A Fast Analytical Model of Fully Associative Caches

Tobias Gysi
ETH Zurich
Switzerland
tobias.gysi@inf.ethz.ch

Tobias Grosser
ETH Zurich
Switzerland
tobias.grosser@inf.ethz.ch

Laurin Brandner
ETH Zurich
Switzerland
laurinb@student.ethz.ch

Torsten Hoefler
ETH Zurich
Switzerland
htor@inf.ethz.ch

Abstract

While the cost of computation is an easy to understand local property, the cost of data movement on cached architectures depends on global state, does not compose, and is hard to predict. As a result, programmers often fail to consider the cost of data movement. Existing cache models and simulators provide the missing information but are computationally expensive. We present a ***lightweight cache model*** for fully associative caches with least recently used (LRU) replacement policy that gives fast and accurate results. We count the cache misses without explicit enumeration of all memory accesses by using ***symbolic counting*** techniques twice: 1) to derive the stack distance for each memory access and 2) to count the memory accesses with stack distance larger than the cache size. While this technique seems infeasible in theory, due to non-linearities after the first round of counting, we show that the counting problems are sufficiently linear in practice. Our cache model often computes the results within seconds and contrary to simulation the execution time is mostly problem size independent. Our evaluation measures modeling ***errors below 0.6%*** on real hardware. By providing accurate data placement information we enable memory hierarchy aware software development.

CCS Concepts • **Software and its engineering** → *Software performance*; *Compilers*.

Keywords static analysis, cache model, performance tool

ACM Reference Format:
Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A Fast Analytical Model of Fully Associative Caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314606

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314606

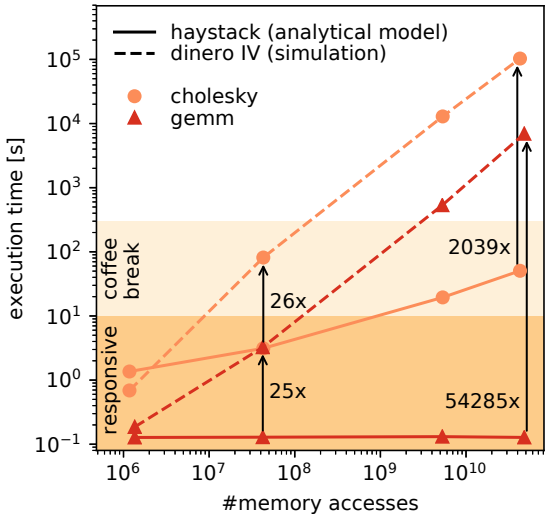


Figure 1. Scaling of the cache model compared to simulation.

1 Introduction

Most programmers know the time complexity of their algorithms and tune codes by minimizing computation. Yet, ever increasing data-movement costs urge them to pay more attention to data-locality as a prerequisite for peak performance. When considering different implementation variants of an algorithm, we typically have a good understanding of which variant performs less computation or can be vectorized well. Selecting the optimal tile size or deciding which loop fusion choice is optimal is far less intuitive. Essentially, we lack a perception of the cache state that allows us to reason about data movement.

Data-locality optimizations are often pushed to the end of the development cycle when the code is available for benchmarking. But at this stage eliminating fundamental design flaws may be hard. We believe a cache model responsive enough to be part of the day-to-day workflow of a performance engineer can provide the necessary guidance to make good design choices upfront. After the completion of the development, the very same model could provide the necessary data for accurate model driven automatic memory tuning.

We present HayStack[1] the first cache model for fully associative caches with least recently used (LRU) replacement policy which is both fast and accurate. At the core of our model, we calculate the LRU stack distance [29] (also called

[1] https://github.com/spcl/haystack

reuse distance [5, 15, 43]) symbolically for each memory access. The stack distance counts the distinct memory accesses between two subsequent accesses of the same memory location. All memory accesses with distance shorter than the cache size hit a fully associative LRU cache.

We show in Figure 1 the scaling of HayStack compared to the Dinero IV [17] cache simulator for increasing problem sizes. The simulation times are proportional to the problem size since simulators [7, 10, 17, 25] enumerate all memory accesses. We use the Barvinok algorithm [40] to count the cache misses. The algorithm avoids explicit enumeration by deriving symbolic expressions that evaluate to the cardinality of the counted affine integer sets and maps. As demonstrated by the flat GEMM scaling curve, this symbolic counting makes the model execution time problem size independent. Even for Cholesky factorization, with its known non-linearities [6] that prevent full symbolic counting, the scaling of the execution time remains flat compared to simulation.

While computing stack distances for static control programs is a well known technique, reducing stack distance information for all dynamic memory accesses to a single cache miss count is difficult. Beyls et al. [6] show that stack distances in general are non-affine. The divisions introduced when modeling cache lines add even more non-affine constraints. While symbolic summation over affine constraint sets is possible with the Barvinok algorithm, symbolic counting over non-affine constraints is considered hard in general.

In this work, we show that this generally hard problem can in practice become surprisingly tractable if non-linearities are carefully eliminated by either specialization or partial enumeration. As a result we contribute:

- The first efficient cache model to accurately predict static affine programs on fully associative LRU caches.
- An efficient hybrid algorithm that combines symbolic counting with partial enumeration to reduce the asymptotic cost of the cache miss counting.
- A set of simplification techniques that exploit the regular patterns induced by the cache line structure to make the stack distance polynomials affine.
- An exhaustive evaluation which shows that our cache model performs well in practice with large speedups compared to existing approaches while achieving high accuracy compared to measurements on real hardware.

2 Background

We first introduce our hardware model, provide background on cache misses, explain the concept of affine integer sets and maps, and discuss the set of considered programs.

2.1 Hardware Model

A cache implements various complex and sometimes undisclosed policies that define the exact behavior. We deliberately model a generic cache with full associativity and LRU replacement policy. When writing, we assume the caches allocate a cache line and load the memory reference if necessary (write-allocate) and then forward the write to all higher-level caches (write-through). We parametrize our cache model with the cache line size L and the cache size C in bytes. When modeling multiple cache hierarchy levels, we assume inclusive caches and specify the cache size for every hierarchy level. These design choices avoid an overly detailed model that is only correct in a very controlled environment with know data alignment and allocation. As shown by Section 4.2, we still model enough detail to produce actionable and accurate results in practice.

2.2 Cache Misses

We assume that the modeled programs run in isolation and that their execution starts with an empty cache. We count data accesses and ignore instruction fetches.

According to Hill [23], we distinguish three types of cache misses; 1) *compulsory misses* happen if a program accesses a cache line for the first time, 2) *capacity misses* happen if a program accesses too many distinct cache lines before accessing a cache line again, and 3) *conflict misses* happen if a program accesses to many distinct cache lines that map to the same cache set of an associative cache before accessing a cache line again. We model fully associative caches and thus compute only compulsory and capacity misses.

Not every access of a program variable translates in a cache access as the compiler may place scalar variables in registers. Compiler and hardware techniques such as out-of-order execution also change the order of the memory accesses. We assume all scalar variables are buffered in registers and count only array accesses in the order provided by the compiler front end.

The cache misses measured when profiling a program depend on many factors generally unknown to an analytical cache model, for example, concurrent programs or the operating system may pollute the caches or the hardware prefetchers may load more data than necessary. We do not consider this system noise and instead provide an approximate but deterministic cache model.

2.3 Integer Sets and Maps

We use sets and maps of integer tuples to count the cache misses. We next define the relevant set and map operations necessary for the model implementation. These operations are a subset of the functionality provided by the integer set library (isl) [38].

An affine set

$$\mathbf{S} = \{(i_0, \ldots, i_n) : con(i_0, \ldots, i_n)\}$$

defines the subset of integer tuples $(i_0, \ldots, i_n) \in \mathbb{Z}^n$ that satisfy the constraints $con(i_0, \ldots, i_n)$. The constraints are Presburger formulas that combine affine expressions with

```
1        int sum = 0;
2        for(int i=0; i<4; ++i)
3  S0:    M[i] = i;
4        for(int j=0; j<4; ++j)
5  S1:    sum += M[3-j];
```

Figure 2. Example program used for illustration.

