Analytical Modeling of Cache Behavior for Affine Programs

WENLEI BAO, The Ohio State University

SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory

LOUIS-NOEL POUCHET, Colorado State University

P. SADAYAPPAN, The Ohio State University

Optimizing compilers implement program transformation strategies aimed at reducing data movement to or from main memory by exploiting the data-cache hierarchy. However, instead of attempting to minimize the number of cache misses, very approximate cost models are used, due to the lack of precise compile-time models for misses for hierarchical caches. The current state of practice for cache miss analysis is based on accurate simulation. However, simulation requires time proportional to the dataset/problem size, as well as the number of distinct cache configurations of interest to be evaluated.

This paper takes a fundamentally different approach, by focusing on polyhedral programs with static control flow. Instead of relying on costly simulation, a closed-form solution for modeling of misses in a set-associative cache hierarchy is developed. This solution can enable program transformation choice at compile time to optimize cache misses. A tool implementing the approach has been developed and used for validation of the framework.

CCS Concepts: • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Compiler optimization, performance analysis, static analysis

ACM Reference Format:

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2018. Analytical Modeling of Cache Behavior for Affine Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 32 (January 2018), 26 pages. <https://doi.org/10.1145/3158120>

1 INTRODUCTION

Optimizing compilers currently use simplified cost models to select the composition of loop transformations applied to programs. For example, Bondhugula et al. maximized idealized measures of data locality [Bondhugula et al. 2008], Kong et al. maximized data locality under single instruction/multiple data (SIMD) constraints [Kong et al. 2013], and other works have examined combined objectives for parallelization and data locality (e.g., [Feautrier 1992; Lim and Lam 1997]). While useful in practice, the major drawback of these approaches is that they are limited to abstract performance models, such as “maximizing parallelism” or “minimizing reuse distance,” which are only indirectly related to concrete performance metrics, such as data cache misses.

Addressing these limitations requires several challenges to be addressed. First, approaches to accurately model cache behavior typically employ actual execution or simulation to measure or analyze the program’s execution on the target device. These approaches can be expensive, with cost proportional to the number of instructions the program executes. For example, running the DineroIV [Edler and Hill 1999] cache simulator on a simple matrix-multiply code for 1024×1024

Authors’ addresses: Wenlei Bao, The Ohio State University, bao.79@osu.edu; Sriram Krishnamoorthy, Pacific Northwest National Laboratory, sriram@pnl.gov; Louis-Noel Pouchet, Colorado State University, pouchet@colostate.edu; P. Sadayappan, The Ohio State University, sadayappan.1@osu.edu.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

© 2018 Association for Computing Machinery.

2475-1421/2018/1-ART32

<https://doi.org/10.1145/3158120>

matrices can take hours. Most importantly, these approaches do not provide an exact model of cache behavior for use at compile time. Second, nearly all practical modern systems use cache hierarchies. A small, first-level (L1) cache is complemented with a larger second-level (L2) cache, and so on – the Intel Haswell CPU may contain up to four levels of caches. To accurately model the accesses to main memory, which are much slower than any cache accesses and can significantly influence program performance, the full hierarchy of caches must be modeled, where accesses to level $L+1$ are derived from the misses at level L . Third, most practical caches are set-associative rather than fully associative, where multiple virtual memory addresses may map to a specific set of physical cache lines, requiring accurate models of conflict misses.

In the quest for an approach that would enable compile-time analysis of runtime cache behavior, we need to: (1) model the cache hierarchy; (2) accurately model cold, capacity, and conflict misses of set-associative caches at each level; and (3) achieve an analysis time that is essentially independent of problem size and cache size. In this work, we achieve all of these goals by focusing specifically on the class of affine programs, where key properties of the control and data flows afford accurate capture of runtime cache behavior. We also propose a novel and purely static analysis to accurately compute cache behavior at compile time, without the need for execution or simulation.

Compile-time cache modeling (e.g., avoiding simulation) for affine programs has been studied in the past. For example, tile size selection has been investigated by approximating capacity misses for regular/affine data tiles, such as the distinct line model [Sarkar and Megiddo 2000] and its generalization with the minimum line model [Shirako et al. 2012]. Ghosh et al. [1998, 1999] proposed Cache Miss Equations (CME) to capture situations when a cache miss occurs in perfectly nested loop programs with uniform references. It handles set-associative caches, but only for single-level caches. However, a major limitation of the approach is the difficulty to translate these equations into a cache miss count, which requires actually solving the CMEs (i.e., determining whether or not they have valid solutions) for a multitude of cases. Furthermore, translating a miss *count* at the first-level cache into a set of cache access for the next level is impractical. Thus, this approach is not suitable for effectively modeling hierarchical caches that are ubiquitous in practical systems.

Chatterjee et al. [2001] made a significant advance by providing an exact closed-form solution for polyhedral programs for direct-mapped caches using Presburger formulae, which is partially generalized to model a set-associative single-level cache. However, none of these prior formulations of the cache miss problem are suited for modeling hierarchical caches. There is a fundamental difference between simply counting cache misses, as in prior work, and modeling the set of accesses that lead to a cache miss, as developed in this work. Such a set of events should be exactly modeled to compute misses in cache hierarchies because only the subset of accesses to L1 that misses in L1 form the set of accesses to L2.

We make the following contributions:

- We present a closed-form solution to the problem of modeling cache behavior for arbitrary polyhedral programs executing on processors with set-associative virtually-indexed hierarchical caches under a variety of write policies.
- We develop a tool that computes the number of misses at each cache level for set-associative caches, from an input C source file containing affine program(s).
- We perform extensive evaluation of our cache miss analysis and validate it against a simulator with identical hardware assumptions, and present several approximation heuristics.

The rest of the paper is organized as follows: Sec. 2 elaborates on the problem. Sec. 3 presents the notation and background on mathematical concepts used in the work. Sec. 4 presents a static analysis for a single-level set-associative cache, while Sec. 5 generalizes to hierarchical caches with various write policies. Extensive evaluation is reported in Sec. 7. Additional optimizations and use cases are discussed in Sec. 8. Related work is presented in Sec. 9 before concluding the paper.

2 OVERVIEW OF MODELING APPROACH

Fig. 1. Illustrative example of a 2-way set-associative cache

Fig. 1 outlines the key steps in computing cache misses for a sample program (shown in Fig. 1(a)). Fig. 1(d) shows the execution trace of the various read/writes for execution of the program. The following includes a walk-through for computing the set of memory accesses that lead to cache misses in a simple 2-way set-associative cache.

A K -way set-associative cache is partitioned into cache sets, each of which can store K cache lines. A cache line stores B bytes of data. For a cache of size n bytes, the total number of cache lines, n_{blines} , is given by n/B , and the total number S of cache sets is $(n/B)/K$. To determine the specific cache set that a memory address maps to, a classical and typical approach is to use the least significant bits of the memory address: a virtual memory address $addr$ maps to cache set $(addr/B)\%S$. Since the cache is K -way associative, up to K lines mapping to the same cache set can co-exist simultaneously in the cache. We illustrate this in Fig. 1(d) for a simple 2-way associative cache of size 32 bytes with 4 cache lines: $B = 8$, $K = 2$, and $S = 2$, that is each cache line stores a single element of A . The trace of accesses in Fig. 1(d) is typical of input traces provided to cache simulators, such as DineroIV [Edler and Hill 1999]. First, the (annotated) program is executed to

build the trace of accesses. Then, the simulator reads this trace in sequential order, one reference at a time. For each address accessed, it computes the mapped cache set and maintains an internal data structure representing the cache state (i.e., what data is currently in cache). When the accessed data item is not currently in cache, a cache miss occurs, possibly leading to the eviction of a cache line previously stored in this cache set, and the loading of the data into a line in this set from a copy in a later level cache (or main memory). Conversely, when the accessed data is already in cache, a cache hit occurs. Fig. 1(d) shows the cache line and cache set to which each address maps. The “virtual cache line” is computed by $L_{id} = addr/B$ and the cache set by $S_{L_{id}} = (addr/B)\%S$. We remark that $S_{L_{id}} = (addr/B)\%S = ((addr/B)\%nblines)\%S$ because we have $nblines = K * S$, $K \geq 1$. That is, L1,S1 is read as $L_{id} = 1, S_{L_{id}} = 1$ for this reference. Fig. 1(e) shows whether each access is a hit or miss for this cache configuration.

Consider a cache line L_{id} to be accessed. This access is a miss if it is the first time L_{id} is accessed or K distinct cache lines that map to the same set have been accessed more recently than L_{id} . Otherwise, it is a cache hit. Indeed, in a set-associative cache, each line maps to a specific set with K slots. A cold miss (an unavoidable miss even in an infinite cache) occurs the first time L_{id} is accessed. A capacity/conflict miss (one that can be avoided in an infinite cache) occurs if K different lines mapping to the same set have been accessed prior to (re)accessing L_{id} . In other words, for a particular cache set, all $K + 1$ distinct accessed lines mapping to this set will lead to a cache miss.

Focusing on Set 1 in the 2-way set-associative cache, Fig. 1(d) shows that first $A[0, 1]$ and then $A[1, 1]$ are loaded in Set 1 (they are both cold misses), and $A[0, 1]$ then is accessed again (it is a hit: no line has been evicted from Set 1 yet). When $A[0, 3]$ is loaded, Set 1 is already full, and the least recently used (LRU) line is evicted. $A[1, 1]$ is deleted, and $A[0, 3]$ is stored in the cache. A cache simulator such as DineroIV operates exactly the same way. Following the sequence order in the program trace, it determines where a reference maps, using L_{id} and $S_{L_{id}}$, and keeps track of the current cache state to determine if an access is a hit or a miss, performing evictions as needed, e.g., based on the LRU policy. The total number of misses is reported, and, for hierarchical caches, accesses that lead to a miss in L1 cache are used as accesses to L2 cache, etc.

In this paper, we describe a compile-time analysis intended to produce the same output as a cache simulation, i.e., the set of accesses that lead to cache misses, *without actually generating the program trace or simulating cache state*. We achieve this by restricting the class of modeled programs to *polyhedral programs* (defined in Sec. 3). In a nutshell, for polyhedral programs, the exact ordered set of all accessed memory locations can be described in a compact form using integer sets and relations, as presented in Sec. 3. From this example, we observe that the following are needed for such an analysis: (1) the ordered set of memory accesses (capturing the same information as the trace in Fig. 1(d)), (2) a mechanism to determine which line and set an address is mapped to (modeling exactly Fig. 1(d)’s mapping), and (3) the ability to determine sequences of accesses to $K + 1$ distinct lines mapping to the same set. Item (1) above is a direct consequence of polyhedral programs. The exact ordered set of accesses can always be built for these programs. For (2), the L_{id} and $S_{L_{id}}$ expressions can be modeled as Presburger formulae if n , B and S are known at compile time, as Chatterjee et al. [2001] showed. Achieving (3) is a core contribution of this paper (Sec. 4).

