\text{-}b}]\!](\vec{i})), \\ &\mathit{SymClCache}((\mathit{sym\text{-}c}, \vec{i}), \mathit{sym\text{-}b}) = \\ &\qquad \mathit{ClCache}(\gamma(\mathit{sym\text{-}c}, \vec{i}), [\![\mathit{sym\text{-}b}]\!](\vec{i})). \end{aligned} \tag{12}$$

For convenience, *SymUpCache* returns both the updated symbolic cache state and the state of the loop iterators. A constructive definition of *SymUpCache* achieving the above equality is given in the supplementary material. This allows our simulation to operate on symbolic cache states in place of concrete ones.

As changes to the loop iterators result in a different concretization of symbolic cache states, these have to be adapted upon any increment Δ of the loop iterators[2]. Appropriately adapting the expressions forming a symbolic cache state allows for an update function *SymUpCache* that satisfies the

[2] Our implementation determines the updated symbolic cache state only on demand, which significantly increases efficiency.

following equality:

$$\begin{aligned} SymUpCache((sym\text{-}c, \vec{i}), \Delta) &= (sym\text{-}c', \vec{i} + \Delta), \\ \gamma(sym\text{-}c', \vec{i} + \Delta) &= \gamma(sym\text{-}c, \vec{i}). \end{aligned} \tag{13}$$

Symbolic cache states are useful to detect opportunities for warping:

Theorem 3 (Symbolic equivalence of cache states)**.** *Let $(sym\text{-}c_0, \vec{i}_0)$, $(sym\text{-}c_1, \vec{i}_1) \in SymCacheState \times \mathbb{Z}^n$ and π_{Set} be a bijection on cache sets, s.t.*

$$sym\text{-}c_1 = sym\text{-}c_0 \circ \pi_{Set}.$$

Then there is a $\pi \in \Pi_{index_=}$, s.t.:

$$\gamma(sym\text{-}c_1, \vec{i}_1) = \pi(\gamma(sym\text{-}c_0, \vec{i}_0)). \tag{14}$$

In other words, if the simulation determines two symbolic cache states that are equal up to a permutation of their cache sets, then their concrete counterparts are also related to each other by a bijection. To further simplify the search for matches, in our implementation, we are not looking for arbitrary permutations, but only for *rotations*, i.e., permutations of the form $\pi_{rot}(c) = \{(i, i + c \bmod s) \mid i \in Set\}$.

Example 5.3. Consider the symbolic cache states $sym\text{-}c_5$ and $sym\text{-}c_6$ in Figure 5, which are the symbolic counterparts to the concrete cache states c_5 and c_6 from Figure 3. We have $c_6 = \pi_{rot}(1)(c_5)$, and thus the two states are symbolically equivalent, which by Theorem 3 implies that their concretizations c_5 and c_6 are related by a bijection.

Having found two symbolic cache states $sym\text{-}c_1$, $sym\text{-}c_2$ that "match" does not immediately guarantee that warping can be applied. This also depends on the accesses between $sym\text{-}c_1$ and $sym\text{-}c_2$ and their relation to the subsequent accesses. The following symbolic warping theorem captures a sufficient condition for warping on symbolic cache states:

Theorem 4 (Symbolic cache warping)**.** *Let $(sym\text{-}c_0, \vec{i}_0)$, $(sym\text{-}c_1, \vec{i}_1) \in SymCacheState \times \mathbb{Z}^n$, and π_{Set} be a bijection on cache sets, and $\sigma \in SymBlock^*$, such that:*

$$\begin{aligned} SymUpCache((sym\text{-}c_0, \vec{i}_0), \sigma) &= (sym\text{-}c_1, \vec{i}_1) \\ &= (sym\text{-}c_0 \circ \pi_{Set}, \vec{i}_1), \end{aligned} \tag{15}$$

and let $\pi \in \Pi_{index_=}$, such that for all $j, 0 \leq j < n$:

$$\gamma(\sigma, \vec{i}_{j+1}) = \pi(\gamma(\sigma, \vec{i}_j)), \tag{16}$$

$$\gamma(sym\text{-}c_0 \circ \pi_{Set}^{j+1}, \vec{i}_{j+1}) = \pi(\gamma(sym\text{-}c_0 \circ \pi_{Set}^{j}, \vec{i}_j)), \tag{17}$$

with $\vec{i}_j = \vec{i}_0 + j \cdot (\vec{i}_1 - \vec{i}_0)$ for $0 \leq j \leq n + 1$. Then:

$$\begin{aligned} &UpCache(\gamma(sym\text{-}c_1, \vec{i}_1), \gamma(\sigma, \vec{i}_1) \circ \cdots \circ \gamma(\sigma, \vec{i}_n)) = \\ &\qquad\qquad \gamma(sym\text{-}c_1 \circ \pi_{Set}^n, \vec{i}_{n+1}), \end{aligned} \tag{18}$$

$$\begin{aligned} &ClCache(\gamma(sym\text{-}c_1, \vec{i}_1), \gamma(\sigma, \vec{i}_1) \circ \cdots \circ \gamma(\sigma, \vec{i}_n)) = \\ &\qquad\qquad n \cdot SymClCache((sym\text{-}c_0, \vec{i}_0), \sigma). \end{aligned} \tag{19}$$

Let's digest this theorem to better understand when and how it is applicable. Equation (15) captures that the matching symbolic cache states need to be equal up to a permutation of their cache sets, and $sym\text{-}c_1$ needs to be obtained from $sym\text{-}c_0$ on the symbolic access sequence σ. Regarding, (18) observe that if all conditions apply, we may warp across n copies of the *same* symbolic access sequence σ. Note, however, that the corresponding n concrete sequences can be different as they concretize under different iterator valuations, e.g. if σ corresponds to $A[i]; i{+}{+}$ this would admit warping across potentially many iterations of a loop traversing an array. The benefit of $sym\text{-}c_1 = sym\text{-}c_0 \circ \pi_{Set}$ is that warping the symbolic state is achieved by simply applying π_{Set}^n to $sym\text{-}c_1$. If π_{Set} is a rotation $\pi_{rot}(c)$ with offset c, then $\pi_{Set}^n = \pi_{rot}(n \cdot c)$.