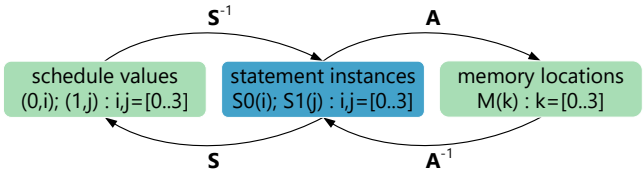


Figure 3. The statement instances and the related schedule values (schedule **S**) and memory accesses (access map **A**) are sufficient to compute the cache misses of a program.

comparison operators, boolean operators, and existential quantifiers. Presburger arithmetic [21] also admits floor division and modulo with a constant divisor.

An affine map

$$\mathbf{R} = \{(i_0, \ldots, i_n) \to (j_0, \ldots, j_m) : con(i_0, \ldots, i_n, j_0, \ldots, j_m)\}$$

defines the relation from integer tuples $(i_0, \ldots, i_n) \in \mathbb{Z}^n$ to integer tuples $(j_0, \ldots, j_m) \in \mathbb{Z}^m$ that satisfy the constraints $con(i_0, \ldots, i_n, j_0, \ldots, j_m)$ where the constraints have the same restrictions as the set constraints. The domain $\mathbf{R}_{dom}$ defines the set of the integer tuples $(i_0, \ldots, i_n)$ of the input dimensions for which a relation exists, and conversely the range $\mathbf{R}_{ran}$ defines the set of integer tuples $(j_0, \ldots, j_m)$ of the output dimensions for which a relation exists.

Both sets and maps support the set operations intersection $\mathbf{S}_1 \cap \mathbf{S}_2$, union $\mathbf{S}_1 \cup \mathbf{S}_2$, projection, and cardinality $|\mathbf{S}|$. The domain intersection $\mathbf{R} \cap_{dom} \mathbf{S}$ intersects the domain of the map **R** with the set **S**. Maps also support the map operations composition $\mathbf{R}_2 \circ \mathbf{R}_1$ and inversion $\mathbf{R}^{-1}$. The operator

$$\begin{aligned}\text{lexmin}(\mathbf{R}) = \{(i_0, \ldots, i_n) \to (m_0, \ldots, m_m) : \\ \nexists (i_0, \ldots, i_n) \to (j_0, \ldots, j_m) \in \mathbf{R}, \\ \text{s.t. } (j_0, \ldots, j_m) \prec (m_0, \ldots, m_m)\}\end{aligned}$$

computes for every input tuple $(i_0, \ldots, i_n)$ the lexicographic smallest output tuple $(m_0, \ldots, m_m)$ of all tuples $(j_0, \ldots, j_m)$ related to the input tuple.

A named set or map prefixes the integer tuples with names that convey semantic information. For example, we prefix the array element M(2) with the array name and the statement instance S0(1) with statement name. We use statement names starting with the letter S and array names starting with any other letter. The names are semantically equivalent to an additional tuple dimension.

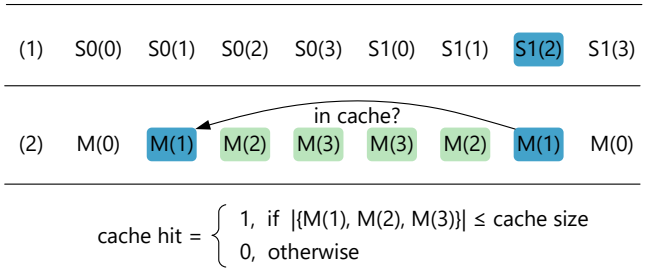


Figure 4. The (1) statement instance and the (2) memory access trace of the example program allow us to compute if the access M(1) of the statement S1(2) hits the cache.

2.4 Static Control Programs

Our cache model analyzes affine *static control programs* consisting of loop nests with known loop bounds that perform array accesses with affine index expressions. Figure 2 shows an example program with two statements: the statement S0 initializes an array M and the statement S1 accumulates the array elements. Before analyzing a program, we extract the sets and maps that specify the statement execution order and the memory access offsets.

The iteration domain

$$\mathbf{I} = \{\texttt{S0}(i) : 0 \le i < 4; \texttt{S1}(j) : 0 \le j < 4\}$$

defines the set of all executed statement instances. For the two statements of the example program, the loop variables i and j are limited to the range zero to three. To define the execution order, the schedule

$$\mathbf{S} = \{\texttt{S0}(i) \to (0, i); \texttt{S1}(j) \to (1, j)\} \cap_{dom} \mathbf{I}$$

maps the statement instances to a multi-dimensional schedule value. The statement instances then execute according to the lexicographic order of the schedule values. The intersection with the iteration domain **I** limits the schedule domain to the program loop bounds. The access map

$$\mathbf{A} = \{\texttt{S0}(i) \to \texttt{M}(i); \texttt{S1}(j) \to \texttt{M}(3 - j)\}$$

maps the array accesses of the statement instances to the accessed array elements. The iteration domain **I**, the schedule **S**, and the access map **A** capture all relevant program properties necessary to evaluate the cache model. Figure 3 shows how the schedule **S** and the access map **A** relate statement instances, schedule values, and memory locations.

3 Cache Model

Our cache model computes for every memory access the stack distance parametric in the loop variables and counts the instances with a stack distance larger than the cache capacity to determine the capacity misses. All memory accesses with undefined backward stack distance access the cache line for the first time and count as compulsory misses.

Figure 4 shows the computation of the capacity misses for the example program introduced by Figure 2: (1) enumerates the statement instances according to the schedule **S** and (2) applies the access map **A** to the statement instances to compute the memory trace. Assuming the array element size is equal to the cache line size, the stack distance corresponds to the cardinality of the set $\{\mathtt{M}(1), \mathtt{M}(2), \mathtt{M}(3)\}$ which contains the array elements accessed between and including the two subsequent accesses of M(1). The second access of M(1) hits the cache if the cardinality of the set is lower than or equal to the cache capacity.

3.1 Computing the Stack Distance

The stack distance computation counts the number of distinct memory accesses between subsequent accesses of the same memory location. We determine for every memory reference the last access to the same memory location and count the set of memory accesses since this last access to obtain the stack distance parametric in the loop variables.

For our example program, the stack distance of the memory access in statement S1 is equal to the loop variable j plus one. We can thus express the stack distance of the memory access with the map

$$\mathbf{D} = \{\mathtt{S1}(j) \longrightarrow j + 1 : 0 \leq j < 4\}$$

limited to the statement iteration domain. As the statement S0 accesses all array elements for the first time its backward stack distance is undefined and the accesses count as compulsory misses.

Our discussion of the stack distance computation initially assumes that every statement performs at most one access of a one-dimensional array with an element size equal to the cache line size. At the end of this section, we show how to overcome these limitations.

The memory accesses execute according to the statement execution order defined by the schedule. The map

$$\begin{aligned}\mathbf{L}_{\prec} = \{(i_0, \ldots, i_n) \longrightarrow (j_0, \ldots, j_n) : \\ (i_0, \ldots, i_n) \prec (j_0, \ldots, j_n) \wedge \\ (i_0, \ldots, i_n), (j_0, \ldots, j_n) \in \mathbf{S}_{ran}\}\end{aligned}$$

relates the schedule values $(i_0, \ldots, i_n)$ to all lexicographically larger schedule values $(j_0, \ldots, j_n)$ and the map

$$\begin{aligned}\mathbf{L}_{\leq} = \{(i_0, \ldots, i_n) \longrightarrow (j_0, \ldots, j_n) : \\ (i_0, \ldots, i_n) \leq (j_0, \ldots, j_n) \wedge \\ (i_0, \ldots, i_n), (j_0, \ldots, j_n) \in \mathbf{S}_{ran}\}\end{aligned}$$

relates the schedule values $(i_0, \ldots, i_n)$ to all lexicographically larger or equal schedule values $(j_0, \ldots, j_n)$. Later on, we use these helper maps to filter relations by execution order.