We illustrate the approach with Fig. 1(f)-(h), using the previously described code and cache configuration. First, a polyhedral representation of the program is generated. The set of executions of the statement is captured by a polyhedron, Fig. 1(b) plots this iteration domain of the program. Each dot corresponds to an execution of the associated color-coded reference, and the coordinates of the dot capture the value of the corresponding loop iterators. The program execution order (the schedule) can always be determined. In Fig. 1(b), it corresponds to the lexicographic ordering.

For each array reference (e.g., $A[i][j+1]$), an access function from each iteration point to the accessed memory cell is built (e.g., $F^A : \{[i, j] \rightarrow [i, j + 1]\}$). Fig. 1(c) displays the set of memory

cells from the array `double A[3][4]` that are accessed by this program. Fig. 1(c) is obtained from Fig. 1(b) by applying the access functions to the iteration domain.

Fig. 1(g) corresponds to the *subset* of cells from A that map to Set 1. This can be obtained by adding constraints so that only accesses with $S_{L_{id}} = 1$ are kept, i.e. $(addr/4)\%2 == 1$, where $addr$ is the accessed memory address. $addr$ is obtained from F^A by linearizing the access function. We do require the array sizes to be known and constant, to obtain $\{[i, j] \rightarrow [A + i * 4 + j + 1]\}$. We remark that because the hit/miss pattern of a cache set is not impacted by the accesses to other cache sets, we can model accesses to each cache set independently and aggregate the results for all cache sets at the end. Fig. 1(f) displays the same subset of references that map only to Set 1, by showing the subset of points in the iteration domain that correspond to these references. This pruned iteration domain forms the (ordered) set of references to Set 1 only, obtained in a manner similar to how Fig. 1(g) is built, but reasoning on iteration points instead.

The last and most critical step is to obtain Fig. 1(h), the subset of only the accesses that lead to a miss (cold and capacity/conflict). This is built in two stages. First, we determine the subset corresponding to cold misses, which is computed as the first access to a unique cache line (unique L_{id}). Given a function **IterToLine** that maps iteration points to the virtual cache line L_{id} they access (as defined in Sec. 4), the first access to each L_{id} is computed by taking the lexicographically first (with respect to the program schedule) iteration accessing L_{id} , i.e., the preimage of $\text{lexmin}(\text{IterToLine})$. This can be computed analytically for polyhedral programs. In Fig. 1(h), these accesses correspond to the dots *not* inside a square.

Second, we must determine the ordered sequence of virtual lines L_{id} accessed by the program, where for a particular set $S_{L_{id}}$, every $K + 1$ -th accessed line mapping to that set will incur a cache miss, and thus must be captured. In Fig. 1(h), it corresponds to the dot inside the square. We solve this problem by building several functions, typically from an iteration point to another (set of) iteration point(s) with some specific properties. For example, the composition $\text{SameSet} = \text{IterToSet} \circ \text{IterToSet}^{-1}$ maps an iteration point $\vec{i}$ to the other iteration points $\{\vec{j}\}$ accessing the same cache set, some of which will be miss events. We successively refine **SameSet** to end up with only the iterations $\{\vec{j}\}$ which do incur a cache miss. Iterations mapping to the same virtual line are computed as $\text{SameLine} = \text{IterToLine} \circ \text{IterToLine}^{-1}$. We can combine **SameSet** and **SameLine** to keep only the iterations that correspond to accesses to distinct cache lines within a set. For a K -way associative cache, we then successively remove the second, third, ..., K -th unique line accessed mapping to the same cache set and retain only the $K + 1$ -th ones, i.e., the collection of miss events. This is achieved by building a function from a point in an iteration domain to the immediately next executed point in this domain. This function can be composed with others to get the immediately next *relevant* iteration for our analysis, e.g., the next iteration accessing the same cache set and a different cache line. We proceed by successively removing such next iterations, K times, to end up with only the iterations incurring a miss, as detailed in Sec. 4.

The resulting set (e.g., Fig. 1(h)) has two fundamental properties: it can be counted to obtain the number of cache misses, and be used to model accesses to the next level cache, since misses at L1 form the set of accesses at L2. By composing this process, we can seamlessly reason on hierarchical set-associative caches and produce per-level cache miss count, paving the way for simulation-free compile-time analysis and optimization of cache behavior for polyhedral programs (discussed in Sec. 8).

3 PROGRAM REPRESENTATION

In this work, we target the compile-time analysis of polyhedral program regions [Feautrier 1992; Girbal et al. 2006], also called *affine* programs. A program region is polyhedral if the only control-flow structures are for loops and if conditionals; the iteration set of each syntactic statement can be described by static analysis using affine inequalities of the surrounding loop iterators and program region parameters (unknown but constant values for one execution of the region); the access functions of arrays are multidimensional affine functions of the surrounding loop iterators and parameters; each for loop has loop-invariant loop bounds, a constant scalar stride, and the exit value of the loop iterator is not read outside the loop; and syntactic statements may contain a data-dependent conditional (e.g., using a ternary operator in C), but both the true and false branch will be assumed to be always taken in the analysis.

Polyhedral programs have essential properties that we rely on to build our static analysis (Sec. 3.2). This program class covers a wide spectrum of compute- and data-intensive processing kernels typically found in linear algebra methods, image processing, or physics simulation [Bao et al. 2016a,b; Girbal et al. 2006] where high performance is a must and, consequently, exploiting cache hierarchies effectively is highly desired. We first describe the key mathematical objects manipulated in this work and their operations. We follow with presenting the representation of affine programs using these structures.

3.1 Modeling Integer Tuples

All mathematical structures and operations used in this work involve modeling integer tuples using Presburger formulae to model sets and relations among them. Operations on these structures are readily available in the Integer Set Library (ISL) [Verdoolaege 2010a], including more advanced operations, such as counting the number of points in a set or relation [Verdoolaege et al. 2007]. All structures and operations are briefly surveyed in this work. As integer sets are manipulated, the worst-case complexity of most operations is NP-hard. However, in practice, quasi-polynomial time is often achieved. Sec. 7.6 details the measured execution time of these operations.

Integer tuple. A point in a set is a multidimensional integer vector, or integer tuple. Its n components take value in $\mathbb{Z}$: $\mathbf{i} \in \mathbb{Z}^n$.

Integer sets. A set S of integer tuples is a subset of $\mathbb{Z}^n$ defined as:

$$S = [p_1, \dots, p_p] \rightarrow \{[i_1, \dots, i_n] : c_1 \wedge \dots \wedge c_m\}$$

where $i_1, \dots, i_n$ index the n dimensions of the set (noted $\vec{i}$); $p_1, \dots, p_p$ are invariant parameters (noted $\vec{p}$); and $c_1, \dots, c_m$ are m Presburger formulae typically in the form of affine inequalities defining constraints on the values of $\vec{i}$. Presburger formulae in this work involve existential quantifiers, modulo operations, integer division, and ceil/floor.

Integer sets are not polyhedra. They can contain “holes,” i.e., they can model the intersection of a polyhedron with an integer lattice, for example, to model a subset containing only some even integers.

Relation. A relation R maps points $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$ and is defined as:

$$R = [p_1, \dots, p_p] \rightarrow \{[i_1, \dots, i_n] \rightarrow [j_1, \dots, j_m] : c_1 \wedge \dots \wedge c_o\}$$

where $\vec{i}$ are the input dimensions and $\vec{j}$ the output dimensions. Similarly to sets, Presburger formulae are used to express a relation between points in two sets of possibly different dimensions but also to express constraints on the input and output sets.

Domain and range of a relation. To retrieve the set of points where a relation can be applied (its domain) and the set of points that may be produced (its range), use:

$$\text{domain}(\mathbf{R}) = \{[i_1, \dots, i_m] : i \text{ s.t. } \exists j, j = \mathbf{R}(i)\}$$

and:

$$\text{range}(\mathbf{R}) = \{[j_1, \dots, j_n] : j \text{ s.t. } \exists i, j = \mathbf{R}(i)\}$$

Composition. Two relations $\mathbf{R}_1 : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$ and $\mathbf{R}_2 : \mathbb{Z}^m \rightarrow \mathbb{Z}^o$ can be composed to form a new relation:

$$\begin{aligned} \mathbf{R}_1 \circ \mathbf{R}_2 &= \{[i_1, \dots, i_n] \rightarrow [k_1, \dots, k_o] : \\ &\exists i \in \text{domain}(\mathbf{R}_1), \exists k \in \text{range}(\mathbf{R}_2) : k = \mathbf{R}_2(\mathbf{R}_1(i))\} \end{aligned}$$

Inverse. A relation can be inverted or reversed, i.e., the input and output dimensions are reversed. For a relation $\mathbf{R}$ as defined previously, its inverse is:

$$\mathbf{R}^{-1} = [p_1, \dots, p_p] \rightarrow [j_1, \dots, j_n] \rightarrow [i_1, \dots, i_m] : c_1 \wedge \dots \wedge c_o\}$$

Application. A relation can be applied to a set, which results in a set of points that are the image of the input set S by the relation $\mathbf{R}$:

$$S_{\mathbf{R}} = \mathbf{R}(S) = \{x \in S : (\exists y \in S \wedge (y \rightarrow x) \in \mathbf{R})\}$$

Union and intersection. We manipulate union, intersection, and difference of sets and relations. These binary operations are written $\cup$ for union, $\cap$ for the intersection, and $-$ for the difference.

Counting. Finally, an essential operation we extensively rely on is the ability to build a counting formula for an integer set (or relation). This formula takes the form of an Ehrhart quasi-polynomial and can be computed using the Barvinok algorithm implementation in ISL [Verdoolaege 2007; Verdoolaege et al. 2007]. We note this operation $\#S$ for the cardinality of a set S .

3.2 Representing Programs

Programs with affine data flow and static control flow are called *static control parts* (SCoP) [Feautrier 1992; Girbal et al. 2006]. Polyhedral programs are represented in this work using (a union of) integer sets and integer relations. Three key structures are needed in this work to define a program. For all statements in the program, we capture its iteration domain, data access relations, and schedule of iterations. We use the illustrative example in Fig. 2.

```

for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {
    for (k = i+1; k < M; k++)
R:      B[i][j] += A[k][i] * B[k][j];
S:      B[i][j] = alpha * B[i][j];
  }

```

Fig. 2. Triangular Matrix-Multiply

Iteration domains. Iteration domains capture the set of runtime executions of a statement. Because programs are polyhedral, this set can be exactly captured using integer sets where the loop bounds are used to constrain the number of points in the set. Each statement R is associated with an iteration vector $\vec{i}_R$ with one component per surrounding loop, and the values $\vec{i}_R$ can take are captured by defining its iteration space $\mathcal{D}_R$. For example, the iteration domain of R in Fig. 2 is:

$$\mathcal{D}_R = [M, N] \rightarrow \{R[i, j, k] : 0 \leq i < M \wedge 0 \leq j < N \wedge i + 1 \leq k < M\}$$

It is possible to count the number of points in this set with:

$$\#\mathcal{D}_R = (-1/2 * M + 1/2 * M^2) * N, \text{ with } M \geq 2 \wedge N > 0.$$

We note *ProgDomain* the union of all per-statement iteration domains, i.e., the set of all iteration domains for the entire program.