Once a match is found, checking applicability requires ensuring that (16) and (17) hold. Here, (16) ensures that the concrete sequences corresponding to the n copies of σ relate to each other under the same bijection π. In practice, (16) usually holds, but is not guaranteed to in general. Consider e.g. the symbolic sequence corresponding to $A[i]; A[2 \cdot i]; i{+}{+}$. For many choices of i and n an appropriate bijection π satisfying (16) exists, but for $i = 0$ and $n = 2$ this is impossible, as $A[0]; A[2 \cdot 0]$ and $A[1]; A[2 \cdot 1]$ cannot be transformed into each other by any bijection. Finally, condition (17) ensures that the bijection π is also compatible with the matching symbolic cache states, and their warped versions.

Example 5.4. In our running example, all conditions of Theorem 4 hold and thus the final cache state of the loop can be obtained by applying $\pi_{rot}^{993}(1) = \pi_{rot}(993 \cdot 1 \bmod 4) = \pi_{rot}(1)$ to $sym\text{-}c_6$ as shown in the top right corner of Figure 5.

5.3 Warping Symbolic Cache Simulation

Let us now explain the warping symbolic cache simulation algorithm in Algorithm 2, which is based upon Theorem 4 and applies polyhedral techniques to ensure applicability of the theorem for warping.

The algorithm differs from Algorithm 1 in two ways: 1. it applies symbolic rather than concrete cache simulation, and 2. it applies warping.

To find matching symbolic cache states, each loop node maintains a separate hash map in which it stores symbolic cache states reached since the last change to an iterator of an enclosing loop. Thus warping is only attempted across different iterations of a loop while staying in the same iteration of all enclosing loops. This is a deliberate choice that slightly reduces the ability to warp but greatly reduces "spurious" matches that do not result in actual warping opportunities.

The hash value of a symbolic cache state is determined based on the symbolic memory blocks in the cache. To identify "rotating" matches, the hash computation does not begin at a fixed set, but rather starts at the most-recently-accessed cache set and from there on cycles around the cache sets. Thus, when a match is determined, the difference between