The stack distance computation first identifies all accesses to the same array element. The equal map

$$\mathbf{E} = \mathbf{S} \circ \mathbf{A}^{-1} \circ \mathbf{A} \circ \mathbf{S}^{-1}$$

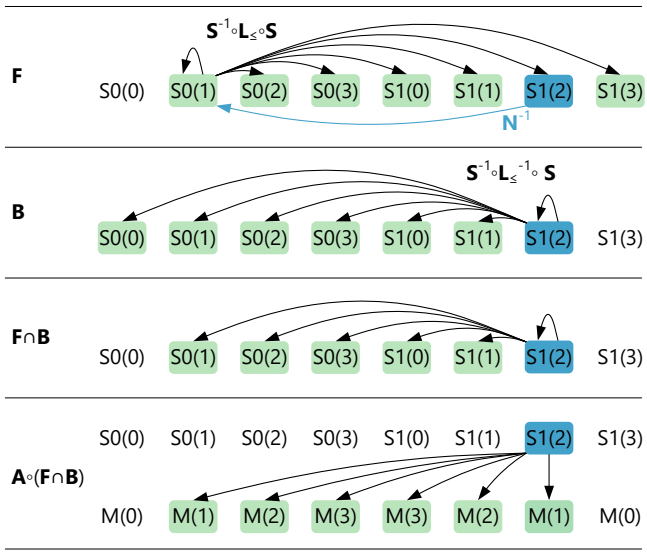


Figure 5. The relations of the forward map **F** and the backward map **B** for the statement instance S1(2) of the example program (the forward map **F** corresponds to the concatenation of the blue backward arrow and the black forward arrows). The map intersection defines the statement instance between and including the two accesses of M(1). The concatenation with the map **A** yields the related memory accesses.

relates each schedule value to all schedule values that access the same array element. The concatenation $\mathbf{A} \circ \mathbf{S}^{-1}$ maps the schedule values to the accessed array elements and its reverse $\mathbf{S} \circ \mathbf{A}^{-1}$ maps the accesses back to the schedule values. For our example program, the composition

$$\begin{aligned}\mathbf{A} \circ \mathbf{S}^{-1} = \{(0, i) \longrightarrow \mathtt{M}(i) : 0 \leq i < 4; \\ (1, j) \longrightarrow \mathtt{M}(3 - j) : 0 \leq j < 4\}\end{aligned}$$

relates the schedule values to the accesses of the array M. The equal map then relates all schedule values that access the same array element. For example, the relations $(0, i) \longrightarrow \mathtt{M}(i)$ and $(1, j) \longrightarrow \mathtt{M}(3 - j)$ access the same array element if i is equal to $3 - j$. The resulting equal map

$$\begin{aligned}\mathbf{E} = \{(0, i) \longrightarrow (0, i) : 0 \leq i < 4; \\ (1, j) \longrightarrow (1, j) : 0 \leq j < 4; \\ (0, i) \longrightarrow (1, j) : j = 3 - i \wedge 0 \leq i < 4; \\ (1, j) \longrightarrow (0, i) : i = 3 - j \wedge 0 \leq j < 4\}\end{aligned}$$

contains the relation $(0, i) \longrightarrow (1, j)$ with $j = 3 - i$ and its reverse but also the self relations of the schedule values.

The lexicographically shortest relations of the equal map denote the subsequent accesses to the same array element which are closest in time. The next map

$$\mathbf{N} = \mathbf{S}^{-1} \circ \text{lexmin}(\mathbf{L}_{\prec} \cap \mathbf{E}) \circ \mathbf{S}$$

intersects the equal map $\mathbf{E}$ with the map $\mathbf{L}_{\prec}$ to filter out all backward in time and self relations and the lexmin operator removes all forward in time relations except for the shortest ones. We compose the result with $\mathbf{S}$ and $\mathbf{S}^{-1}$ to convert the schedule values to statement instances. The next map consequently relates every statement instance to the next statement instance that accesses the same array element. For our example program, the equal map contains only the forward relation $(0, i) \rightarrow (1, j)$ which means the lexmin operator has no effect since there is only one relation per statement instance. The next map

$$\mathbf{N} = \{\mathtt{S0}(i) \rightarrow \mathtt{S1}(j) : j = 3 - i \wedge 0 \leq i < 4\}$$

thus relates the instances of statement S0 to the instances of statement S1 that access the same array element.

The next map contains subsequent statement instances that access the same array element but not the statement instances executed in between. To compute them, we intersect the set of statement instances executed after the first access with the set of statement instances executed before the second access of the same array element. Figure 5 illustrates this intersection. The backward map

$$\mathbf{B} = \mathbf{S}^{-1} \circ \mathbf{L}_{\preceq}^{-1} \circ \mathbf{S}$$

relates the statement instances to all statement instances with lexicographically smaller or equal schedule value. The maps $\mathbf{S}$ and $\mathbf{S}^{-1}$ convert from statement instances to schedule values and back. The forward map

$$\mathbf{F} = (\mathbf{S}^{-1} \circ \mathbf{L}_{\preceq} \circ \mathbf{S}) \circ \mathbf{N}^{-1}$$

relates the statement instances to all statement instances with lexicographically larger or equal schedule value than the statement instance that last accessed the same array element. We reverse the next map $\mathbf{N}$ to compute the statement instance that accessed the array element last. The intersection of the forward map and the backward map contains all statement instances executed between subsequent accesses of the same array element.

Figure 5 shows the forward and backward map relations for the statement instance S1(2) of the example program that accesses the array element M(1). The forward map $\mathbf{F}$ corresponds to the concatenation of the blue backward arrow and the black forward arrows. The intersection of the two maps contains the statement instances executed between the subsequent accesses of the array element M(1). We finally concatenate this intersection with the access map $\mathbf{A}$ to obtain the stack distance map that relates every statement instance to the array accesses performed since the last access of the same array element.

The number of related array elements defines the stack distance of the statement instances in the stack distance map. We use the isl [38] implementation of the Barvinok algorithm [40] to count the relations symbolically. The algorithm computes the map cardinality by counting the points of the range related to every point of the domain. The result of the computation are quasi polynomials parametric in the input dimensions of the map that evaluate to the number of related range points. As the domain is not always homogeneous, the algorithm splits the map domain into pieces that consist of a quasi polynomial and the subdomain of the map domain where the polynomial is valid. After counting the stack distance map, the distance set

$$\mathbf{D} = \{|\mathbf{A} \circ (\mathbf{F} \cap \mathbf{B})|\}$$

contains pieces with quasi polynomials parametric in the schedule input dimensions that for a subdomain of the iteration domain evaluate to the stack distance. The pieces do not overlap and together cover the full iteration domain. For our example program, the distance set

$$\mathbf{D} = \{\mathtt{S1}(j) \rightarrow j + 1 : 0 \leq j < 4\}$$

contains one piece with the polynomial $\mathtt{S1}(j) \rightarrow j + 1$ and the domain $0 \leq j < 4$ covering the entire iteration domain.

cache lines and multi-dimensional arrays An adapted access map $\mathbf{A}$ that relates statement instances to cache lines instead of array elements suffices to support cache lines and multi-dimensional arrays. Let us assume our example program initializes the diagonal elements of a two-dimensional array $\mathtt{M}(i, i)$. Then the access map

$$\mathbf{A} = \{\mathtt{S0}(i) \rightarrow \mathtt{M}(i, c = \lfloor i * E/L \rfloor)\}$$

models the accessed cache lines given the size of the array elements E and cache line size L in bytes. We replace the innermost dimension of the array access with the cache line index c, which multiplies the array index with the element size and divides the result by the cache line size. As a result, accesses of neighboring array elements map to the same cache line. The outer dimensions of the array index remain unchanged since we assume the innermost dimension is cache line aligned and padded to an integer multiple of the cache line size. This restriction can be lifted at the expense of a more complex formulation.

multiple memory accesses per statement An extension of the schedule $\mathbf{S}$ and the access map $\mathbf{A}$ with an additional schedule dimension that orders the memory accesses of the statements allows us to model more than one memory access per statement. Let us assume the statement S0 of the example program reads the array element $\mathtt{I}(i)$ and writes the result to the array element $\mathtt{M}(i)$. We then extend the schedule

$$\mathbf{S} = \{\mathtt{S0}(i, a) \rightarrow (0, i, a); \mathtt{S1}(j, a) \rightarrow (1, j, a)\}$$

with the access dimension a that orders the memory accesses of the statement. Then the access map

$$\mathbf{A} = \{\mathtt{S0}(i, 0) \rightarrow \mathtt{I}(i); \mathtt{S0}(i, 1) \rightarrow \mathtt{M}(i); \mathtt{S1}(j, 0) \rightarrow \mathtt{M}(3 - j)\}$$

assigns every array access to a unique statement instance since the access dimension enumerates the array accesses of every statement in the order provided by the compiler

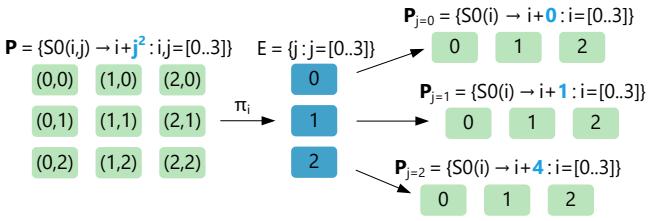


Figure 6. To count the non-affine piece **P**, we project out the affine i-dimension to obtain the enumeration domain **E**. We next bind the j-dimension of the piece **P** to the j-values in the enumeration domain and separately count the cache misses for the resulting affine pieces $\mathbf{P}_{j=0}$, $\mathbf{P}_{j=1}$, and $\mathbf{P}_{j=2}$.

front end. The extended schedule executes only one array access per statement instance and thus requires no further modifications of the stack distance computation.