Data access functions. An essential part of our analysis is based on representing the data accessed by each program iteration. For polyhedral programs, the function that maps a statement instance to the array cell being accessed is, by definition, an affine relation, involving surrounding loop iterators and parameters. We will distinguish the read and write references, and the access relation maps an iteration domain to the multidimensional array index being accessed. For example, the function that relates the iterations of R with the location read in array A for the reference $A[k][i]$ in Fig. 2 is:

$$\text{Read}_R^A = \{R[i, j, k] \mapsto A[i_1, j_1] : (i_1 = k) \wedge (j_1 = i)\}$$

We note **Write** for the write references, and the set of all read and/or write access functions for the program is obtained by building the union of the per-statement data access relations. Furthermore, it also is possible to build the relation restricted to the set of iterations of R by computing $\mathbf{R} = \text{Read}_R^A \cap \mathcal{D}_R$, i.e., embed the constraints on the possible values for $R[i, j, k]$ directly in the relation. The set of R iterations then can be recovered by computing $\text{domain}(\mathbf{R})$ or the set of array cells accessed by computing $\text{range}(\mathbf{R})$.

Finally, we note **ReadRefs**, the union of all **Read** access relations for the program; **WriteRefs**, the union of all **Write** relations in the program; and the union of all access relations for the program $\text{ProgRefs} = \text{ReadRefs} \cup \text{WriteRefs}$.

Program execution order. A schedule is a relation used to specify the execution order of all statement instances. It maps points in the iteration domain to those in an integer set (the set of timestamps). As such, statement instances in the iteration domain are executed following the lexicographic ordering $<$ of their associated timestamp. $<$ is defined as $(a_1, \dots, a_n) < (b_1, \dots, b_m)$ iff there exists an integer $1 \leq i \leq \min(n, m)$ s.t. $(a_1, \dots, a_{i-1}) = (b_1, \dots, b_{i-1})$ and $a_i < b_i$.

The original program schedule is modeled using $2d + 1$ timestamps, where d is the maximal nesting depth in the program [Girbal et al. 2006]. For example, the schedules of R and S in Fig. 2 are:

$$\text{Sched}_R = \{R[i, j, k] \mapsto [0, i, 0, j, 0, k, 0]\}$$

$$\text{Sched}_S = \{S[i, j] \mapsto [0, i, 0, j, 1, 0, 0]\}$$

where each odd dimension of the output space is a scalar dimension whose value denotes the lexical abstract syntax tree (AST) ordering of the loops surrounding the statement. For statements surrounded by less than d loops, the even schedule components associated with the missing loops is set to 0. This approach seamlessly models imperfectly nested loops. A schedule can be constrained by the iteration domain of its statement, e.g., via $\text{Sched}_R \cap \mathcal{D}_R$. Consequently, the set of all distinct statement iterations in the program can be built by making the union of all schedules constrained by their respective statement iteration domain. **Sched** denotes this union.

4 SINGLE-LEVEL CACHE ANALYSIS

We begin in this section by modeling accesses to a single-level set-associative cache and then extend to hierarchical caches in Sec. 5. For a set-associative cache with associativity K and S sets, for each data read/write executed in the program, we represent the specific set in the cache to which the corresponding cache line L is mapped. If it is the first time L is accessed, the access is a miss. It is also a miss if K distinct cache lines that map to the same set have been accessed more recently than L . Otherwise, the access is a cache hit. Indeed, in a set-associative cache, each cache line maps to a specific set, which contains K slots. By reasoning on the sequence of events (i.e., data read/write events in the program) that lead to access of distinct cache lines mapping to the same set, we can build the set of events corresponding to the $K + 1$ distinct line accessed for a set, i.e., the set of events leading to a cache miss. This information can be built in a closed-form for affine programs.

4.1 Modeling Cache Accesses

To model events corresponding to accessing different cache lines, we must first translate array indices given by access relations into unique cache line indices and their associated set in the cache. We assume least-recently-used (LRU) replacement policy and define a cache and the mapping of virtual memory address to cache elements as follows.

Definition 4.1 (Set-associative cache). A set-associative cache C with associativity K , cache line (i.e., block) size of B bytes, and size n bytes contains S sets, with $S = n/B/K$. A virtual memory address addr maps to a unique line index $L_{id} = \text{floor}(\text{addr}/B)$, and the line maps to a unique set $S_{L_{id}} = L_{id} \% S$.

We now assume that all array accesses have been linearized. This is always possible if the array extents are known at compile time. For example, the linearization transformation from a two-dimensional access relation R^A for 2D array A with size sz along the fastest varying dimension (i.e., number of columns of A for row-major linearization as used in C) and start_A as its starting address is:

$$\text{Linearize} = \{[i, j] \rightarrow [m] : m = \text{start}_A + i * sz + j\}$$

Generalizing to n -dimensional arrays is straightforward.

The value of B and K are assumed to be known at compile time, which means the value of S also is a numerical constant. Given a virtual memory address, the unique cache line index is given by applying the relation:

$$\text{MemToLineId} = \{[m] \rightarrow [lid] : lid = \text{floor}(m/B)\}$$

The set to which a cache line maps is given by the relation:

$$\text{LineIdToCacheSet} = \{[lid] \rightarrow [cset] : cset = lid \% S\}$$

It follows the definition of the relation from an array access function to a cache set.

Definition 4.2 (Array to Cache set index). Given an access function F^A to array A of size $\vec{sz}$ and starting address start_A for a cache as defined in Def. 4.1, the associated cache line in C is identified by **AccessToLine** as:

$$\text{AccessToLine} = F^A \circ \text{Linearize}(\vec{sz}, \text{start}_A) \circ \text{MemToLineId}$$

The associated cache set in C is identified by **AccessToSet** as:

$$\text{AccessToSet} = \text{AccessToLine} \circ \text{LineIdToCacheSet}$$

With these relations, we can now reason in the cache space used by the program. For example, the set of distinct cache lines accessed in the program is modeled as:

$$\text{Clines} = \text{range}(\text{Sched}^{-1} \circ (\text{ProgRefs} \circ \text{AccessToLine}))$$

and can be counted immediately with $\#Clines$. This expression corresponds exactly to computing, in a general form and for arbitrary affine programs, the Distinct Line (DL) expression used in prior work for tile size selection [Sarkar and Megiddo 2000; Shirako et al. 2012]. Notably, it is significantly simpler than the original DL formulation, with no loss of accuracy.

4.2 Miss Events in Set-Associative Caches

Equipped with the ability to reason on cache lines and cache sets being accessed, we can now model *if* a particular access is a cache miss by reasoning on the sequence of accesses to distinct lines mapping to the same set.

Modeling the next iteration. Intuitively, we want to model consecutive (in terms of program execution order) accesses to the same cache set. Thus, we need to model a notion of sequencing in program execution by using the program schedule **Sched**. Specifically, we want to capture the point(s) j of the program iteration domain that is executed immediately after point i such that i and j have some properties, such as accessing the same cache set but a different cache line. Given an iteration i_1 , i_2 is consecutive to i_1 if there is no iteration i_3 in between. The non-existence of such point i_3 can be equivalently expressed using relations and set/relation differences.

The relation **LexSucc** maps a point to all points that are executed after it. We have:

$$\begin{aligned} \text{LexSucc} &= \{[i_1, \dots, i_n] \rightarrow [j_1, \dots, j_n] : (j) \text{ s.t.} \\ &\exists i, j \in \text{ProgDomain} \wedge \text{Sched}(i) < \text{Sched}(j)\} \end{aligned}$$

Similarly, we build **LexSuccEq**, the relation that generates points that are executed after an input point, including it; and **LexPrec**, the relation that generates points executed prior to an input point. Note these relations are each specifically built with respect to the particular program we are analyzing.

Modeling iterations accessing the same line/set. To model cache misses, we start by modeling a relation from the iteration domain to the cache line indices and cache sets:

$$\begin{aligned} \text{IterToLine} &= (\text{Sched} \cap \text{ProgDomain})^{-1} \circ \text{ProgRefs} \circ \text{AccessToLine} \\ \text{IterToSet} &= \text{IterToLine} \circ \text{LineIdToCacheSet} \end{aligned}$$

These relations associate to each point of the iteration space the set of cache lines (and cache sets) it accesses. To reason about iterations accessing the same lines or sets, we use the relation inversion to build a map from iterations to iterations accessing the same cache line/set, i.e.:

$$\text{SameLine} = \text{IterToLine} \circ \text{IterToLine}^{-1}$$

Modeling sets of relevant iterations. Relations such as **LexSucc** are meant to be intersected with other relations/sets that encode specific properties. For example, one can model the relation from an iteration i to all iterations accessing the same set *that are executed after i* as:

$$\text{SameLineSuc} = \text{SameLine} \cap \text{lexPrec}$$

A complete procedure is built by assembling the set of iterations that access different cache lines for each different set, retaining only the ones leading to the $K + 1^{\text{th}}$ distinct line accessed in a particular set, as this incurs a miss, as described below.

4.3 Algorithm for Miss Calculation

Algorithm 1 provides the complete procedure to obtain the set of iterations incurring in a cache miss event for a set-associative cache, as well as the count of these events.

Algorithm 1 SingleLevelMisses: Compute Misses for Single-level Cache

Input: Program: ProgRefs , ProgDomain , Sched
 Array parameters: $(sz_{Array}, start_{Array}) \forall Array$
 Cache parameters: B, S, K

Output: Set of iterations incurring a cache miss
 miss count
 // Apply schedule to iteration domain.

- 1: $\text{Prog} = \text{Sched} \cap \text{ProgDomain}$
 // Iterations to cache line indices:
- 2: $\text{IterToLine} = \text{Prog}^{-1} \circ \text{ProgRefs} \circ \text{AccessToLine}$
 // Iterations to cache sets:
- 3: $\text{IterToSet} = \text{IterToLine} \circ \text{LineIdToCacheSet}$
 // Compute cold misses, i.e., first access to a line.
- 4: $\text{coldmiss} = \text{range}(\text{lexmin}(\text{IterToLine}) \circ \text{Sched}^{-1})$
 // Iterations to iterations accessing the same cache line:
- 5: $\text{SameLine} = \text{IterToLine} \circ \text{IterToLine}^{-1}$
 // Keep only output iterations lexicographically after the input iteration:
- 6: $\text{SameLineSuc} = \text{SameLine} \cap \text{lexPrec}$
 // Keep only output iterations lexicographically equal or after to the input iteration:
- 7: $\text{SameLineSucEq} = \text{SameLine} \cap \text{lexSuccEq}$
 // Iterations to iterations accessing the same cache set:
- 8: $\text{SameSet} = \text{IterToSet} \circ \text{IterToSet}^{-1}$
 // Keep only output iterations lexicographically after the input iteration:
- 9: $\text{SameSetSuc} = \text{SameSet} \cap \text{lexPrec}$
 // Starting relation: all accesses to a cache set:
- 10: $\text{Miss} = \text{SameSetSuc} - (\text{SameLineSuc} \circ \text{lexSuccEq})$
- 11: **for all** assoc in $2, K$ **do**
- 12: $\text{Next} = \text{Miss} - (\text{Miss} \circ \text{lexPrec})$
 // Build the next iteration relation:
- 13: $\text{Miss} = \text{Miss} - \text{Next} \circ \text{SameLineSucEq}$
- 14: **end for**
- 15: $\text{Misses} = \text{coldmiss} \cup \text{range}(\text{Miss})$
- 16: **return** $(\text{Misses}, \#\text{Misses})$

Set specialization. Notably, building a formulation for all cache sets at once is unnecessary. To manipulate simpler systems, it is possible to embed an additional constraint, e.g., in LineIdToCacheSet , where the set is fixed to a constant value, e.g., $cset = x$, where $x \in \mathbb{N}$ is a known constant. Then, an iterative algorithm can be built, specializing SingleLevelMisses for each set value $S_i : [0, S]$ and forming the union of all Miss_i sets to form the complete set of all misses for all cache sets.