Algorithm 2 Warping symbolic cache simulation

```
 1: procedure LoopNode::WarpingSimulate($\vec{j}$)
 2:     $\vec{i} \leftarrow this.initial(\vec{j})$
 3:     $\vec{final} \leftarrow this.final(\vec{j})$
 4:     $x \leftarrow new\ HashMap()$
 5:     while $\vec{i} \leq \vec{final}$ do
 6:         if $x.contains(sym\text{-}c)$ then
 7:             $(\vec{i}_0, m_0, \pi_{rot}) \leftarrow x.get(sym\text{-}c)$
 8:             $\vec{\Delta} \leftarrow \vec{i} - \vec{i}_0$                          ▷ Match delta
 9:             $n \leftarrow$ IterationsToWarp($sym\text{-}c, \vec{i}_0, \vec{i}, \vec{final}, \vec{\Delta}, \pi_{rot}$)
10:             $\vec{i} \leftarrow \vec{i} + n \cdot \vec{\Delta}$                   ▷ Warp $n \cdot \vec{\Delta}$ iterations
11:             $sym\text{-}c \leftarrow sym\text{-}c \circ \pi^n_{rot}$
12:             $m \leftarrow m + n \cdot (m - m_0)$
13:         $x.put(sym\text{-}c, (\vec{i}, m))$
14:         if $\neg x.contains(sym\text{-}c) \vee n = 0$ then ▷ Could not warp
15:             for all $child \in this.children$ do
16:                 $child$.WarpingSimulate($\vec{i}$)
17:             $(sym\text{-}c, \vec{i}) \leftarrow SymUpCache(sym\text{-}c, this.\vec{stride})$
18: procedure AccessNode::WarpingSimulate($\vec{j}$)
19:     if $\vec{j} \in this.dom$ then
20:         $(sym\text{-}c, \vec{i}) \leftarrow SymUpCache(sym\text{-}c, (this.access, \vec{j}))$
21:         $m \leftarrow m + SymClCache(sym\text{-}c, (this.access, \vec{j}))$
```

```
 1: procedure IterationsToWarp($sym\text{-}c, \vec{i}_0, \vec{i}_1, \vec{final}, \vec{\Delta}, \pi_{rot}$)
 2:     $\vec{i}_{f_c} \leftarrow$ FurthestByDomains($\vec{i}_0, \vec{i}_1, \vec{final}, \vec{\Delta}$)
 3:     $\vec{i}_{f_a} \leftarrow$ FurthestByOverlap($\vec{i}_0, \vec{final}$)
 4:     $\vec{i}_f \leftarrow lexmin(\vec{i}_{f_a}, \vec{i}_{f_c})$
 5:     if CacheAgrees($sym\text{-}c, \vec{i}_0, \vec{i}_1, \vec{i}_f, \vec{\Delta}, \pi_{rot}$) then
 6:         return $lexmax\{n \mid \vec{i}_1 + n \cdot \vec{\Delta} \prec \vec{i}_f\}$
 7:     return 0
 1: procedure FurthestByDomains($\vec{i}_0, \vec{i}_1, \vec{final}, \vec{\Delta}$)
 2:     $I_m \leftarrow interval(\vec{i}_0, \vec{i}_1)$                      ▷ Match interval
 3:     $I_w \leftarrow interval(\vec{i}_1, \vec{final})$                 ▷ Max. warp interval
 4:     $C \leftarrow \emptyset$                                           ▷ Conflict set
 5:     for all $AccessNode\ a \in this.children^*$ do
 6:         $C_a \leftarrow \{\vec{i}_c \mid \neg(\vec{i}_c \in (a.dom \cap I_w) \Leftrightarrow$
 7:                 $(\vec{i}_0 + ((\vec{i}_c - \vec{i}_1) \bmod \vec{\Delta})) \in (a.dom \cap I_m))\}$
 8:         $C \leftarrow C \cup C_a$
 9:     if $C = \emptyset$ then return $\vec{final}$
10:     return $lexmin(C)$
 1: procedure FurthestByOverlap($\vec{i}_0, \vec{final}$)
 2:     $I \leftarrow interval(\vec{i}_0, \vec{final})$                    ▷ Access interval
 3:     $C \leftarrow \emptyset$                                           ▷ Conflict set
 4:     for all $AccessNode\ a, b \in this.children^*$ do
 5:         if $a.access$ and $b.access$ have the same coefficients then
 6:             continue
 7:         $C_{a,b} \leftarrow \{\vec{i} \mid \exists \vec{j}_a \in (a.dom \cap I) : \exists \vec{j}_b \in (B.dom \cap I) :$
 8:                 $a.access(\vec{j}_a) = b.access(\vec{j}_b) \wedge \vec{j}_a \preceq \vec{i} \wedge \vec{j}_b \preceq \vec{i}\}$
 9:         $C \leftarrow C \cup C_{a,b}$
10:     if $C = \emptyset$ then return $\vec{final}$
11:     return $lexmin(C)$
 1: procedure CacheAgrees($sym\text{-}c, \vec{i}_0, \vec{i}_1, \vec{i}_f, \vec{\Delta}, \pi_{rot}$)
 2:     $\pi \leftarrow$ ConstructAccessMapping($\vec{i}_0, \vec{i}_f, \vec{\Delta}$)
 3:     $c_0 \leftarrow \gamma(sym\text{-}c, \vec{i}_0)$
 4:     $c_1 \leftarrow \gamma(sym\text{-}c, \vec{i}_1)$
 5:     for all $set\ s$, $line\ l$ do
 6:         $b_0 \leftarrow c_0(s).m(l)$
 7:         $b_1 \leftarrow c_1(\pi_{rot}(s)).m(l)$
 8:         if $b_0 \in \pi_{dom} \wedge \pi(b_0) \neq b_1$ then return $false$
 9:         if $b_1 \in \pi_{ran} \wedge \pi^{-1}(b_1) \neq b_0$ then return $false$
10:     return $true$
11: procedure ConstructAccessMapping($\vec{i}_0, \vec{i}_f, \vec{\Delta}$)
12:     $I \leftarrow interval(\vec{i}_0, \vec{i}_f)$                      ▷ Access interval
13:     $\pi \leftarrow \emptyset$
14:     for all $AccessNode\ a \in this.children^*$ do
15:         $\pi_a \leftarrow \{(b_1) \rightarrow (b_2) \mid \exists \vec{j} \in (a.dom \cap I) \wedge$
16:                 $b_1 = a.access(\vec{j}) \wedge b_2 = a.access(\vec{j} + \vec{\Delta})\}$
17:         $\pi \leftarrow \pi \cup \pi_a$
18:     return $\pi$
```

the indexes of the most-recently-accessed cache sets of the matching cache states determines the relative rotation.

If a matching cache state is found, IterationsToWarp determines the number of iterations to warp across. Lines 10 to 12 carry out the warping and update the number of misses m. In the worst case $n = 0$ and the simulation needs to proceed via ordinary symbolic cache simulation in lines 14 to 17, which is also applied if there is no match.

The procedure IterationsToWarp relies on three subprocedures to determine how many iterations to warp across:

1. FurthestByDomains determines up to which iteration the future symbolic memory accesses are identical to the symbolic memory accesses in the match interval. This is determined by separately considering the domains of every access node that is a descendant of the warping loop node. The set C_a is constructed such that it contains all iterator valuations that conflict with the corresponding iteration in the match interval, i.e., either the corresponding iteration was present in the match interval but is missing in $\vec{i}_c$ or vice versa. Based on Theorem 4 warping is limited to repetitions of the same symbolic access sequence, and so warping across such conflicts is impossible. Thus the earliest conflict is determined using $lexmin(C)$.

2. To satisfy (16), there needs to be a single bijection π that applies to all accesses in $\gamma(\sigma, \vec{i}_i)$ for all i. These accesses may stem from different access nodes, which may each depend differently on loop iterators. Consider for example two access nodes with access expressions $A[i + 50]$ and $A[i + j]$. Warping in a loop node with loop iterator j, where loop iterator i corresponds to an enclosing loop, implies that the bijection π must map $A[i + 50]$ to $A[i + 50]$ for the first access node, and $A[i + j]$ to $A[i + j + 1]$ for the second access node (assuming the matching cache states differ by 1 in j). If j may obtain the value 50 this would yield conflicting requirements on the joint bijection π. Thus, for any two access

nodes with conflicting coefficients on the warped loop iterator, FurthestByOverlap determines the maximal loop iteration for which the ranges of the iterators do not overlap.