The output of the stack distance computation is a set of polynomials that defines the backward stack distance for every array access of the static control program.

3.2 Counting the Capacity Misses

All memory accesses with stack distance larger then the cache size count as capacity miss. As discussed in Section 3.1, the stack distance computation splits the iteration domain into pieces. Each piece defines the stack distance for a subdomain of the iteration domain. To obtain the capacity misses, we count for every piece the points of the subdomain for which the polynomial evaluates to a stack distance larger than the cache size.

The piece with polynomial $\mathtt{S1}(j) \longrightarrow j + 1$ and domain $0 \leq j < 4$ defines the stack distance for the entire iteration domain of our example program. The cache miss set

$$\mathbf{M} = \{\mathtt{S1}(j) : j + 1 > C \wedge 0 \leq j < 4\}$$

contains all points of the piece with stack distance larger than cache size C which means the cardinality of the cache miss set $|\mathbf{M}|$ is equal to the number of capacity misses. Assuming cache size two, the cache miss set contains the statement instances S1(2) and S1(3) that cause two capacity misses.

The distance set specifies the stack distance for all program statements. To count the capacity misses per statement, we split the distance set by statement and compute the cache misses separately. Without loss of generality, we discuss the cache miss computation for a statement S0.

The Barvinok algorithm also computes the set cardinality by counting the points symbolically. We use the algorithm to count affine cache miss sets and resort to explicit enumeration for non-affine sets. As explicit enumeration is expensive, we only enumerate the non-affine polynomial dimensions and count the affine dimensions symbolically. This *partial enumeration* technique splits cache miss sets into pieces with affine lower-dimensional polynomials. Figure 6 demonstrates the technique for an example polynomial with non-affine j-dimension. Section 3.3 discusses further techniques to split non-affine pieces into multiple affine pieces.

Algorithm 1: counting the capacity misses

input : **D** distance set of pieces
output : T total number of cache misses
parameter : C cache size

```
1  T ← 0
2  foreach P in D do
3      if isPieceAffine(P) then
4          T ← T + countAffinePiece(P, C)
5      else
6          E ←getNonAffineDomain(P)
7          foreach pt in E do
8              P_pt ←bindNonAffineDimensions(P, pt)
9              T ← T + countAffinePiece(P_pt, C)
10         end
11     end
12 end
13 return T
```

Algorithm 1 counts the total number of cache misses T given the distance set **D** of the program. The algorithm enumerates all pieces **P** of the distance set (lines 2-12). Every piece **P** consists of a polynomial and a domain that define the stack distance of a memory access for a subdomain of the iteration domain. If the polynomial of the piece **P** is affine we count the cache misses symbolically (lines 3-4), otherwise the *partial enumeration* projects the non-affine dimensions out of the domain of the piece **P** and enumerates all points of the resulting non-affine enumeration domain **E** (lines 6-9). For every such point pt, we bind the non-affine dimensions of the piece **P** to the coordinates of the point pt and count the cache misses of the affine piece $\mathbf{P}_{\text{pt}}$ symbolically. Figure 6 illustrates the splitting of non-affine pieces (lines 6-9).

The method *countAffinePiece* counts the cache misses of the piece **P** with affine stack distance polynomial. A polynomial is affine if its degree is zero or one. We first compute the cache miss set

$$\mathbf{M} = \{\mathtt{S0}(i_0, \ldots, i_n) : \mathbf{P}_p(i_0, \ldots, i_n) > C \wedge (i_0, \ldots, i_n) \in \mathbf{P}_D\}$$

where $\mathbf{P}_p$ denotes the polynomial and $\mathbf{P}_D$ the domain of the piece **P**. The cache miss set contains all memory accesses with stack distance larger than cache size C. To count the cache misses, we compute the cardinality $|\mathbf{M}|$ using the Barvinok algorithm.

The method *getNonAffineDomain* projects all points of the piece **P** to the non-affine dimensions to obtain the enumeration domain **E**. For example, Figure 6 projects the piece

$$\mathbf{P} = \{\mathtt{S0}(i, j) \longrightarrow i + j^2 : 0 \leq i < 3 \wedge 0 \leq j < 3\}$$

which contains the quadratic term j^2. We project the points to the non-affine j-dimension to compute the enumeration

domain $\mathbf{E} = \{j : 0 \le j < 3\}$. The enumeration always spans all dimensions with degree larger than one. But the polynomial may also contain product terms with multiple dimensions. We then greedily select the dimensions that conflict with most other dimensions. For example, if the polynomial contains the products ij and ik we enumerate the i-dimension since it conflicts with both other dimensions.

The method *bindNonAffineDimensions* binds the non-affine dimensions of the piece $\mathbf{P}$ to the values of the point pt. For example, Figure 6 binds the j-dimension of the piece

$$\mathbf{P} = \{\mathtt{S0}(i,j) \longrightarrow i + j^2 : 0 \le i < 3 \wedge 0 \le j < 3\}$$

to the value two and obtains the piece

$$\mathbf{P}_{j=2} = \{\mathtt{S0}(i) \longrightarrow i + 4 : 0 \le i < 3\}$$

which we can count with the method *countAffinePiece*.

The counting algorithm works for all static control programs and avoids complete enumeration except all dimensions are non-affine.

3.3 Eliminating Non-Affine Terms

Many stack distance polynomials contain non-affine terms that prevent fast symbolic counting. We develop rewrite strategies that eliminate non-affine terms containing floor expressions. The floor expressions themselves are quasi-affine but often appear in products with other non-constant operands modeling effects such as the stack distance variation for different cache line offsets. We specialize the stack distance polynomials for different cache line offsets to make them affine which enables the efficient symbolic counting.

The floor expressions of some polynomials differ only by a constant offset. For example, the piece

$$\mathbf{P} = \{\mathtt{S0}(i,j) \longleftarrow (\lfloor (1+i)/3 \rfloor - \lfloor i/3 \rfloor) j : 0 \le i < 3 \wedge 0 \le j < 2\}$$

contains the floor expressions $\lfloor (1+i)/3 \rfloor$ and $\lfloor (i)/3 \rfloor$. The two floor expressions are equal except if i modulo three is equal to two. Then the second floor expression is larger by one. The difference of the two floor expressions thus evaluates to zero for the first two elements and to one for the last element of every cache line. Figure 7 shows how to introduce simplified polynomials for the first two and the last element of every cache line. This *equalization* technique splits the cache line in multiple regions that typically contain more than one element.

The polynomials may also contain terms with the plain variable and other terms which compute the floor of the variable. For example, the piece

$$\mathbf{P} = \{S0(i,j) \longrightarrow (i - 3\lfloor i/3 \rfloor) j : 0 \le i < 3 \wedge 0 \le j < 2\}$$

contains the floor expression $3\lfloor i/3 \rfloor$ which is equal to i except for a constant that depends on the cache line offset. Figure 8 shows how to replace the polynomial with one simplified polynomial per cache line offset. This *rasterization* technique enumerates all cache line offsets.