5 WRITE POLICIES AND HIERARCHICAL CACHES

In the preceding section, we modeled accesses to cache sets of a single-level cache from read and write operations in an affine program. To model a multi-level cache, we need to determine the read and write operations that are performed on the next level of the cache. The reads and writes that arrive at a cache at level $L + 1$ depend on the policies employed by the cache at level L .

Thus far, we have modeled caches that manage reads and writes in a symmetric fashion. In this section, we model caching approaches that employ alternative strategies to manage the write operations. When a cache encounters a write operation, it can: (a) *Write-through*: immediately

Table 1. Handling of write operations by the different cache writing policies considered

	Write-through	Write-back
Write allocate	All writes are cached. No dirty evictions. All writes forwarded to next cache level.	All writes are cached. All writes lead to dirty evictions. Only dirty evictions are seen as writes at next cache level.
No-write allocate	Write hits are cached. No dirty evictions. All writes forwarded to next cache level.	Write hits are cached. Write hits can lead to subsequent dirty evictions. Write misses and dirty evictions forwarded to next cache level.

forward the write to the next cache level, or (b) *Write-back*: write to the cache and forward to the next level only if the cache line is evicted.

In addition, a write to a line not in the cache (a write miss) can be handled in two ways: (a) *Write allocate*: allocate a line in the cache, read in the current contents of that cache line, and then perform the write in cache, or (b) *No-write allocate*: do not allocate a line but forward the write operation to the next cache level.

These policies change the cache contents at any point in time, impacting the cache hits, misses, dirty cache lines, and action taken on evictions. Table 1 describes the various scenarios.

Write-allocate write-through policy. The cache miss model for this policy is exactly the same as that described in Algorithm 1. All misses at cache level L with write-allocate, write-through policy become reads to the cache at level $L + 1$. All writes to such a cache, irrespective of whether they hit or miss in the cache at level L , become writes to the next cache level.

Write-allocate write-back policy. The cache miss model for this policy is exactly the same as that discussed in Algorithm 1. All misses at cache level L with write-allocate, write-back policy become reads to the cache at level $L + 1$. Evictions of dirty cache lines become writes to the next cache level. This is illustrated in Algorithm 2. The algorithm takes as input the **Miss** relation computed in line 13 of Algorithm 1, the relation from the iteration that allocates a cache line to the nearest following iteration that evicts it.

Algorithm 2 Compute Dirty Evictions: Writes To Next Cache Level For Write-Back Caches

Output: Dirty evictions

```

//evictions to writes that follow the immediate preceding allocation in execution order
1:  $\text{SameLineWriteSuccEq} = (\text{IterToLine} \circ \text{WriteIterToLine}^{-1}) \cap \text{lexSuccEq}$ 
2:  $\mathbf{A} = \text{Miss}^{-1} \circ \text{SameLineWriteSuccEq}$ 
   //miss iteration to all iterations that causally follow it
3:  $\mathbf{B} = (\text{range Miss}) \cap \text{lexSuccEq}$ 
   //dirty evictions to writes that made them dirty
4:  $\mathbf{C} = \mathbf{A} - \mathbf{B}$ 
   //dirty evictions to last write to before the eviction
5:  $\mathbf{D} = \text{lexmax } \mathbf{C}$ 
   //iteration of dirty eviction to the evicted cache line
6: return  $(\mathbf{D} \circ \text{IterToLine})$ 
```

No-write-allocate write-through policy. A cache with this policy caches all read operations and only the write operations that result in a cache hit. In particular, while write misses do not affect

the cache state, a write hit updates a cache line's priority, affecting subsequent evictions under LRU replacement policy. Therefore, modeling the misses for a no-write-allocate cache requires that we first model the write hits, and thus, in turn, write misses. To overcome this challenge, we present an approximate solution. We employ a cache miss formulation that is the same as in Algorithm 1, except that only the read references (**ReadRefs**) are used in formulating the cache misses. All read misses at cache level L become reads to the cache at level $L + 1$. All writes to a no-write-allocate write-through cache become writes to the next cache level.

No-write-allocate write-back policy. We approximate the cache misses using the same strategy as with a no-write-allocate write-through cache. We compute the read misses using only **ReadRefs**. All read misses at cache level L become reads to the cache at level $L + 1$. Write miss operations are forwarded to the next cache level. We compute the write misses using the Algorithm 1 but instead to compute K distinct reads between a write and its immediately preceding read. Specifically, to compute the write misses, line 5 in Algorithm 1 is changed to:

$$\text{SameLine} = \text{IterToLine} \circ \text{WriteIterToLine}^{-1}$$

where **WriteIterToLine** considers only iterations involving write operations. Evictions of dirty cache lines (Algorithm 2) and write misses become writes to the next cache level.

6 CACHE MODELING ACROSS PROGRAM PHASES

We have presented an analysis of cache misses for an affine program executed with an initial cold cache (all cache lines are invalid). Programs can consist of affine and non-affine phases. Our modeling approach can be used in the context of a whole program analysis framework for the affine program phases, with a conventional trace-based simulation being used for the non-affine phases. To enable such a “hybrid” analysis, we need to adapt our modeling approach to produce the actual final state of the cache at the end of an affine phase, so that it can be provided to a conventional cache simulator to model the subsequent non-affine program phase.

6.1 Final Cache State

The final cache state is defined by the set of cache lines in the cache, with information on whether or not they are dirty, and the recency order among the lines in each set. This information about the final cache state after the affine phase is needed to ensure correct simulation for cache accesses in the subsequent non-affine phase using standard cache simulation. Given a cache of associativity K , the cache lines resident in the set S in the final cache state can be computed as the last K distinct accessed cache lines that map to S .

Cache lines in final cache state. Algorithm 3 shows the steps involved in computing this information for a write-allocate cache.

It starts with computing the last iteration (A_0) that accesses each cache set using the lexicographic maximum operation. Then, all iterations that access the same cache line as this last iteration are removed from the set of all program iterations. This procedure is repeated until K iterations are computed. The union of all these cache lines (identified in terms of the iterations that access them) defines the cache lines in the final cache state. Note that this procedure works even if fewer than K distinct cache lines are accessed for some cache sets. Write-allocate caches allocate cache lines for both read and write accesses. Thus, the algorithm is invoked with the set of all program references (**ProgRefs**) to compute the final state.

Given the relation returned by the algorithm, the actual lines in the cache are given by:

$$\text{FinalCacheLines} = \text{range}(\text{FinalAccess} \circ \text{IterToLine})$$

Algorithm 3 FinalAccess(Refs)**Input:** Program: Refs, ProgDomain, SchedArray parameters: $(sz_{Array}, start_{Array}) \forall Array$ Cache parameters: B, S, K **Output:** For each cache set S , compute the references in Refs (identified by their schedule time) that access the last K distinct cache lines mapping to S

// Apply schedule to iteration domain

1: $\mathbf{Prog} = \mathbf{Sched} \cap \mathbf{ProgDomain}$

// Iteration to cache line:

2: $\mathbf{IterToLine} = \mathbf{Prog}^{-1} \circ \mathbf{Refs} \circ \mathbf{AccessToLine}$

// Iteration to cache set:

3: $\mathbf{IterToSet} = \mathbf{IterToLine} \circ \mathbf{LineIdToCacheSet}$

// Cache set to all iterations that access that set

4: $\mathbf{SetToIter} = \mathbf{IterToSet}^{-1}$ // Iteration i to all iterations that access the same cache line as i 5: $\mathbf{SameLine} = \mathbf{IterToLine} \circ \mathbf{IterToLine}^{-1}$ 6: $\mathbf{tmp} = \mathbf{SetToIter}$ 7: **for** assoc in 1, $K-1$ **do**8: $\mathbf{A}_{assoc} = \mathbf{lexmax} \ \mathbf{tmp}$ // Compute the last assoc-th access to each cache set

// Remove the assoc-th access from the list of accesses to be considered

9: $\mathbf{tmp} = \mathbf{tmp} - (\mathbf{A}_{assoc} \circ \mathbf{SameLine})$ 10: **end for**11: $\mathbf{A}_K = \mathbf{lexmax} \ (\mathbf{tmp})$ // Relation from each cache set to the iterations that access the last K distinct cache lines12: $\mathbf{FinalAccess} = (\bigcup_{i=1}^K \mathbf{A}_i)$ 13: **return** FinalAccess

Dirty cache lines. In the case of write-through caches, no cache lines need to be marked as dirty because all write operations are also reflected in the next cache level. In the case of write-back caches, cache lines that were written to after they were allocated in the cache for the last time need to be marked as being dirty in the final cache state. This computation is performed as shown in Algorithm 4.

Algorithm 4 Determine Dirty Cache Lines In The Final Cache State For Write-Back Caches**Output:** Set of dirty cache lines in the final cache state

// Determine the Misses associated with each cache line

1: $\mathbf{LineToMisses} = (\mathbf{IterToLine})^{-1} \cap \mathbf{Misses}$

// Determine the last miss for each line in the final cache state

2: $\mathbf{FinalAllocations} = \mathbf{lexmax} \ (\mathbf{LineToMisses} \cap \mathbf{FinalCacheLines})$

// Compute all write iterations that follow a given iteration

3: $\mathbf{SameLineWriteSuccEq} = (\mathbf{IterToLine} \circ (\mathbf{WriteIterToLine}^{-1})) \cap \mathbf{lexSuccEq}$

// Compute the domain of relation that compute all final misses that have a following write

4: $\mathbf{DirtyCacheLines} = \text{domain} \ (\mathbf{FinalAllocations} \circ \mathbf{SameLineWriteSuccEq})$ 5: **return** DirtyCacheLines

We now have the collection of dirty and non-dirty cache lines in the final cache state. The program's schedule gives us the last iteration that accesses each line in the final cache state. Accessing the cache lines in the final cache state—reading (writing) non-dirty (dirty) cache lines—will reproduce the final cache accesses in program order, preserving the LRU characteristics.

Together, the cache lines, dirty markers, and recency information in the cache final state mirror those that would be observed in an actual program execution.