3. Finally, CacheAgrees checks whether the relation induced by the access sequences is compatible with the matching symbolic cache states. To this end, ConstructAccessMapping incrementally constructs the minimal required relation to satisfy (16) and CacheAgrees checks whether this conflicts with the induced relation between the concretizations of the matches, which corresponds to (17).

6 Experimental Evaluation

In our evaluation we aim to answer the following questions: 1. What are the benefits of warping cache simulation in terms of simulation performance? 2. How does warping cache simulation compare with analytical approaches such as HayStack and PolyCache? 3. How strong is the influence of different replacement policies on cache performance?

6.1 Experimental Setup

We implemented our approach as a cache simulation tool which takes as input the cache parameters and a C program, and outputs cache access and miss counts. We use *pet-0.11* (Polyhedral Extraction Tool) [51] to extract the polyhedral model from the C source and *isl-0.22* (Integer Set Library) [49] to perform operations on integer sets. We plan to release the source code of our tool as open source.

We evaluate our cache simulation tool on *PolyBench 4.2.1-beta* [41], a benchmark suite of numerical computations implemented as SCoPs. PolyBench benchmarks are configurable with different problem sizes. The experiments that we present here are for the large (L) and extra large (XL) problem sizes; the two largest ones.

We run our experiments single threaded using only one core on a test system with Intel Core i9-10980XE (Cascade Lake) processors. Unless stated otherwise, the cache simulation assumes the cache configuration found in the test system itself: Each core has an 8-way set-associative 32 KiB L1 cache with Pseudo-LRU replacement policy and a 16-way set-associative 1 MiB L2 cache with Quad-age LRU replacement policy both with a block size of 64 bytes. Both L1 and L2 caches are write-back write allocate and the inclusion policy between L1 and L2 is non-inclusive non-exclusive [42].

6.2 Warping vs Non-Warping Simulation

Warping vs non-warping simulation. We first simulate the L1 cache of the test system for problem size L. To investigate the effect of the replacement policy on the warping performance, in addition to the Pseudo-LRU policy of the test system, we also simulate LRU, FIFO, and Quad-age LRU.

Figure 6 shows for each benchmark the speedup of warping simulation compared to the non-warping simulation (bottom) and the share of non-warped accesses (top). The first observation is that the speedup is roughly inversely proportional to the share of non-warped accesses. For, e.g. *adi*, about 0.3% of all accesses cannot be warped and we observe a speedup of about 300*x*.

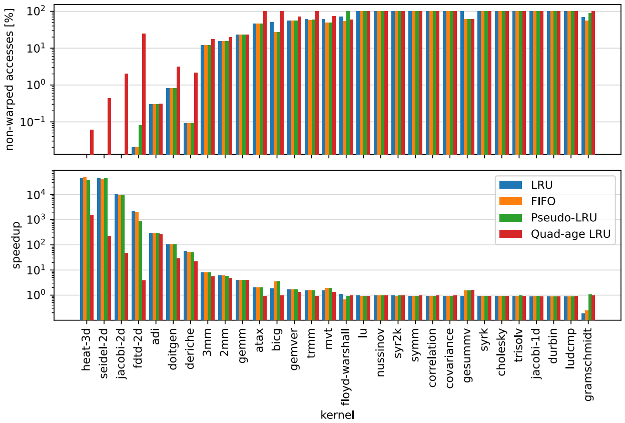

Figure 6. Speedup of L1 warping simulation compared to non-warping simulation (bottom) and share of non-warped accesses (top) for LRU, FIFO, Pseudo-LRU, and Quad-age LRU and problem size L.

The stencil kernels *adi*, *fdtd-2d*, *heat-3d*, *jacobi-2d*, and *seidel-2d* exhibit large speedups. Stencils have *uniformly generated references* [21, 54], and thus give rise to recurring patterns in the cache if there are enough accesses relative to the cache size. As we discussed earlier, warping aims to exploit these patterns to accelerate the simulation. The consistent speedups for the stencil kernels show that warping simulation is indeed able to achieve this. The *jacobi-1d* kernel does not benefit from warping since its working set is too small to fill the cache.

While there are many kernels that benefit from warping, there are others that do not. We observed that there were no (or very few) symbolically equivalent cache states during the simulation of these kernels, and thus, no (or very few) opportunities for warping. As we show later, some of these kernels benefit from warping when simulating a different cache. However, for the current cache configuration, we conclude that warping does not decrease the simulation times of these kernels.

Overall, the differences between the replacement policies are fairly small, with LRU, Pseudo-LRU, and FIFO often exhibiting similar speedups. Quad-age LRU is scan- and thrash-resistant [36], which may result in "old" memory blocks remaining in the cache, while scanning through new ones, which in some cases results in a greater number of classic simulation steps before detecting warping opportunities.

Impact of problem size. Figure 7 shows the change in warping and non-warping L1 simulation times between problem sizes L and XL for the configuration of the test system, i.e., with Pseudo-LRU replacement. We can see that for many

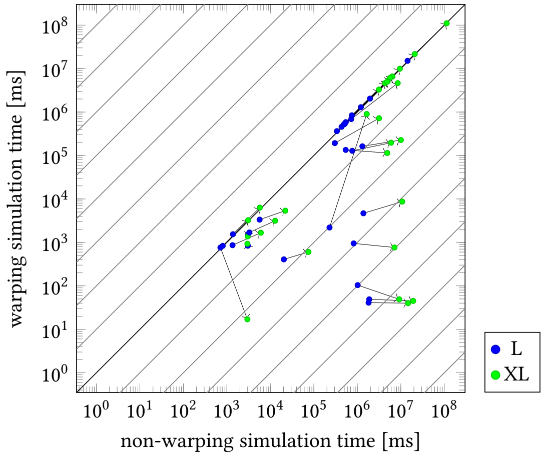

Figure 7. L1 warping and non-warping simulation times for problem sizes L and XL.