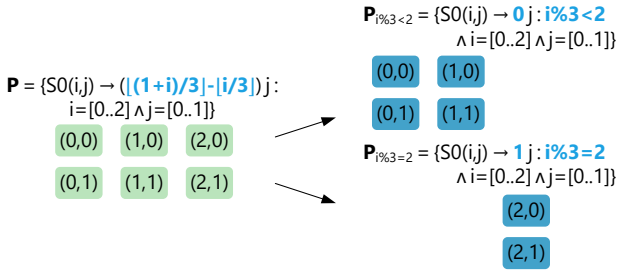


Figure 7. *Equalization* replaces the non-affine piece $\mathbf{P}$ with the affine pieces $\mathbf{P}_{i\%3<2}$ and $\mathbf{P}_{i\%3=2}$ to model a stack distance that varies at the last cache line offset.

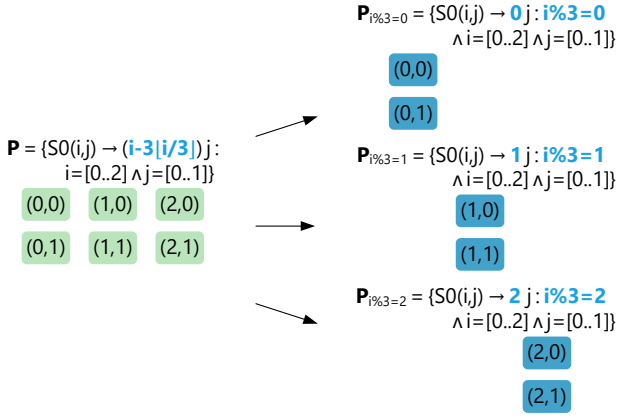


Figure 8. *Rasterization* replaces the non-affine piece $\mathbf{P}$ with the affine pieces $\mathbf{P}_{i\%3=0}$, $\mathbf{P}_{i\%3=1}$, and $\mathbf{P}_{i\%3=2}$ to model a stack distance that varies at every cache line offset.

We apply the two floor elimination techniques in the order of presentation and only keep the results if the degree of at least one simplified polynomial is lower than the degree of the original polynomial.

3.4 Counting the Compulsory Misses

All memory accesses that touch a cache line for the first time are compulsory misses.

As the array `M` of our example program is initialized by the statement `S0`, the first map

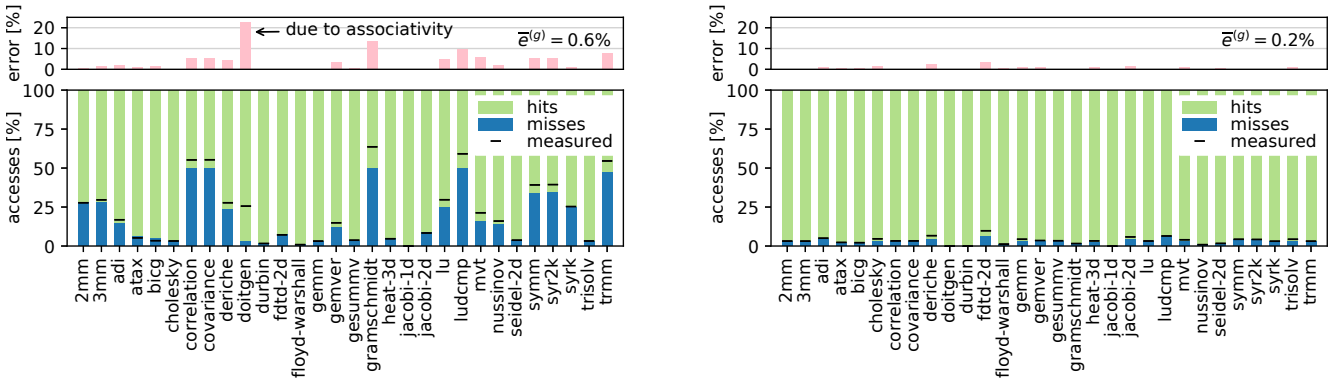
$$\mathbf{F} = \{\mathtt{M}(i) \longrightarrow \mathtt{S0}(i) : 0 \le i < 4\}$$

relates every array element to the statement instance that accesses the element first which means the cardinality $|\mathbf{F}_{dom}|$ of the first map domain counts the compulsory misses.

The compulsory misses are the memory accesses with lexicographically minimal schedule value. The first map

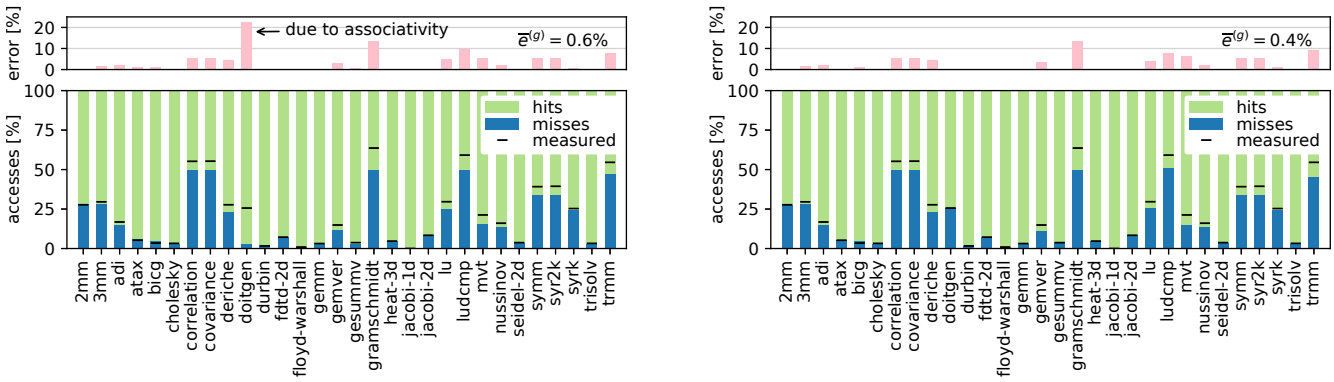
$$\mathbf{F} = \mathbf{S}^{-1} \circ \text{lexmin}(\mathbf{S} \circ \mathbf{A}^{-1})$$

thus selects for every memory access the lexicographically minimal relation of the composition $\mathbf{S} \circ \mathbf{A}^{-1}$ that relates memory accesses to schedule values and composes the result with the inverse schedule $\mathbf{S}^{-1}$ to obtain the related statement



(a) L1 cache **(b)** L2 cache

Figure 9. Cache misses and hits predicted by HayStack compared to the measured cache misses (median of 10 measurements) for the PolyBench kernels with the prediction error relative to the number of memory accesses on top.



(a) L1 cache (fully associative) **(b)** L1 cache (8-way associative)

Figure 10. Cache misses and hits simulated by Dinero IV compared to the measured cache misses (median of 10 measurements) for the PolyBench kernels with the prediction error relative to the number of memory accesses on top.

instances. The composition with the inverse schedule allows us to intersect the range of the first map with the iteration domain of the individual statements to count the compulsory misses per statement. For our example program, the composition

$$\mathbf{S} \circ \mathbf{A}^{-1} = \{\mathtt{M}(i) \longrightarrow (0, i) : 0 \leq i < 4; \\ \mathtt{M}(j) \longrightarrow (1, 3 - j) : 0 \leq j < 4\}$$

contains two accesses for every array element. The lexmin operator removes the second access due to the lexicographically larger schedule value. After the composition with the inverse schedule $\mathbf{S}^{-1}$, we use the Barvinok algorithm to count the compulsory misses $|\mathbf{F}_{dom}|$.

3.5 Computational Complexity

All compute-heavy parts of our cache model perform Presburger arithmetic that in general is known to have very high computational complexity [21, 30]. The established complexity bounds range from polynomial time decidable [26] for expressions with fixed dimensionality and only existential quantification to double exponential [19] for arbitrary expressions. Haase [21] presents further results that show a complexity increase with the dimensionality and the number of quantifier alternations of the Presburger expression.

The Presburger relations computed by our cache model have only existential quantification and the dimensionality is limited by the loop depth suggesting polynomial complexity. Yet, the cache model may introduce further variables to model divisions or modulo operations making the complexity exponential in the number of dimensions.

Although the cache model has exponential worst-case complexity, the empirical performance evaluation presented in Section 4.3 shows that our cache model performs well for typical input programs. The dimensionality of the observed Presburger relations remains limited since most real-world programs do not make extensive use of branch conditions and index expressions that result in integer divisions or modulo operations.