6.2 Initial Cache State

The cache behavior of an affine program phase is dependent on the initial cache state when starting the phase. The cache state at any point can be encoded with an affine program that, if executed, would lead to such cache state. When composing two affine program phases, The final cache state of the first phase can be represented as an affine program to be executed before the second affine phase. This will correctly evaluate the cache behavior.

For arbitrary initial cache states, this program representation can be as large as the cache and expensive to manipulate in the form of integer sets. In this scenario, the first K distinct cache lines to be accessed in each set can be computed analogous to computing the final cache state. We enumerate each such access to check if it is a hit or a cold miss. The remaining accesses are modeled as presented in preceding sections.

The overall procedure to combine simulation of arbitrary program phases and the affine program phase involves the following steps:

- (1) Formulate a model of the first access to K distinct cache lines to each set in the affine phase.
- (2) For each such initial access, determine if it is a hit or a miss based on the initial cache state.
- (3) Model the affine phase using the integer set formulation.
- (4) Reconstruct the final cache state (Sec. 6.1).

We note that while the affine phase is modeled in a problem- and cache-size independent fashion, the interface between the affine and non-affine phases incurs costs proportional to the cache size.

7 EXPERIMENTAL EVALUATION

The following details an extensive evaluation of our framework.

7.1 Experimental Setup

The presented approach to cache modeling has been implemented in the PolyCache tool, which takes as input a C program and information about the cache parameters and size and starting address of arrays. PolyCache outputs the cache miss count of the program, and was used to generate all results presented in this paper. It is implemented using ISL-0.17 [Verdoolaege 2010b] (with barvinok-0.38 and pet-0.07), an integer set library for the polyhedral model. The Polyhedral Extraction Tool (PET) [Verdoolaege and Grosser 2012], a powerful SCoP extractor on top of Clang, detects affine regions and extracts the polyhedral model from C source code. ISL [Verdoolaege 2010a] is used to perform the operations in the algorithms described in previous section. We use the set-specialization approach discussed in Sec. 4.3 to compute the analysis for each set independently, and then accumulate the results as appropriate. These computations are run in parallel, using one process per set in the cache, since each set's behavior is independent of the others.

7.2 Evaluation of Hierarchical Set-Associative Cache

Benchmarks. We evaluate PolyCache's performance with the PolyBench/C benchmarking suite [Pouchet 2017b], which contains key numerical kernels and programs written as SCoPs. We also select some scientific computing benchmark kernels from HPGMG [Adams 2014], which is a representative high-performance computing benchmark based on geometric multigrid methods, and CCSD(T) kernels from the computational chemistry suite NWChem [Valiev et al. 2010].

Table 2 shows the various codes (64 in total) that were evaluated. We employed the standard dataset size provided for PolyBench/C, which typically leads to a data footprint exceeding 1 MB.

Table 2. Benchmarks and Summary of Hierarchical Set-Associative Cache

Benchmark	Description	Cache Accesses		Cache Misses		Time (sec)	
		L1	L2	L1	L2	Dinero	PolyCache
2mm	2 Matrix Multiplications	8589934592	2150100992	2149969920	2150092800	5780	16
3mm	3 Matrix Multiplications	12884901888	3225151488	3224954880	3225139200	8633	27
adi	Alternating Direction Implicit solver	1571123200	59063040	39376448	39066432	639	78
atax	Matrix Transpose Vector Multiply	134217728	5243392	4194560	1049600	54	4
bicg	BiCGStab Linear Solver Kernel	134217728	4571902	3523070	1049856	56	3
cholesy	Cholesky Decomposition	359486976	12019059	11495238	11339977	153	52
correlation	Correlation Computation	9437184	2097408	2097280	2097408	13	13
covariance	Covariance Computation	2156921856	1068171390	1067548735	1067582079	2096	38
doitgen	Multiresolution kernel (MADNESS)	1077936128	271090688	270828544	263168	527	4
durbin	Toeplitz system solver	67149812	50878969	34089213	34010918	182	128
dynprog	Dynamic programming (2D)	1727040000	91000172	46880104	36361199	729	9
fdtd-2d	2-D Finite Different Time Domain	732979500	32752000	22928000	22928000	302	25
fdtd-apml	Anisotropic Perfectly Matched Layer	765734400	8251741	5481573	4241215	382	2859
floyd-warshall	Shortest paths for weighted graph	7516192768	134215936	67107968	67166336	2829	64
gemm	Matrix-multiply C=a.A.B+b.C	4294967296	1075050496	1074984960	1075046400	2905	2
gemver	Vector Multiply & Matrix Addition	234893312	22793716	21621493	18881787	210	38
gesummv	Scalar, Vector & Matrix Multiply	134230016	3900412	3645950	2097920	57	7
gramschmidt	Gram-Schmidt decomposition	537133568	335954432	268706048	268714496	502	29
jacobi-1D	1-D Jacobi stencil computation	5998800	375000	250000	1250	2	1
jacobi-2D	2-D Jacobi stencil computation	167117440	7851520	5235200	5235200	70	4
lu	LU decomposition	1431130624	45261695	22647007	22671492	562	5
ludcmp	LU decomposition	719498750	371973118	370932158	365399675	577	229
mvt	Matrix Vector Product & Transpose	134217728	17844726	17844214	17832188	188	6
reg-detect	2-D Image processing	11656000	596	300	300	4	2
seidel	2-D Seidel stencil computation	208896800	2618880	1310720	1310720	75	1
symm	Symmetric matrix-multiply	3215988736	2143794634	1607465303	1603222155	3463	323
syr2k	Symmetric rank-2k operations	8592031744	166429766	166298694	134537936	3498	151
syrk	Symmetric rank-k operations	4294967296	67238980	67173444	67169874	1677	39
trisolv	Triangular solver	33566720	530944	530688	527104	13	3
trmm	Triangular matrix-multiply	2145386496	34175998	34110526	34106357	838	36
chebyshev	Chebyshev smoother in HPGMG	2228640704	536867608	471319352	471319352	1018	2861
heat	2-D heat equation stencil	5836644000	166594560	133301760	133301760	2507	298
minimgm	Compact Geometric Multigrid	7341404672	3703992592	721079584	782498816	820	710
poisson	Poisson solver stencil computation	14856912000	166855680	133562880	133562880	4577	1890
j3d7pt	3-D Jacobi stencil with 7 points	4244832000	166594560	133301760	133301760	1901	195
j3d13pt	3-D Jacobi stencil with 13 points	1573158144	786627840	3179606544	3179606544	3731	390
j3d27pt	3-D Jacobi stencil with 27 points	10612080000	166855680	133562880	133562880	5844	963
ccsd_d1_1	NWchem tensor contraction kernel1	1073741824	270598144	269549568	2777765	444	2
ccsd_d1_2	NWchem tensor contraction kernel2	1073741824	271581184	270532608	2263970	439	2
ccsd_d1_3	NWchem tensor contraction kernel3	1073741824	287309824	286261248	2587673	449	2
ccsd_d1_4	NWchem tensor contraction kernel4	1073741824	270598144	269549568	1723455	537	2
ccsd_d1_5	NWchem tensor contraction kernel5	1073741824	271581184	270532608	1505958	529	2
ccsd_d1_6	NWchem tensor contraction kernel6	1073741824	287309824	286261248	1830622	534	2
ccsd_d1_7	NWchem tensor contraction kernel7	1073741824	270598144	269549568	1784865	494	2
ccsd_d1_8	NWchem tensor contraction kernel8	1073741824	271581184	270532608	1552533	484	2
ccsd_d1_9	NWchem tensor contraction kernel9	1073741824	287309824	286261248	1877197	491	2
ccsd_d2_1	NWchem tensor contraction kernel10	1073741824	3211264	2162688	1444382	397	2
ccsd_d2_2	NWchem tensor contraction kernel11	1073741824	18939904	17891328	1447441	418	2
ccsd_d2_3	NWchem tensor contraction kernel12	1073741824	18954304	17905728	1754040	415	2
ccsd_d2_4	NWchem tensor contraction kernel13	1073741824	3211264	2162688	1489164	397	2
ccsd_d2_5	NWchem tensor contraction kernel14	1073741824	18939904	17891328	1508608	418	2
ccsd_d2_6	NWchem tensor contraction kernel15	1073741824	18954304	17905728	1813508	407	2
ccsd_d2_7	NWchem tensor contraction kernel16	1073741824	3149824	2101248	2101248	399	2
ccsd_d2_8	NWchem tensor contraction kernel17	1073741824	18886144	17837568	2464140	411	2
ccsd_d2_9	NWchem tensor contraction kernel18	1073741824	18893824	17845248	2708672	410	2
ccsd_s1_1	NWchem tensor contraction kernel19	67108864	2162704	1114128	1114128	28	1
ccsd_s1_2	NWchem tensor contraction kernel20	67108864	2162704	1114128	1114128	27	1
ccsd_s1_3	NWchem tensor contraction kernel21	67108864	2162704	1114128	1114128	27	1
ccsd_s1_4	NWchem tensor contraction kernel22	67108864	2162704	1114128	1114128	27	1
ccsd_s1_5	NWchem tensor contraction kernel23	67108864	2162704	1114128	1114128	28	1
ccsd_s1_6	NWchem tensor contraction kernel24	67108864	2162704	1114128	1114128	28	1
ccsd_s1_7	NWchem tensor contraction kernel25	67108864	2101264	1052688	1052688	27	1
ccsd_s1_8	NWchem tensor contraction kernel26	67108864	2101264	1052688	1052688	27	1
ccsd_s1_9	NWchem tensor contraction kernel27	67108864	2101264	1052688	1052688	28	1

The number of data access operations range from millions to billions. These benchmarks provide

a variety of challenges, including out-of-cache dataset sizes, multiple arrays, imperfectly nested loops, triangular loops, non-uniform reuse distance, etc.

Tools and setup. As a comparison point for both correctness and performance of PolyCache, we used the DineroIV simulator [Edler and Hill 1999], a uniprocessor cache simulator that can handle hierarchical set-associative caches, as well as numerous replacement and write policies. All experiments were performed on a cluster of Intel Xeon E5640 processors running at 2.67 GHz with 32 KB L1 cache. The programs were all compiled using a GCC-6.1.0 compiler with -O3 optimization. We parallelized the PolyCache computation across sets, as mentioned above, using as many processes as sets in the cache and report the time to completion.

Evaluation. The first experiment aims to validate the correctness of the framework against a trace-based simulator for a real-life complex scenario: a 2-Level cache memory hierarchy with 32 KB 4-way set-associative L1 cache and 256 KB 4-way set-associative L2 cache, both implementing a write-allocate write-back policy. Both caches have 64-byte cache line size. Table 2 compares the cache accesses/misses computed by DineroIV with the accesses/misses obtained with PolyCache, as well as the execution time (in seconds) for both systems. The number of data read/writes issued by the program is not reported herein. Instead, we focus solely on the cache access/miss count.

A key observation is that *for all situations, there is a perfect match between DineroIV and PolyCache*. We performed this validation for all experiments reported in this paper, we always obtained an exact match in the number of accesses/misses reported by DineroIV and PolyCache for the non-approximated schemes.