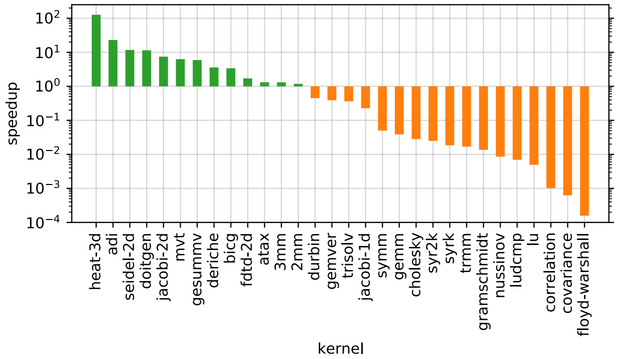

Figure 8. Speedup of L1 warping simulation compared to HayStack for problem size L.

benchmarks the warping simulation times are not proportional to the number of memory accesses while the non-warping simulation times are. On the other hand, there are also benchmarks whose warping simulation times change considerably between L and XL problem sizes. One interesting observation is that there are benchmarks whose simulation is faster with the XL problem size. This is unintuitive at first but it can happen when the simulator is able to warp across more accesses. Consider the simulation of a loop that has 10 iterations left. A matching cache state from 20 iterations ago cannot be used to warp across the last 10 iterations, as 10 is not a multiple of 20. However, for a larger problem size with e.g. 1000 iterations left, the analysis could warp to the end of the loop. This can have a considerable effect on the simulation time, especially when it applies at the outermost level of a deeply nested loop.

6.3 Warping Simulation vs Analytical Cache Modeling Approaches

We compare the performance of warping simulation to the analytical models PolyCache [7] and HayStack [32].

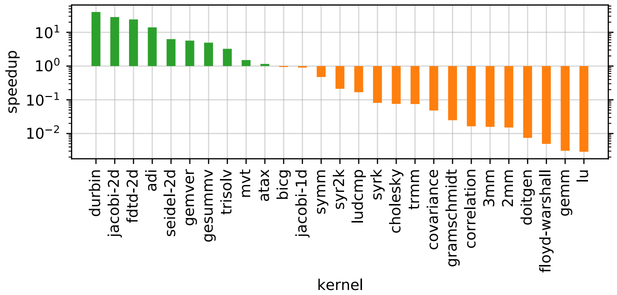

Figure 9. Speedup of L1-L2 warping simulation compared to PolyCache for problem size L.

Warping simulation vs HayStack. We compare warping simulation to the analytical cache model HayStack [32]. HayStack provides a replication package [31], which we use to replicate their experimental systems on our test system. We simulate the fully-associative LRU version of the L1 cache of the test system as HayStack can only model fully-associative caches with LRU replacement.

Figure 8 shows the speedup of warping simulation compared to HayStack for each kernel when only the L1 cache is simulated. We can see that HayStack is faster than warping cache simulation on most, but not all benchmarks. In particular, warping cache simulation outperforms HayStack on most stencil benchmarks.

Warping simulation vs PolyCache. We compare warping simulation to the analytical model PolyCache [7] using the published results as no replication package is available. For this purpose, we run warping simulation using the same PolyBench problem size (L) and cache configuration as PolyCache: a two-level cache with 32 KiB 4-way set-associative L1 and 256 KiB 4-way set-associative L2 caches. Both caches employ LRU replacement, write-allocate write-back write policy, and 64-byte cache blocks. Figure 9 shows the speedup of warping simulation compared to PolyCache for each benchmark. On the average, PolyCache outperforms warping cache simulation, but the relative performance varies greatly across the set of benchmarks. Note that this experiment is missing some of the PolyBench kernels as they are not included in the PolyCache results. We cannot correct for differences in the hardware on which the simulation is carried out. In contrast to our single-threaded implementation, PolyCache analyzes each of the 128 cache sets in a separate process. Thus, the single-thread performance of PolyBench would be expected to be about 128*x* slower.

6.4 Influence of Parameters on Cache Performance

Influence of the replacement policy. To determine the influence of the replacement policy, we simulate each benchmark under the four replacement policies LRU, FIFO, Pseudo-LRU, and Quad-age LRU on a 32 KiB 8-way set-associative L1 cache with 64-byte cache blocks. The results of this experiment are depicted in Figure 10. For most PolyBench benchmarks, cache performance does not vary dramatically depending on the replacement policy, but there are notable exceptions. In particular, on a number of benchmarks, e.g. *durbin* and *doitgen*, Quad-age LRU achieves significant improvements over LRU, while FIFO sometimes incurs significantly more misses. This demonstrates that accurately modeling the replacement policy can be important.

Comparison with measurements on actual hardware. We also evaluate the accuracy of cache simulation by comparing the number of cache misses predicted by the various simulators to PAPI-C [44] measurements on a real system. We compile the PolyBench [41] kernels with -O2 GCC optimization level and PAPI-C support to measure the cache misses on the test system. Note that PolyBench flushes the cache before executing each kernel. To minimize measurement errors, we repeat them 10 times and take the median. We also disable cache prefetching in our test system.