4 Evaluation

We next evaluate the performance of HayStack and compare its accuracy to simulated and measured results.

4.1 Setup and Methodology

We evaluate on a test system with two 18-core Intel Xeon Gold 6150 processors. Every core has a 32KiB L1 cache (8-way set associative) and an inclusive 1MiB L2 cache (16-way set associative). The non-inclusive 18x1.375MiB L3 cache (11-way set associative) is shared among all cores. A non-inclusive cache may and an inclusive cache has to duplicate all cache lines stored by the lower-level caches. All caches load the cache line before writing (write-allocate) and forward the write only if the cache line is evicted (write-back).

We compile with GCC 6.3 and use the Dinero IV cache simulator [17] to compute and the PAPI-C library [34] to measure the number of cache misses. We evaluate the model for a number of different kernels. PolyBench 4.2.1-beta [32] is a collection of static control programs that implement algorithmic motifs from scientific computing. If not stated otherwise the PolyBench experiments use the default configuration (large) and the model emulates fully associative L1 and L2 caches with the capacities of the test system.

All performance measurements run single-threaded using only one core of the test system. To quantify measurement noise, the execution times show the median and the non-parametric 95% confidence intervals [24] of 10 measurements.

4.2 Accuracy Overview

All mathematical models are a trade-off between accuracy and complexity. A static cache model cannot predict dynamic measurement noise for example due to concurrent code execution. We aim at an accurate prediction of the cache misses without modeling too many implementation details.

A comparison to measurements on a real system is the main benchmark for every cache model. To measure the cache misses, we compile the PolyBench [32] kernels with PAPI [34] support using GCC optimization level O2. PolyBench [32] flushes the caches before every kernel execution which allows us to measure compulsory and capacity misses. We collect the counters `PAPI_L1_DCM` and `PAPI_L2_DCM` that sum the data cache misses for the L1 and L2 caches, respectively. Figure 9 compares the sum of the compulsory and capacity misses predicted by HayStack to the measured cache misses shown by black lines. Most kernels cause more cache misses than predicted which is expected since we model idealized fully associative caches with LRU instead of pseudo-LRU replacement policy. We also do not consider possible overfetch due to the hardware prefetchers. To quantify the error, Figure 9 shows for every kernel the prediction error relative to the total number of memory accesses computed by the model. Most kernels have low single digit prediction errors with a geometric mean error of 0.6% and 0.2% for the L1 cache and the L2 cache, respectively. Only doitgen and gramschmidt have prediction errors above 10%.

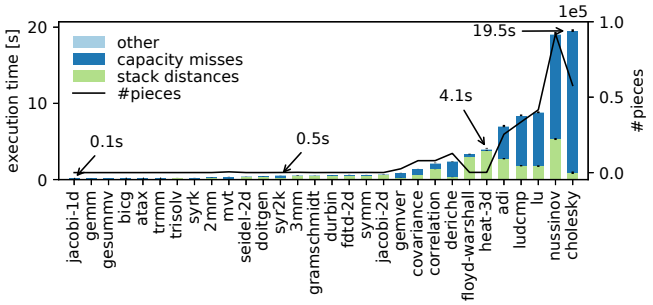


Figure 11. Execution times for the main components of HayStack compared to the number of separately counted pieces for the PolyBench kernels sorted by execution time.

We also execute the PolyBench kernels with Dinero IV [17] to simulate the number of cache misses with full associativity and with the associativity of our test system. Figure 10 compares the sum of the simulated compulsory, capacity, and conflict misses to the measured cache misses shown by black lines. We observe that the simulation results for the fully associative L1 cache qualitatively agree with the model. All simulation results are within 0.1% of the model for the L1 cache and within 3% of the model for the L2 cache (relative to the total number of memory accesses). We conclude that our design decisions of padding the innermost dimension of multi-dimensional arrays, discussed in Section 3.1, and modeling only array accesses and not scalar accesses, discussed in Section 2.2, have no significant impact on the accuracy of the model. The simulation results with test system associativity eliminate the error for the doitgen kernel. We conclude that modeling set associativity is only relevant for one of the PolyBench kernels. The error of the remaining kernels is dominated by other error sources such as the difference between LRU and pseudo-LRU replacement policy that are neither considered by the simulator nor by the model.

HayStack reproduces the simulation results for full associativity and the associativity mismatch compared to the test system does not dominate the modeling error.

4.3 Performance Overview

We next analyze the performance of HayStack and its sensitivity to model parameters such as the problem size or the number of cache hierarchy levels.

Two components dominate the model execution time: 1) the stack distance computation discussed in Section 3.1 and 2) the capacity miss counting discussed in Section 3.2. Figure 11 shows the cost of the two components compared to the total model execution times for the PolyBench kernels. The analysis of most kernels terminates within 5 seconds (jacobi-1d to heat-3d) while the more expensive kernels take up to

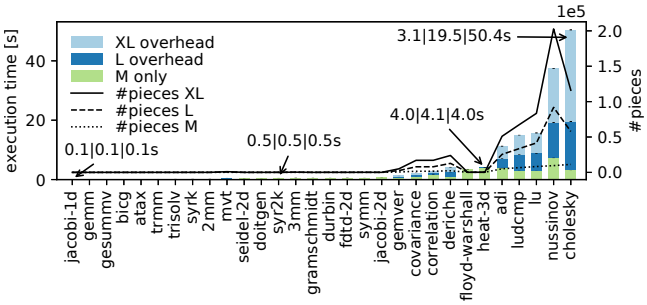


Figure 12. Execution times for the extra large (XL), large (L), and medium (M) problem sizes of PolyBench compared to the number of counted pieces.

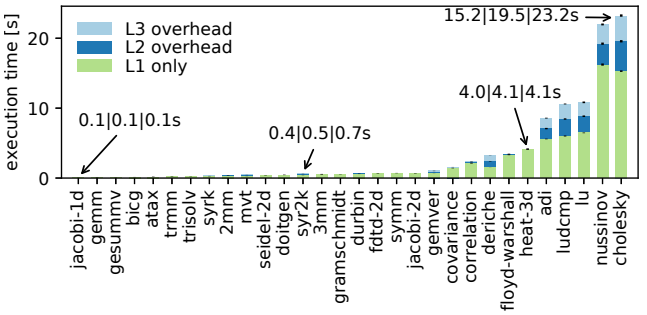


Figure 13. Comparison of the execution times when modeling one, two, or three cache hierarchy levels.