To illustrate the intricacy of the situations modeled by our analytical framework, we detail results for the Floyd-Warshall benchmark. Of note, the total number of misses in L2 is actually slightly higher than the number of misses in L1, 64,000 higher exactly. This is the expected result, but it results from a complicated effect. First, cold misses are the same for L1 and L2. Capacity misses are also the same. Floyd-Warshall uses a matrix of size 1024×1024 , and the reuse distance between two iterations of the outer loop is larger than both the 32- and the 256-KB cache. Moreover, there is nearly no conflict miss either in L1 or L2 for this experiment due to the large number of capacity misses. Thus, cache misses for L1 and L2 are expected to be mostly the same, yet there are 64,000 more misses in L2. In the 2-level cache hierarchy with write-back policy implemented for both L1 and L2, a line evicted from L1 is written to L2 to maintain coherency between the caches, and a line evicted from L2 is written back to memory. Writing a line evicted from L1 in L2 is a cache hit (write hit) only if the line is already in L2. Otherwise, it will generate an L2 miss (write miss). Furthermore, it may lead to evicting a line in L2 if the set to which this line maps to in L2 is already full with other lines. Therefore, events in L1, such as dirty line eviction, may result in a cache miss in L2. In fact, there are 64,000 lines written in Floyd-Warshall (using a matrix of size 1024×1024), which corresponds exactly to the additional 64,000 misses in L2.

We note that there is significant variability in execution time across benchmarks and methods. As expected, for a trace-driven simulator, the DineroIV time is proportional to the trace size. For example, the 2mm benchmark comprised of a sequence of two gemm matrix multiplication takes about twice as much time to simulate than gemm (there are about a billion operations in gemm). Our analytical approach is much faster, up to $365\times$ faster in this case. However, there exists some cases (2 over 64, shown in bold in Table 2) where the compile-time approach is slower than the simulation. Unfortunately, predicting *a priori* the execution time of PolyCache is infeasible. By design, we operate on integer linear programs, where even a simple polyhedron emptiness test is NP. However, for many typical cases the observed complexity is polynomial in practice. This issue is addressed in greater detail in Sec. 7.6.

7.3 Evaluation of Write Policies

As discussed in Sec. 5, the framework is capable of handling exactly various write-allocate policies. In PolyCache, we have implemented the write-allocate write-back policy used in Table 2, and evaluate below our implementation for the no-write-allocate write-back policy. Table 3 compares the cache misses computed by PolyCache (L1 and L2 columns) with DineroIV (% diff. columns, -0.1% means PolyCache under-approximated the misses by 0.1%). We use L1 and L2 caches of same size and associativity as in Table 2, but both using instead the no-write-allocate-write-back policy.

Table 3. Summary of No-Write Allocate Write Back Cache

Benchmark	Cache Misses				Time (sec)	
	L1	L1 %diff	L2	L2%diff	Dinero	PolyCache
3mm	3224954880	0%	3225139200	0%	8663	16
adi	39376448	0%	39066432	0%	633	107
atax	4194560	0%	1049600	0%	54	8
bicg	3523070	0%	1049856	0%	57	3
correlation	2097280	0%	2097408	0%	5	199
covariance	1067548736	0%	1067582080	0%	2024	34
doitgen	270828544	0%	263168	0%	525	8
dynprog	71540106	0%	60921211	0%	737	4
fdtd-2d	22928000	0%	22928000	0%	299	30
floyd-warshall	67107968	0%	67104232	-0.1%	2823	583
gemm	1074984960	0%	1075046400	0%	2869	2
gemver	21736183	0%	18882296	0%	209	41
gesummv	3645950	0%	2097920	0%	56	18
gramschmidt	268706560	0%	268706248	-0.003%	465	171
jacobi-1d	2124600	0%	11248	0%	2	26
jacobi-2d	44398240	0%	44398240	0%	79	4
ludcmp	371409006	0%	365825035	0%	592	318
mvt	17844214	0%	17832188	0%	170	14
seidel-2d	1310720	0%	1310720	0%	73	2
syr2k	166298694	0%	134468056	-0.1%	3487	7720
syrk	67173444	0%	67169874	0%	1685	764
trisolv	530944	0%	527488	0%	12	2
chebyshev	409900120	0%	409900120	0%	1003	3135
heat	630612960	0%	630612960	0%	2573	324
minigmg	659660352	0%	659660352	0%	803	776
poisson	630874080	0%	630874080	0%	4650	3650
j3d7pt	630612960	0%	630612960	0%	1953	208
j3d13pt	3179606544	0%	3179606544	0%	3780	1664
j3d27pt	630874080	0%	630874080	0%	5986	4749
ccsd_d1_1	269549568	0%	2531989	0%	438	3
ccsd_d1_6	286261248	0%	1798021	0%	534	4
ccsd_d2_1	2162688	0%	1444382	0%	398	3
ccsd_d2_9	17845248	0%	2466976	0%	405	3
ccsd_s1_1	1114128	0%	1114128	0%	27	2
ccsd_s1_9	1052688	0%	1052688	0%	27	2

We make three observations. First, as the formulation is significantly more complex for this write policy, the execution time of PolyCache significantly increases compared to Table 2. In fact, 8 benchmarks (2mm, cholesky, durbin, fdtd-apml, lu, reg-detect, symm and trmm) timed out, that is PolyCache did not complete the calculation within 120 minutes, our timeout for this experiment. Second, for most benchmarks in our experimental setup, there is 0% difference in the cache miss count computed by PolyCache compared to DineroIV's output. That is, for these experiments, approximating the no-write-allocate by only considering read references does not change the miss count at either L1 or L2. This highlights that ignoring write-hits on LRU in these experiments does not change the miss count. Third, only three benchmarks (shown in bold) shows differences between our approximation and DineroIV: PolyCache under-approximates the miss count by less

than 0.1%. Note all CCSD benchmarks showed 0% difference between PolyCache and DineroIV, we only display representative ones.

7.4 Evaluation of Loop Tiling

We now report cache miss data when loop tiling is applied to the programs. We process all 30 Polybench benchmarks with the PoCC compiler [Pouchet 2017a], using the Pluto algorithm [Bondhugula et al. 2008] to automatically compute a complex sequence of loop transformation to make loop tiling legal and then apply loop tiling wherever valid, using a tile size of 32. This transformation can dramatically increase the code size with up to hundreds of syntactic statements and tens of loop nests in the generated code. Table 4 reports the number of misses without and with this tile size selection for a single-level 32 KB, 4-way set-associative L1 cache. All of the cache miss numbers were validated for correctness against DineroIV.

Table 4. Summary of Tiling (PolyCache is Poly. below)

Benchmark	L1 Cache Misses		Time (sec)		Benchmark	L1 Cache Misses		Time (sec)	
	No tiling	Naive tiling	Dinero	Poly.		No tiling	Naive tiling	Dinero	Poly.
2mm	2149969920	4362076160	4184	60	gemver	21621493	22011647	121	50
3mm	3224954880	519479636	6195	485	gesummv	3645950	33604605	70	13
adi	39376448	454822592	663	385	gramschmidt	268706048	268713488	288	52
atax	4194560	4194560	57	3	jacobi-1D	250000	5000	20	16
bicg	3523070	18530940	67	20	jacobi-2D	5235200	4198090	350	330
cholesy	11495238	11495301	161	11	lu	22647007	370408773	766	72
correlation	2097280	4100	982	38	ludcmp	370932158	34195872	656	83
covariance	1067548735	539130944	983	19	mvt	17844214	18531068	68	23
doitgen	270828544	1327104	463	80	reg-detect	300	300	4	2
durbin	34089213	34152702	102	91	seidel	1310720	2152592	77	14
dynprog	46880104	47070106	746	10	symm	1607465303	1607465299	3858	148
fdtd-2d	22928000	15187943	333	436	syr2k	166298694	2161626400	3875	219
fdtd-apml	5481573	250113688	741	3403	syrk	67173444	1077905600	1879	62
floyd-warshall	67107968	99645071	3123	176	trisolv	530688	535167	13	13
gemm	1074984960	71303168	2000	6	trmm	34110526	49805889	872	318

In two cases (Table 4, in bold) the execution times are longer than simulation, for example, fdtd-apml, where it takes almost 1 hour to complete. The reason is the same as explained earlier. Due to the inherent difficulty of predicting when this analytical formalism will lead to prohibitive execution time, we advocate a timeout-based approach. Users may set the timeout to a few minutes. If no answer is provided, users can, for example, fall back to use of approximation heuristics (refer to Sec. 7.5).

7.5 Evaluation of Approximation Heuristics

Here, we discuss possible acceleration techniques with the goal of reducing static analysis time, while allowing for an approximate result. Thus, the methods presented in this section do not guarantee exact miss count, is in contrast to the results presented earlier that are exact by definition.

Per-array analysis. We have developed a simple heuristic that analyzes only the references to a single array in the program at a time, and the process is repeated for all arrays. In other words, we only capture self-interference in this process. Table 5 reports the number of misses computed for each array in the program, where for programs with more than four arrays, we have summed the number of misses obtained for all subsequent arrays. We compare the sum obtained by approximation of misses in the program based solely on self-interference, versus the exact number of misses in the full program, as well as the execution time of both this heuristic and the

full program analysis with PolyCache. The average speedup of the analysis time is 8x, and, in many situations, there are no cross-interference misses between the various arrays in the benchmark.

Table 5. Summary of Per-Array Heuristic

Benchmark	Number of Cache Misses							Time (sec)		Speedup
	Array 1	Array 2	Array 3	Array 4 - ...	Sum	Actual	% of diff.	Sum	Full prog.	
3mm	65536	1073741824	65536	2147745792	3221618688	3224954880	0%	1	6	5.3×
adi	13053312	13107648	13107712		39268672	39376448	0%	17	57	3.4×
bicg	1048576	256	256	512	1049600	3523070	-70%	0	3	10.9×
covariance	1066991616	64	557023		1067548703	1067548735	0%	3	11	4.1×
dynprog	156	44800000	156		44800312	46880104	-4%	1	4	3.8×
gemm	65536	1073741824	65536		1073872896	1074984960	0%	0	1	2.9×
gemver	18874368	256	256	1536	18876416	21621493	-13%	1	23	17.3×
jacobi-2d	2618880	2616320			5235200	5235200	0%	1	3	3.1×
ludcmp	370931709	64	64	64	370931901	370932158	0%	17	70	4.1×
trisolv	526336	256	256		526848	530688	-1%	0	3	5.8×

However, for benchmarks such as bicg, only considering self-interference is not a valid approach. In this case, we compute capacity misses because some arrays can fit into the cache separately but not all together, resulting in the difference in cache miss numbers.