We compare the cache misses simulated by Dinero IV, warping simulation, and HayStack to the measured misses. Dinero IV simulates a set-associative LRU cache whereas HayStack models a same-size fully-associative LRU cache. Warping simulation simulates the system cache as it is, set-associative with Pseudo-LRU replacement policy. Note that as memory accesses, Dinero IV considers both array and scalar accesses while warping simulation and HayStack consider only array accesses.

When comparing the measured misses to the simulated or modeled cache misses, we consider two main metrics:

1. Absolute error: the absolute value of the difference between actual and predicted number of cache misses.
2. Relative error: absolute error divided by the actual number of misses.

The results of this experiment are depicted in Figure 11. For most benchmarks, all analytical approaches are similarly accurate, with some exceptions, e.g. on *atax* and *doitgen*, HayStack is significantly less accurate, due to its modeling of a fully-associative cache. The main takeaway though is that other aspects of modern microarchitectures, such as memory reordering and speculative execution, have a strong influence on cache performance that is not captured by any of the present approaches. Future work will have to further investigate this discrepancy.

7 Related Work

Traditional cache simulators. Cache simulators such as Dinero IV [18] and CASPER [34] simulate the cache behavior of a program by explicitly iterating over the memory access traces that are generated by the program. This approach applies to arbitrary programs and can model modern memory hierarchies precisely, including inclusive, non-inclusive non-exclusive, and exclusive cache hierarchies as well as sophisticated cache replacement policies such as Pseudo-LRU [3] and Quad-age LRU [35, 36], which are employed in recent real-world microarchitectures [2, 53]. The main drawback is that the simulation cost is proportional to the number of memory accesses that the simulated program performs.

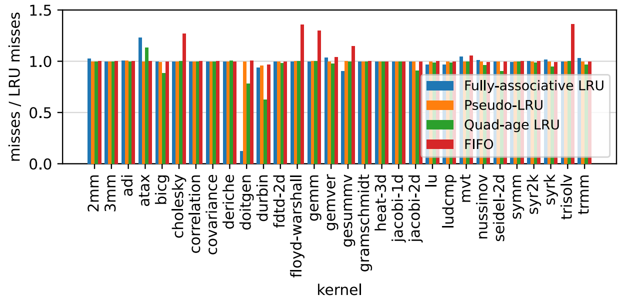

Figure 10. Number of misses relative to set-associative LRU.

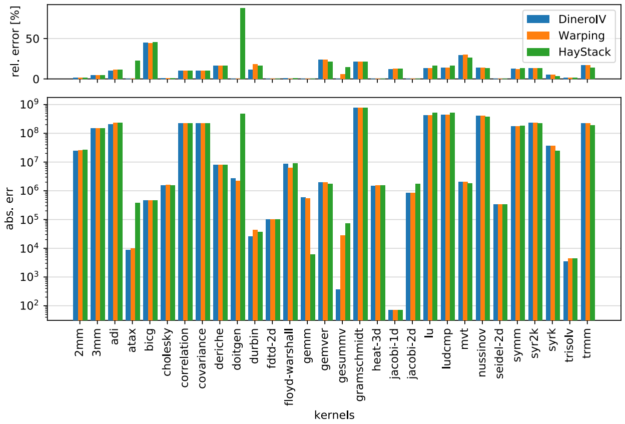

Figure 11. Accuracy relative to measurements on the actual hardware using PAPI.

Analytical cache models. There is a long history of analytical cache models [7, 11, 14, 15, 23, 24, 32, 47, 48]. Seminal work by Ghosh et al. [23, 24] introduces cache miss equations (CMEs), systems of linear Diophantine equations, that capture the set of cache misses of a loop nest. Their approach builds upon Wolf and Lam's [54] characterization of data reuse in loop nests via reuse vectors, and its classification into self-spatial, self-temporal, group-spatial, and group-temporal reuse. CMEs capture when these different types of reuse do not result in cache hits in single-level set-associative caches with LRU replacement. An inherent limitation of reuse vectors is their inability to accurately capture reuse in programs with conditional statements and between different references that are not uniformly generated.

Cascaval and Padua [14] present an exact approach to compute stack histograms [40] of programs at compile time. Stack histograms immediately reveal the number of cache misses under single-level fully-associative LRU caches for any given cache size, thus allowing to gauge the impact of different cache sizes on a program's cache performance. Their work is limited to the same class of programs as Ghosh et al.'s

CMEs [23, 24], but a larger class of programs can be modeled approximately in this framework, sacrificing accuracy.

Vera and Xue [48] extend the applicability of CMEs to a larger class of programs involving conditional statements, multiple loop nests, and subroutines by transforming such projects into a more restricted normal form. Some of these transformations, however, approximate the original program's behavior, rendering the analysis inexact.

Chatterjee et al. [15] introduce a compositional characterization of the cache behavior of polyhedral programs [19] via Presburger formulas [33] for single-level set-associative caches with LRU replacement. Their approach takes into account the initial cache state and distinguishes interior misses and boundary misses, which allows to analyze sequential programs in a compositional manner. At the time of publication, the approach did not scale to realistic levels of associativity.

More recent work by Bao et al. [7] introduces PolyCache, a tool applicable to polyhedral programs, just like Chatterjee et al. [15] and our work. By analytically characterizing the sequence of cache misses at a given cache level, it can incrementally handle write-allocate non-inclusive non-exclusive [42] *multi-level* set-associative caches with LRU replacement. Non-write allocate caches are handled approximately. Similarly to [15] their method constructs an integer set consisting of a program's cache misses. Then, *isl*'s [49] implementation of Barvinok's algorithm is used to compute the integer set's size, and thus the number of cache misses. As we have shown in the experimental evaluation, PolyCache outperforms warping cache simulation on the average, but the relative performance varies greatly across the set of benchmarks, and unlike our work the approach is not applicable to replacement policies other than LRU and it handles non-write allocate caches approximately.