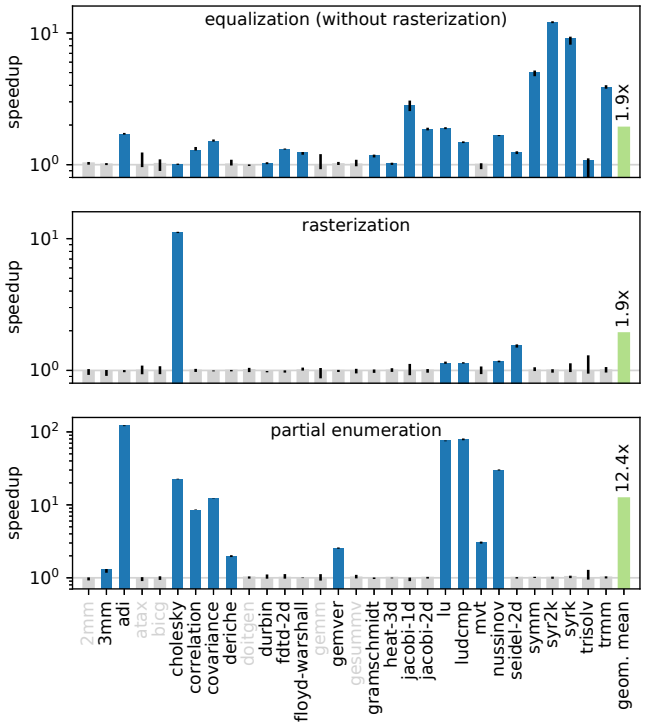


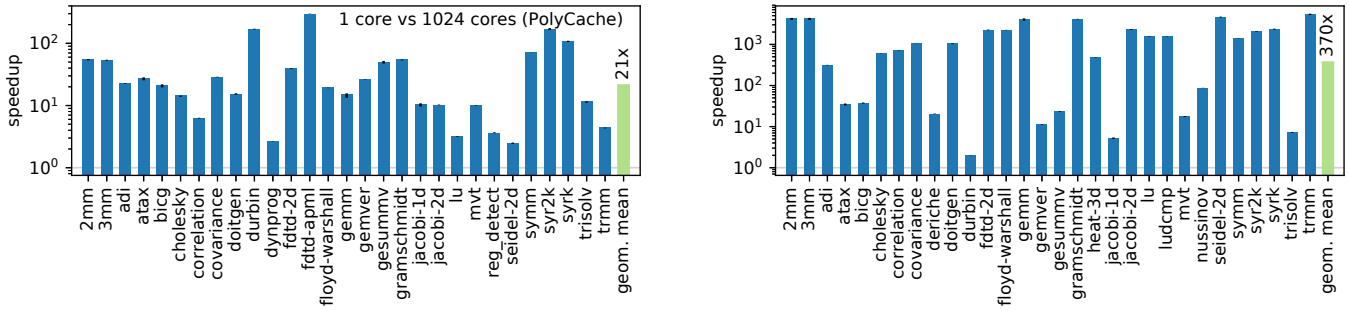
Figure 14. Speedup due to *equalization*, *rasterization*, and *partial enumeration*. All kernels without speedup (gray bars) are not included in the geometric mean. Only few kernels run fast without any optimization (gray labels).

20 seconds (adi to cholesky). The capacity miss counting dominates the cost of the expensive kernels. When counting the capacity misses, the *partial enumeration* and to a lesser extend the *equalization* and *rasterization*, discussed in Section 3.3, split the iteration domain into pieces with affine stack distance polynomials that support symbolic counting. The solid line in Figure 11 shows the number of counted pieces. We observe that the expensive kernels require more splits due to non-affine stack distance polynomials and that the counting costs correlate with the number of pieces.

Other than for a cache simulator, the model execution time is not proportional to the number of memory accesses. Figure 12 shows the model execution times for the three largest PolyBench problem sizes. The large (L) and the extra large (XL) problem size perform roughly 100 and 1000 times more memory access than the medium (M) problem size, respectively. Yet, the execution times remain constant for a majority of the kernels. Only the execution times of the expensive kernels increase since the *partial enumeration* requires more splits. The number of counted pieces, shown by the solid, dashed, and dotted lines in Figure 12, correlate with the cost increase for the larger problem sizes. Even for the expensive kernels, the increase of the execution time is not proportional to the number of memory accesses since we enumerate only the non-affine dimensions of the stack distance polynomials.

When counting the cache misses for multiple cache hierarchy levels, we reuse the stack distance polynomials and enumerate the non-affine dimensions only once. The counting of the individual pieces is the only step repeated for every cache size. As the Barvinok algorithm [40] supports parametric counting, we can count the capacity misses parametric in the cache size which avoids any additional overhead when modeling additional cache hierarchy levels. We benchmark the non-parametric version of the code as it runs faster even when modeling three cache hierarchy levels. Figure 13 shows minor increases of the total execution time for two and three cache hierarchy levels.

The *partial enumeration*, discussed in Section 3.2, combines enumeration of the non-affine dimensions with symbolic counting of the affine dimensions. Figure 14 compares *partial enumeration* to the explicit enumeration of all points. When considering only kernels with non-affine stack distance polynomials, we measure a geometric mean speedup of 12.4x with pieces that contain 4,400 points on average. The more points per piece the bigger the efficiency gain due to our hybrid counting approach. We still require explicit enumeration for all non-affine polynomials without affine dimension. Table 1 shows that most non-affine polynomials



(a) PolyCache **(b)** Dinero IV

Figure 15. Speedup of HayStack compared to PolyCache and Dinero for the PolyBench 3.2 and 4.2.1 kernels, respectively.

Table 1. Number of non-affine polynomials with zero, one, or two affine dimensions.

	3mm	adi	cholesky	correlation	covariance	deriche	gemver	lu	ludcmp	mvt	nussinov
0d-affine			3					4	3		7
1d-affine		7	11	3	3	6	4	48	52	2	18
2d-affine	1	59	85	1	1			27	20		48

have at least one affine dimension. For these polynomials, *partial enumeration* reduces the asymptotic complexity of the capacity miss counting.

As discussed by Section 3.3, the floor elimination techniques simplify non-affine stack distance polynomials with less splits than *partial enumeration* but are less generic and do not apply to all polynomials. Figure 14 shows the speedups for *equalization* compared to a baseline without *equalization* and *rasterization*. We disable both techniques since otherwise *rasterization* optimizes the polynomials normally handled by *equalization*. We observe a geometric mean speedup of 1.9x for the kernels that benefit. Figure 14 also compares the speedups for *rasterization* to a baseline without *rasterization*. We measure a geometric mean speedup of 1.9x for cholesky, lu, ludcmp, nussinov, and seidel-2d. Overall the floor elimination techniques reduce the number of counted pieces by more than 80% which results in bigger pieces with better counting performance.

A majority of the kernels perform well independent of problem size and number of cache hierarchy levels. Yet, the model execution times for kernels with non-affine polynomials are higher and problem size dependent. We mitigate this with efficient enumeration and floor elimination techniques.

4.4 Comparison to PolyCache and Dinero

The polyhedral cache model PolyCache [2] and the cache simulator Dinero IV [17] are alternative cache modeling tools. We compare their performance to HayStack.

PolyCache models set associative caches with an LRU replacement policy. We compare to the published results that show the performance for the default problem size of PolyBench 3.2 and adapt the configuration of our model to match the cache sizes of the published experiments (32KiB of L1 cache and 256KiB of L2 cache). The only difference is that we model fully associative caches instead of 4-way associative caches. Figure 15a shows an average speedup of 21x (geometric mean) of HayStack compared to PolyCache even though PolyCache computes the cache misses for all 1024 cache sets in parallel.

Dinero IV is a trace driven cache simulator which means the expected simulation cost are proportional to the number of memory accesses (Figure 1). Figure 15b shows the speedup of HayStack compared to the Dinero IV simulation times that include the trace generation with QEMU [3]. Dinero IV simulates the associativity of our test system while we model fully associative caches. As simulation and model run single core, the execution times are comparable. We measure an average speedup of 370x (geometric mean) for the large problem size that would be even bigger for the extra large problem size. Simulating full associativity further increases the average simulation time by factor 2.2x (geometric mean).

PolyCache models cache behavior in-depth, which allows developers to analyze the effects of set associativity and different write policies, but its high accuracy can make it costly to compute. Dinero IV works for small problem sizes but the cost increase for realistic problem sizes is dramatic.

4.5 Performance for Tiled Codes

A tiled code decomposes the iteration domain into tiles and executes tile-by-tile to improve the spacial locality. Tiling can double the loop nest depth which allows us to evaluate our approach for more complex codes. At the same time, estimating the benefits of tiling or even selecting optimal tile sizes is an important application for a cache model.

We employ the PPCG [39] source-to-source compiler to tile all PolyBench kernels with tile size 16. We limit the sum of all scheduling coefficients to one and disable loop fusion

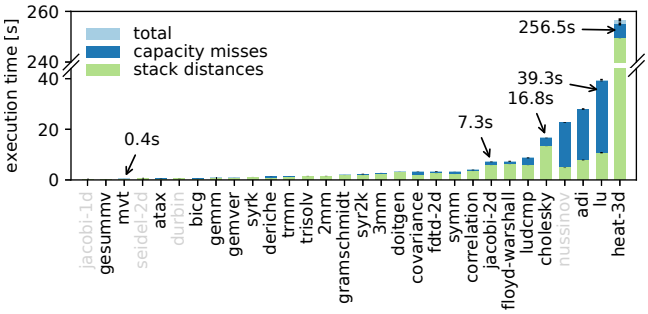


Figure 16. Execution times for the main components of HayStack for tiled versions of the PolyBench kernels. A few kernels (gray labels) have no rectangular tiling.

to obtain a rectangular tiling without loop skewing (time-tiling). All kernels except for jacobi-1d, durbin, seidel-2d, and nussinov have a rectangular tiling. Figure 16 shows the model execution times for the tiled kernels. Tiling makes the cache miss computation more expensive. Especially the stack distance computation of the head-3d kernel runs long. We attribute the cost increase to the more complex iteration domains and memory access patterns.