Table 6. Summary of Set-0 Heuristic

Benchmark	Min	Max	Avg.	Stdev	Sum	Set-0 approx	% diff.	Time Set-0 (sec)	Time All Sets (sec)
2mm	16795648	16796672	16796640	179	2149969920	2149842944	0%	7	8
3mm	25193472	25195008	25194960	268	3224954880	3224764416	0%	6	6
adi	306803	358054	307629	6375	39376448	39270784	0%	56	57
atax	32770	32770	32770	0	4194560	4194560	0%	4	4
bicg	27522	27524	27524	0	3523070	3523072	0%	3	3
cholesy	552	262641	89807	79529	11495238	33618048	192%	6	7
correlation	12602	20168	16385	2225	2097280	2581504	23%	16	19
covariance	8336440	8344009	8340224	2225	1067548735	1068033152	0%	7	11
doitgen	2115584	2116608	2115848	450	270828544	270794752	0%	4	4
durbin	134303	396386	266322	77049	34089213	17190784	-50%	14	33
dynprog	4	650001	366251	188950	46880104	23040000	-51%	4	4
fdtd-2d	179100	179150	179125	25	22928000	22931200	0%	6	10
fdtd-apml	32832	68796	42825	10183	5481573	5197312	-5%	318	2657
floyd-warshall	524281	524281	524281	0	67107968	67107968	0%	15	22
gemm	8397824	8398336	8398320	89	1074984960	1074921472	0%	1	1
gemver	168910	168918	168918	1	21621493	21621120	0%	23	23
gesummv	28482	28484	28484	0	3645950	3645696	0%	2	3
gramschmidt	2099204	2099328	2099266	37	268706048	268698112	0%	11	18
jacobi-1D	1800	2000	1953	85	250000	256000	2%	1	1
jacobi-2D	40900	40900	40900	0	5235200	5235200	0%	3	3
lu	8128	262639	176930	77374	22647007	1040384	-95%	3	5
ludcmp	323544	4202981	2897907	1166399	370932158	536934272	45%	54	70
mvt	139403	139408	139408	1	17844214	17843584	0%	3	4
reg-detect	0	4	2	1	300	512	71%	0	1
seidel	10240	10240	10240	0	1310720	1310720	0%	1	1
symm	4280917	20824324	12558323	4859581	1607465303	550904576	-66%	50	83
syr2k	1298961	1299472	1299209	256	166298694	166267008	0%	12	17
syrk	524792	524794	524793	1	67173444	67173504	0%	2	3
trisolv	2114	6178	4146	1187	530688	790784	49%	2	3
trmm	8441	524535	266488	151925	34110526	67140352	97%	4	5

Sampling cache sets. We also evaluate another heuristic based on the key observation that there is often a high similarity between the misses across different sets, especially for regular applications. Table 6 summarizes our set-0 heuristic. This heuristic is limited to computing the misses for the cache set-0 and estimates the total misses by multiplying the resulting value by the number of sets (128 in this experiment). We show several metrics on the number of misses for all of the sets,

such as the minimum, maximum, average, and standard deviation, and report the miss count error (% difference) and execution time of the heuristic compared to the execution time of PolyCache. Note an essential difference in the setups between the set-0 heuristic and full program analysis. For the set-0 heuristic, we compute only one set, meaning PolyCache is executed on a single core without any parallelism. In contrast, for “Time all sets,” the time reported is the time for PolyCache to complete when using S -way parallelism, i.e., using S cores.

First, we observe a variety of benchmarks where the set-0 heuristic provides actually near-exact results. These illustrate regular cache access patterns. Conversely, the approach may dramatically fail, such as for LU, where the number of misses is under-approximated by 20x. Yet, these result show a surprisingly good approximation can be obtained very quickly. For all but two (LU and symm), the cache misses are off by a factor of 2 or less with 18/30 benchmarks giving nearly exact value. We believe these results pave the way for additional focused experimental studies of the regularity of cache accesses for affine programs with the goal of determining when the cache behavior may be approximated effectively by the behavior of a single (or few) set(s). As future work, we will investigate sampling heuristics, computing the value of a few randomly chosen sets and using their average miss instead of remaining limited to set-0.

7.6 Time Complexity of PolyCache

Table 2 shows very large execution time variation across benchmarks for PolyCache. The explanation is simple to state but the execution time is infeasible to characterize *a priori*. In this work, we rely extensively on manipulating sets modeled by Presburger formulae, in particular, on counting the number of points in these sets. This computation is NP-hard. As such, we obviously see situations where a much more complex system (i.e., miss set) is generated and counting takes a very long time, despite an excellent implementation from the Barvinok library [Verdoolaege 2007; Verdoolaege et al. 2007] integrated in ISL. To the best of our understanding, it is impossible to predict *a priori* the execution time of PolyCache because it relates to the irregularities in the generated cache miss set, not to simple input program features, such as the number of loops or problem size.

In this section, we provide details on two representative cases, Gemm and Symm. While both have roughly similar numbers of cache misses and similar DineroIV execution times (shown in Table 2), for Symm, PolyCache is 150× slower to compute the solution than for Gemm. For each operator type invoked by PolyCache, Table 7 details the number of times this operator is used (Nb Ops) and the total time to execute the Nb Ops calls.

For both computations, the operators are called exactly the same number of times. This is expected, given that it follows Algorithm 1 using the same cache configuration, but it highlights the fact that the number of operator calls does not determine execution time. Interestingly, we see that the time taken for set difference explodes for Symm. This is due to the shape of the sets being subtracted. The result is not a convex set but a union of small, distinct convex sets (pieces). For Symm, the number of pieces explodes, indicating high irregularity in the Miss set. The coalesce operation similarly explodes. Coalescing is a form of simplification of the representation, and tries to combine smaller pieces into a larger convex piece. It is invoked prior to counting (the final operation of our algorithm), i.e., on the result of the difference operation. Intuitively, the execution time appears to be driven by how complicated the intermediate polyhedral sets we manipulate are, which itself strongly depends on the cache configuration being modeled. Because predicting when PolyCache’s execution time will be very high is impractical, the approximation techniques outlined herein can prove critical to quickly compute a solution.

Table 7. Per-operator Execution Time
 s : set, m : map, x : set or map, p : integer

Operator	Syntax	Execution time (sec)			
		Gemm	No. Ops	Symm	No. Ops
Domain	$s := \text{domain } m$	0.005	3	0.080	3
Range	$s := \text{range } m$	0.000	4	0.001	4
Composition	$m3 := m1 \circ m2$	0.206	28	7.657	28
Inverse	$m2 := m1^{-1}$	0.058	6	0.132	6
Application	$s2 := m(s1)$	0.001	1	0.024	1
Union	$x3 := x1 \cup x2$	0.005	2	0.021	2
Intersection	$x3 := x1 \cap x2$	0.062	6	1.697	6
Difference	$x3 := x1 - x2$	1.136	16	113.030	16
Coalesce	$s2 := \text{coalesce } s1$	0.717	12	218.142	12
Counting	$p := \text{card } x$	0.081	2	1.518	2
Total time		2.273		342.303	

8 DISCUSSION AND OTHER USE CASES

Discussion and limitation. We have presented a closed-form formulation of the cache behavior for hierarchical set-associative caches. The formulation is independent of the problem and cache sizes. However, the formulation is influenced by the associativity of the cache being modeled. The integer set formulation has an exponential worst-case complexity [Oppen 1978]. As observed in the preceding sections, this can occasionally result in very high evaluation costs.

A study of exact cache behavior requires mapping the array addresses to linearized memory addresses. While polyhedral approaches can deal with parametric problem sizes, this linearization requirement forces the use of constant array sizes. Therefore, similar to all prior work addressing this problem, the exact set of misses must be computed for each constant array size.

In addition to linearized memory addresses, our formulation requires that the function mapping an address to a cache line be affine. This restricts it to virtually indexed caches. Several architectures (e.g., SPARC, PowerPC, ARM) employ virtually indexed caches for portions of their cache hierarchy, enabling direct use of our formulation.

Even with physically indexed caches where the virtual-to-physical mapping is determined at runtime, our approach can be useful when applications employ huge pages (e.g., 1 GB pages). This mapping, obtained after executing the program, can be extracted as a trace. Given the small number of virtual and physical pages when using huge pages, tracing this mapping is much cheaper than tracing every cache miss. Given a mapping extracted from a run, one can construct the virtual address to cache set mapping using piece-wise affine functions (one per huge page) and reconstruct the cache behavior for that run using our formulation.

Cache vulnerability analysis. Cache vulnerability factor (CVF), the probability that soft errors (e.g. bit flips) impacting the cache get propagated to other hardware components, was introduced by Zhang [2005] to evaluate the reliability of the cache hierarchy. Its calculation involves tracking, for each bit, the fraction of program execution time during which a bit flip may escape the cache into an architecturally visible state (registers or memory). For example, corruption of a dirty cache line that is written back to memory leads to an escaping error, whereas a non-dirty cache line that is just replaced can mask bit flips. Vulnerability calculation requires exact information about the operations on each data word of the cache—reads, writes, misses, evictions—throughout program execution, including the time when each event occurs. This is typically achieved using expensive

simulations, where the lifetime of cache lines can be precisely monitored via hooks in the simulator. However, it is usually too expensive for practical use in design space exploration to optimize vulnerability. Instead, our formulation can be extended to perform detailed lifetime analysis of each cache line so as to characterize vulnerability of the entire cache hierarchy at compile time.

Array padding. Inter-array padding has been shown to impact cross-interference, or, conflict misses between arrays, and can be impacted by the padding between arrays (e.g., [Hong et al. 2016; Rivera and Tseng 1998]). The inter-array offset can be cast as a linear parameter that needs to be determined to minimize the cache misses.

Cache design analysis. Our formulation enables two optimizations when exploring cache configurations by varying its associativity (cache lines per set) and number of sets, for a given input program. First, as seen in Algorithm 1, an analysis of an input program for a K -way cache with a given number of sets incorporates an analysis of caches with the same number of sets but with lower associativity. Therefore, only the max-associativity for a given number of sets needs to be evaluated. On the other hand, a simulation may have to consider all combinations of associativity and number of sets. Second, given configuration for a cache at level L , our approach can present the reads and writes that reach the cache at level $L + 1$. This affords reuse of the cache evaluation at level L to explore the choice of cache configurations at level $L + 1$. Together, these can significantly reduce the cost to explore the space of possible cache configurations.

Bounding worst-case execution time. The execution time of programs executing on real-time systems is bounded to guaranteed response times. Cache-based systems complicate this analysis by introducing unpredictable data access latencies [Alt et al. 1996]. Our exact analysis of cache misses can assist in more accurate determination of cache miss costs and, thus, better bounds on the execution times.

9 RELATED WORK

Cache simulation. Simulators such as Dinero [Edler and Hill 1999] and Sniper [Carlson et al. 2014] characterize an application's cache behavior with varying degrees of fidelity. Also, approaches have been developed to simulate multiple levels of associativity simultaneously [Hill and Smith 1989]. In general, simulation cost is proportional to the number of executed operations, with the approaches either executing the program or manipulating large memory reference traces [Bao et al. 2017; Wang and Baer 1990]. These approaches do not provide a closed-form model that can be used by compile-time optimizers.