Gysi et al. [32] present HayStack, the most scalable analytical cache analysis approach to date. Their approach applies to polyhedral programs, but it is limited to fully-associative LRU caches and inclusive hierarchies of such caches, which do not require to model the interaction between different cache levels. HayStack performs symbolic counting twice: first, a Presburger relation is constructed that relates each access A to its "conflict set", i.e., the set of distinct memory blocks accessed between the most-recent access to the memory block accessed by A. The size of this conflict set, is the access's stack distance. In a fully-associative LRU cache, an access results in a cache miss if and only if its stack distance is greater than the cache's associativity. In the first step, the stack distances of all accesses are determined by symbolic counting of the conflict sets. This step is inspired by prior work of Beyls and D'Hollander [11]. In the second step, the number of accesses with a stack distance greater than the cache's associativity is determined, again by symbolic counting. This second step is challenging, as the stack distances are generally non-affine. Non-affine terms are eliminated by a partially explicit enumeration before applying symbolic counting. The experimental evaluation demonstrates that HayStack is more scalable than our approach on the average, but it is limited to fully-associative LRU caches.

Analytical program representations. The polyhedral model [10, 19, 20] is the basis of our work and many of the recent analytical cache models [7, 15, 32]. One of its original applications and that of related techniques [37, 38] was to capture data dependencies to facilitate the automatic generation of parallel schedules. More recently [6, 13, 52, 54] it has also been applied to generate schedules that exhibit more locality.

Static cache analysis. Static cache analyses [4, 16, 25–30, 45, 46] bound a program's cache behavior for all possible program executions. This is different from cache simulation, which applies to a particular program execution. For polyhedral programs, whose data-access behavior is input independent these goals align. However, existing static cache analyses do not classify each dynamic memory access separately, but rather collectively classify sets of accesses, e.g. all accesses corresponding to a memory reference. Due to their coarse classification granularity even exact static analyses [16, 45, 46] overapproximate a polyhedral program's cache misses. Existing cache analyses are generally also handcrafted to particular replacement policies in contrast to this work, which applies to arbitrary policies that satisfy the data-independence property. Most cache analyses are tailored to LRU [4, 16, 45, 46], while some work is dedicated to FIFO [26, 27, 30], NMRU [29], and Pseudo-LRU [25, 28]. To our knowledge, there is no static cache analysis for Quad-age LRU.

Acceleration techniques. Warping cache simulation bears some resemblance of *acceleration* techniques [5, 8, 12, 17, 22, 39, 43]. Such techniques accelerate the computation of the reachable set of states of a given model by computing the exact effect of iterating through a control cycle in the model, where the control cycle to iterate is determined dynamically during the analysis. In contrast, the cycles that warping cache simulation iterates across are generated by the composition of a polyhedral program and a cache model, and thus the cycle is not present explicitly in the input to the analyzer. Nevertheless, further exploring the connection to acceleration techniques might be fruitful.

8 Conclusions and Future Work

We have introduced warping cache simulation and demonstrated its benefits experimentally: Warping may speed up simulation by several orders of magnitude. In contrast to existing analytical approaches, warping cache simulation may accurately model replacement policies of real-world cache architectures. A natural target for future work is to apply warping to efficiently simulate modern speculative out-of-order processors core and branch prediction mechanisms and their interaction with the cache, which promises to increase the accuracy of the predictions w.r.t. real hardware.

References

[1] Andreas Abel and Jan Reineke. 2013. Measurement-based modeling of the cache replacement policy. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 65–74. https://doi.org/10.1109/RTAS.2013.6531080

[2] Andreas Abel and Jan Reineke. 2020. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 34–46. https://doi.org/10.1109/ISPASS48437.2020.00014

[3] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. 2004. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*. ACM, 267–272.

[4] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache Behavior Prediction by Abstract Interpretation. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*. 52–66. https://doi.org/10.1007/3-540-61739-6_33

[5] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. 2001. TReX: A Tool for Reachability Analysis of Complex Systems. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 368–372. https://doi.org/10.1007/3-540-44585-4_34

[6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (1994), 345–420. https://doi.org/10.1145/197405.197406

[7] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 32:1–32:26. https://doi.org/10.1145/3158120

[8] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. 2003. FAST: Fast Acceleration of Symbolic Transition Systems. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 118–121. https://doi.org/10.1007/978-3-540-45069-6_12

[9] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–303.

[11] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *J. Syst. Archit.* 51, 4 (2005), 223–250. https://doi.org/10.1016/j.sysarc.2004.09.004

[12] Bernard Boigelot and Pierre Wolper. 1994. Symbolic Verification with Periodic Sets. In *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 818)*, David L. Dill (Ed.). Springer, 55–67. https://doi.org/10.1007/3-540-58179-0_43

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 101–113. https://doi.org/10.1145/1375581.1375595

[14] Calin Cascaval and David A. Padua. 2003. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 150–159. https://doi.org/10.1145/782814.782836

[15] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. 2001. Exact Analysis of the Cache Behavior of Nested Loops. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 286–297. https://doi.org/10.1145/378795.378859

[16] Sudipta Chattopadhyay and Abhik Roychoudhury. 2013. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real Time Syst.* 49, 4 (2013), 517–562. https://doi.org/10.1007/s11241-013-9178-0

[17] Hubert Comon and Yan Jurski. 1998. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1427)*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer, 268–279. https://doi.org/10.1007/BFb0028751

[18] Jan Edler and Mark D. Hill. 1999. Dinero IV Trace-Driven Uniprocessor Cache Simulator.