Approximate analysis of cache behavior. Approximate metrics, such as reuse distance, have been used as an approximate measure of cache reuse by compile-time optimization techniques [Ahmed et al. 2001; Bondhugula et al. 2008; Carr et al. 1994; Kelly and Pugh 1993; Lim and Lam 1997]. Ferrante et al. [1991] estimate the number of distinct cache lines accessed by a loop nest. Similar analyses have been developed to predict cache miss ratios for set-associative and fully associative caches [Agarwal et al. 1989; Harper et al. 1999; Singh et al. 1992]. Other approaches to estimate cache misses include probabilistic estimation of an array reference incurring a miss [Fraguela et al. 1999, 2003] and sampling the iteration to approximate the absolute miss ratio for each static array reference [Vera et al. 2004], dynamic memory reference [Berg and Hagersten 2004], and individual instructions [Fang et al. 2005, 2004]. These techniques provide inexact cache behavior analysis while being potentially applicable to a larger class of programs. Frumkin and Van W. [2002] developed bounds on cache misses for stencil operations on rectangular grids. Our approach can generate exact cache miss information for a larger class of programs that includes such stencil codes.

Exact analysis of cache misses. Cascaval and Padua [2003] estimate cache misses using stack distances to construct a stack histogram. The model is accurate for fully associative caches with LRU replacement policy and provides approximate solutions for set-associative caches. This work is restricted to perfectly nested loops and dependence characterized as distance vectors. Beyls and D'Hollander [2005] develop a compile-time analysis of cache misses for fully associative caches by analytical modeling of reuse distances, defined as the number of other distinct cache lines accessed between two successive references to a particular cache line. By identifying cache misses and hits in certain parts of the program, Alt et al. [1996] and Bao et al. [2014] model cache behavior using abstract interpretation to improve worst-case execution time bounds.

Ghosh et al. [1998, 1999] present cache miss equations (CMEs), an approach to counting the exact number of cache misses in perfectly nested loop nests in the form of the number of solutions to a set of affine equations. The number of solutions to the cache miss equations gives an exact count of the set of misses in direct-mapped and set-associative caches. The CMEs have been extended for use in a variety of contexts, including bounding worst-case data cache behavior and execution time for real-time systems [Ramaprasad and Mueller 2005] and cache vulnerability analysis [Shrivastava et al. 2010]. Solving cache miss equations is expensive [Vera et al. 2005; Vera and Xue 2002], motivating the design of approximate counting strategies [Ghosh et al. 1999; Vera et al. 2003]. Vera et al. [Vera and Xue 2002; Xue and Vera 2004] extend the use of CMEs to count cache misses in programs consisting of imperfectly nested loops and non-recursive subroutine calls using abstract inlining and loop normalization, sinking with if-conditionals. CMEs and approaches that build on them are restricted to handling dependences characterized as distance vectors.

Chatterjee et al. present an exact solution for direct-mapped caches using Presburger formulae [Chatterjee et al. 2001]. Their formulation for set-associative caches is limited to interior misses, but it appears extensible to boundary misses. As discussed previously, such formulation cannot be used to build the ordered set of events (e.g., cache miss) needed for hierarchical caches.

10 CONCLUSION

Cache behavior analysis is an essential tool for program optimization. Optimizing the data traffic for cache-equipped processors has been investigated both via software transformations and hardware cache design space exploration. However, the current state of practice to analyze cache behavior is limited to actual execution and/or cache simulation, both with time complexity proportional to the number of executed operations, making compile-time optimization of cache behaviors difficult.

In this work, we proposed a fully compile-time approach to static analysis of cache behavior for hierarchical set-associative virtually-indexed caches for the class of affine (polyhedral) programs. The framework was validated by implementation of a tool to analyze affine C programs to compute cache misses at any level, producing cache statistics equivalent to trace-based cache simulators.

Since the framework makes extensive use of complex polyhedral operations with NP-hard worst-case complexity, long analysis times remain possible. As future work, we plan to investigate additional acceleration schemes for the formulations, as well as the development of compiler analyses and optimizations driven by cache miss estimations.

ACKNOWLEDGMENTS

We thank the anonymous referees for the feedback and many suggestions that helped in improving the presentation. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Awards 66905 and DE-SC0014135, program manager Lucy Nowell, by the U.S. National Science Foundation through awards 1513120 and 1731612, and by computational resources from the Ohio Supercomputer Center. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

REFERENCES

- M. Adams. 2014. HPGMG: a benchmark for ranking high performance computing systems. (2014). <https://www.hpgmg.org/>
- A. Agarwal, J. Hennessy, and M. Horowitz. 1989. An Analytical Cache Model. *ACM Transactions on Computer Systems* (1989), 184–215.
- N. Ahmed, N. Mateev, and K. Pingali. 2001. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming* (2001), 493–544.
- M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. 1996. Cache behavior prediction by abstract interpretation. In *International Static Analysis Symposium (SAS'96)*. 52–66.
- W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, N. Pouchet, F. Rastello, and P. Sadayappan. 2016a. Static and Dynamic Frequency Scaling on Multicore CPUs. *ACM Transactions on Architecture and Code Optimization* (2016), 1–26.
- W. Bao, S. Krishnamoorthy, L. Pouchet, F. Rastello, and P. Sadayappan. 2016b. PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)* (2016), 539–554.
- W. Bao, P. Rawat, M. Kong, S. Krishnamoorthy, L. Pouchet, and P. Sadayappan. 2017. Efficient Cache Simulation for Affine Computations. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'17)*.
- W. Bao, S. Tavarageri, F. Ozguner, and P. Sadayappan. 2014. PWCET: Power-Aware Worst Case Execution Time Analysis. In *43rd International Conference on Parallel Processing Workshops*. 439–447.
- E. Berg and E. Hagersten. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. 20–27.
- Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223 – 250.
- U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*.
- T. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization* (2014).
- S. Carr, S. McKinley, and C. Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*. 252–262.
- C. Cascaval and A. Padua. 2003. Estimating cache misses and locality using stack distances. In *17th Annual International Conference on Supercomputing (ICS'03)*. 150–159.
- S. Chatterjee, E. Parker, J. Hanlon, and R. Lebeck. 2001. Exact Analysis of the Cache Behavior of Nested Loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. 286–297.
- J. Edler and M. Hill. 1999. Dinero IV Trace-Driven Uniprocessor Cache Simulator. <http://pages.cs.wisc.edu/~markhill/DineroIV>
- C. Fang, S. Can, S. Onder, and Z. Wang. 2005. Instruction based memory distance analysis and its application to optimization. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 27–37.
- C. Fang, S. Carr, S. Onder, and Z. Wang. 2004. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proc. 2004 Workshop on Memory System Performance*. 60–68.
- P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* (1992), 389–420.
- J. Ferrante, V. Sarkar, and W. Thrash. 1991. On estimating and enhancing cache effectiveness. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'91)*. 328–343.
- B. Fraguera, R. Doallo, and L. Zapata. 1999. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*. 221–231.
- B. Fraguera, R. Doallo, and L. Zapata. 2003. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Trans. Comput.* (2003), 321–336.
- A. Frumkin and Rob F. Van W. 2002. Tight bounds on cache use for stencil operations on rectangular grids. *J. ACM* (2002), 434–453.
- S. Ghosh, M. Martonosi, and S. Malik. 1998. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*. 228–239.
- S. Ghosh, M. Martonosi, and S. Malik. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems* (1999), 703–746.
- S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parelo, M. Sigler, and O. Temam. 2006. Semi-Automatic Composition of Loop Transformations. *International Journal of Parallel Programming* (2006), 261–317.
- S. Harper, J. Kerbyson, and R. Nudd. 1999. Analytical modeling of set-associative cache behavior. *IEEE Trans. Comput.* (1999), 1009–1024.
- D. Hill and J. Smith. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput.* (1989), 1612–1630.

- C. Hong, W. Bao, A. Cohen, S. Krishnamoorthy, L. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. 2016. Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (2016), 129–144.
- W. Kelly and W. Pugh. 1993. *A Framework for Unifying Reordering Transformations*. Technical Report.
- M. Kong, R. Veras, K. Stock, F. Franchetti, L. Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 127–138.
- W. Lim and S. Lam. 1997. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. 201–214.
- C. Oppen. 1978. A $2^{2^{p^n}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. System Sci.* (1978), 323–332.
- L. Pouchet. 2017a. PoCC, the Polyhedral Compiler Collection 1.4. <http://pocc.sourceforge.net>
- L. Pouchet. 2017b. PolyBench/C 4.0. <http://polybench.sourceforge.net>
- H. Ramaprasad and F. Mueller. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*. 148–157.
- G. Rivera and C. Tseng. 1998. Data transformations for eliminating conflict misses. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI'98)*. 38–49.
- V. Sarkar and N. Megiddo. 2000. An analytical model for loop tiling and its solution. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*. IEEE, 146–153.
- J. Shirako, K. Sharma, N. Fauzia, L. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. 2012. Analytical bounds for optimal tile size selection. In *International Conference on Compiler Construction (CC'12)*. Springer, 101–121.
- A. Shrivastava, J. Lee, and R. Jeyapaul. 2010. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'10)*. 143–152.
- P. Singh, S. Stone, and F. Thiebaut. 1992. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Trans. Comput.* (1992), 811–825.
- M. Valiev, J. Bylaska, N. Govind, K. Kowalski, Tjerk P. Straatsma, Hubertus J J. Van D., D. Wang, J. Nieplocha, E. Apra, L. Windus, et al. 2010. NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* (2010), 1477–1489.
- X. Vera, J. Abella, A. González, and J. Llosa. 2003. Optimizing program locality through CMEs and GAs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. 68–78.
- X. Vera, J. Abella, J. Llosa, and A. González. 2005. An Accurate Cost Model for Guiding Data Locality Transformations. *ACM Transactions on Programming Languages and Systems* (2005), 946–987.
- X. Vera, N. Bermudo, J. Llosa, and A. González. 2004. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems* (2004), 263–300.
- X. Vera and J. Xue. 2002. Let's study whole-program cache behaviour analytically. In *International Symposium on High-Performance Computer Architecture (HPCA'02)*. 175–186.
- S. Verdoolaege. 2007. Barvinok, a library for counting the integer points in parametric and non-parametric polytopes. <http://barvinok.gforge.inria.fr>
- S. Verdoolaege. 2010a. ISL: An integer set library for the polyhedral model. In *the 3rd International Congress on Mathematical Software*.
- S. Verdoolaege. 2010b. ISL, the Integer Set Library. <http://repo.or.cz/w/isl.git>
- S. Verdoolaege and T. Grosser. 2012. Polyhedral extraction tool. In *2nd International Workshop on Polyhedral Compilation Techniques*.
- S. Verdoolaege, R. Seghir, K. Beyls, V. Lochner, and M. Bruynooghe. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* (2007), 37–66.
- W. Wang and L. Baer. 1990. Efficient Trace-driven Simulation Method for Cache Performance Analysis. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*. 27–36.
- J. Xue and X. Vera. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput.* (2004), 547–566.
- W. Zhang. 2005. Computing cache vulnerability to transient errors and its implication. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*. 427–435.