[19] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[20] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Program.* 21, 5 (1992), 313–347.

[21] Dennis Gannon, William Jalby, and Kyle A. Gallivan. 1988. Strategies for Cache and Local Memory Management by Global Program Transformation. *J. Parallel Distributed Comput.* 5, 5 (1988), 587–616. https://doi.org/10.1016/0743-7315(88)90014-7

[22] Thomas Gawlitza, Jérôme Leroux, Jan Reineke, Helmut Seidl, Grégoire Sutre, and Reinhard Wilhelm. 2009. Polynomial Precise Interval Analysis Revisited. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 5760)*, Susanne Albers, Helmut Alt, and Stefan Näher (Eds.). Springer, 422–437. https://doi.org/10.1007/978-3-642-03456-5_28

[23] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1997. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the 11th international conference on Supercomputing, ICS 1997, Vienna, Austria, July 7-11, 1997*, Steven J. Wallach and Hans P. Zima (Eds.). ACM, 317–324. https://doi.org/10.1145/263580.263657

[24] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. Program. Lang. Syst.* 21, 4 (July 1999), 703–746. https://doi.org/10.1145/325478.325479

[25] David Griffin, Benjamin Lesage, Alan Burns, and Robert I. Davis. 2014. Lossy Compression for Worst-Case Execution Time Analysis of PLRU Caches. In *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza (Eds.). ACM, 203. https://doi.org/10.1145/2659787.2659807

[26] Daniel Grund and Jan Reineke. 2009. Abstract Interpretation of FIFO Replacement. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 120–136. https://doi.org/10.1007/978-3-642-03237-0_10

[27] Daniel Grund and Jan Reineke. 2010. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*. IEEE Computer Society, 155–164. https://doi.org/10.1109/ECRTS.2010.8

[28] Daniel Grund and Jan Reineke. 2010. Toward Precise PLRU Cache Analysis. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium (OASICS, Vol. 15)*, Björn Lisper (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik,

Germany, 23–35. https://doi.org/10.4230/OASIcs.WCET.2010.23

[29] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. 2014. WCET Analysis with MRU Cache: Challenging LRU for Predictability. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 123 (April 2014), 26 pages. https://doi.org/10.1145/2584655

[30] Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. 2013. FIFO Cache Analysis for WCET Estimation: A Quantitative Approach. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Grenoble, France) *(DATE 2013)*. EDA Consortium, San Jose, CA, USA, 296–301. http://dl.acm.org/citation.cfm?id=2485288.2485362

[31] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. [n.d.]. *Replication Package for Article: A Fast Analytical Model of Fully Associative Caches.* https://doi.org/10.1145/3325990

[32] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 816–829. https://doi.org/10.1145/3314221.3314606

[33] Christoph Haase. 2018. A Survival Guide to Presburger Arithmetic. *ACM SIGLOG News* 5, 3 (July 2018), 67–82. https://doi.org/10.1145/3242953.3242964

[34] Ravi Iyer. 2003. On modeling and analyzing cache hierarchies using CASPER. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.* 182–187. https://doi.org/10.1109/MASCOT.2003.1240655

[35] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. 2012. Power management of the third generation Intel Core Micro Architecture formerly Codenamed Ivy Bridge. *2012 IEEE Hot Chips 24 Symposium (HCS)* (Aug 2012). https://doi.org/10.1109/hotchips.2012.7476478

[36] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[37] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. 1967. The Organization of Computations for Uniform Recurrence Equations. *J. ACM* 14, 3 (July 1967), 563–590. https://doi.org/10.1145/321406.321418

[38] Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (1974), 83–93. https://doi.org/10.1145/360827.360844

[39] Jérôme Leroux and Grégoire Sutre. 2007. Accelerated Data-Flow Analysis. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4634)*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 184–199. https://doi.org/10.1007/978-3-540-74061-2_12

[40] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117. https://doi.org/10.1147/sj.92.0078

[41] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite.

[42] Yan Solihin. 2015. *Fundamentals of Parallel Multicore Architecture* (1st ed.). Chapman & Hall/CRC.

[43] Zhendong Su and David A. Wagner. 2004. A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings and Narrowings. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 280–295. https://doi.org/10.1007/978-3-540-24730-2_23

[44] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[45] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2017. Ascertaining Uncertainty for Efficient Exact Cache Analysis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 22–40. https://doi.org/10.1007/978-3-319-63390-9_2

[46] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and exact analysis for LRU caches. *Proc. ACM Program. Lang.* 3, POPL (2019), 54:1–54:29. https://doi.org/10.1145/3290367

[47] Xavier Vera, Nerina Bermudo, Josep Llosa, and Antonio González. 2004. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Trans. Program. Lang. Syst.* 26, 2 (2004), 263–300. https://doi.org/10.1145/973097.973099

[48] Xavier Vera and Jingling Xue. 2002. Let's Study Whole-Program Cache Behaviour Analytically. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), Boston, Massachusettes, USA, February 2-6, 2002.* IEEE Computer Society, 175–186. https://doi.org/10.1109/HPCA.2002.995708

[49] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302.

[50] Sven Verdoolaege. 2016. Presburger formulas and polyhedral compilation. https://lirias.kuleuven.be/retrieve/361209

[51] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France.

[52] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[53] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: learning replacement policies from hardware caches. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 519–532. https://doi.org/10.1145/3385412.3386008

[54] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 30–44. https://doi.org/10.1145/113445.113449