Tiling increases the model execution times but for a majority of the kernels the cache miss computation still takes only a few seconds.

5 Related Work

Cache behavior analysis is a prerequisite when tuning for the memory hierarchy. We distinguish three main approaches: 1) simulation, 2) profiling, and 3) analytical modeling.

Simulators Dinero [17] and CASPER [25] are examples of trace-based cache simulators that compute the cache misses for the full memory hierarchy. Sniper [10] and gem5 [7] have a broader scope and simulate the full system including the caches. All simulators execute the program to count the cache misses which means the simulation costs are proportional to the number of executed memory accesses.

Profiling Multiple works discuss the analysis of memory access traces to extract locality metrics. Mattson et al. [29] compute the stack distance using a linked list and derive the cache hit rate for different cache sizes. Tree based implementations [4, 31, 33] reduce the cost of the stack distance computation. Kim et al. [28] apply hashing and approximation to increase the efficiency. Ding et al. [15] discuss tree based approximate algorithms that reduce the time and space complexity of the stack distance computation and predict the stack distance histogram for arbitrary problem sizes given training inputs for few different problem sizes. Eklov et al. [16] sample the reuse distance for a few memory accesses and employ statistics to estimate stack distances and cache miss ratio. Xiang et al. [43] discuss five different locality metrics and show how to derive miss rate and reuse distance given the a single measure called average footprint which they compute with an efficient linear time algorithm [42]. A disadvantage of the profiling approaches is the acquisition and the handling of the large program traces. Chen et al. [14] sample the reuse time during compilation which allows them to estimate the cache miss ratio of complex loop nests.

Analytical models Agarwal et al. [1] develop an analytical model that uses parameters extracted from the program trace. Harper et al. [22] model set associative caches for regular loop nests. Cost models [8, 11, 27] allow compilers to decide if data-locality transformations are beneficial. All of these models only approximate the number of cache misses.

Ferdinand et al. [18] use abstract interpretation to model set associative LRU caches. Model-checking [13, 35] increases the accuracy of this analysis that distinguishes always hit, always miss, and not classified. Touzeau et al. [36] show how to attain high accuracy without costly model-checking. The abstract interpretation approaches are complementary to our cache model since they support dynamic control flow but approximate the cache misses of loop nests by classifying all instances of a memory access at once.

Ghosh et al. [20] derive cache miss equations to count the cache misses for perfect loop nests with data dependencies represented by reuse vectors [41]. Assuming an LRU replacement policy, a cache miss occurs if the number of solutions to a cache miss equality exceeds the cache associativity. Counting the solutions for every point of the iteration domain is expensive. Vera and Xue et al. [37, 44] thus sample the iteration domain to speedup the cache miss computation which allows them to perform approximate whole-program analysis. Cascaval et al. [9] compute the stack distance histogram symbolically for perfect loop nests with uniform data dependencies. They model fully associative caches with an LRU replacement policy and use statistics to model set associative caches. Chatterjee et al. [12] use Presburger formulas to express the set of compulsory and capacity misses of imperfect loop nests for associative caches. At the time, their approach was limited to small problem sizes and low associativity since the computation of analytical results for realistic hardware and even small benchmarks kernels was prohibitively complex. While Beyles et al. [6] did not address the cache miss problem, they use analytically computed stack distance to generate cache hints at runtime. Their stack distance computation, extended by our cache miss counting technique for non-affine polynomials, is the foundation of our cache model. PolyCache [2] presented the first analytical approach fast enough to compute the cache behavior of static control programs for interesting benchmark kernels and realistic hardware parameters. Its analytical model relates for every cache set successive accesses of distinct cache lines and repeatedly removes the shortest relations to model set associativity with LRU replacement policy. While PolyCache also uses symbolic counting techniques to avoid a complete

enumeration of the computation, its complexity increases with high associativity. Our work provides a fast analytical model for fully associative caches and shows that fully associative models introduce only small errors compared to measurements on actual hardware.

6 Conclusion

As memory behavior depends on the cache state, understanding the cost of memory accesses is much more difficult than understanding the cost of arithmetic instructions. With HayStack, we close this gap by providing developers with accurate information about the interaction of memory accesses with the large and deep cache hierarchy of modern processors. HayStack allows the programmer to predict memory access costs accurately and to develop programs well optimized for the memory hierarchy. When striving for ultimate performance, both a good baseline and an accurate surrogate model accelerates empirical tuning. As a result, cache-aware program optimization becomes accessible.

Responsiveness is key for the adoption of any cache model. We demonstrate excellent often problem size independent response times that for the first time make analytical cache modeling practical. In addition, the cache size independent costs allow our model to easily scale to future hardware. We show the practicality of our deliberate decision against high fidelity and in favor of a generic fully associative cache model. The proposed model is robust to memory layout choices and hardware implementation details and yet reaches very high accuracy on real hardware across a wide range of computations.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880), the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016), and ARM Holdings plc and Xilinx Inc in the context of Polly Labs. We also would like to thank the Swiss National Supercomputing Center for providing the computing resources.

References

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. 1989. An analytical cache model. *ACM Transactions on Computer Systems (TOCS)* 7, 2 (1989), 184–215.

[2] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 32.

[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[4] Bryan T Bennett and Vincent J. Kruskal. 1975. LRU stack processing. *IBM Journal of Research and Development* 19, 4 (1975), 353–357.

[5] Kristof Beyls and Erik H. D'Hollander. 2001. Reuse Distance as a Metric for Cache Behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*. 617–662.

[6] Kristof Beyls and Erik H D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[8] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, Vol. 43. ACM, 101–113.

[9] Calin CaBcaval and David A Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 150–159.

[10] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 52.

[11] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. 1994. *Compiler optimizations for improving data locality*. Vol. 29. ACM.

[12] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. 2001. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices* 36, 5 (2001), 286–297.

[13] Sudipta Chattopadhyay and Abhik Roychoudhury. 2013. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems* 49, 4 (01 Jul 2013), 517–562.

[14] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality analysis through static parallel sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 557–570.

[15] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Notices*, Vol. 38. ACM, 245–257.

[16] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 55–65.

[17] Jan Elder and Mark D. Hill. 2003. Dinero IV Trace-Driven Uniprocessor Cache Simulator.

[18] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. 1999. Cache Behavior Prediction by Abstract Interpretation. *Sci. Comput. Program.* 35, 2-3 (Nov. 1999), 163–189.

[19] M. J. Fischer and M. O. Rabin. 1974. *SUPER-EXPONENTIAL COMPLEXITY OF PRESBURGER ARITHMETIC*. Technical Report. Cambridge, MA, USA.

[20] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 4 (1999), 703–746.

[21] Christoph Haase. 2018. A Survival Guide to Presburger Arithmetic. *ACM SIGLOG News* 5, 3 (July 2018), 67–82.

[22] John S Harper, Darren J Kerbyson, and Graham R Nudd. 1999. Analytical modeling of set-associative cache behavior. *IEEE Trans. Comput.* 48, 10 (1999), 1009–1024.

[23] Mark D Hill. 1987. *Aspects of cache memory and instruction buffer performance*. Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.

[24] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. ACM, 73.

[25] Ravi Iyer. 2003. On modeling and analyzing cache hierarchies using CASPER. In *null*. IEEE, 182.

[26] LENSTRA JR. H.W. 1983. Integer Programming with a Fixed Number of Variables. *Report 81-03, Mathematisch Instituut Amsterdam (1981)* 8 (11 1983).

[27] Ken Kennedy and Kathryn S McKinley. 1992. Optimizing for parallelism and data locality. In *Proceedings of the 6th international conference on Supercomputing*. ACM, 323–334.

[28] Yul H Kim, Mark D Hill, and David A Wood. 1991. *Implementing stack simulation for highly-associative memories*. Vol. 19. ACM.

[29] Richard L Mattson, Jan Gecsei, Donald R Slutz, and Irving L Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.

[30] Danh Nguyen Luu. 2018. *The Computational Complexity of Presburger Arithmetic*. Ph.D. Dissertation. UCLA.

[31] Frank Olken. 1981. Efficient methods for calculating the success function of fixed space replacement policies. (1981).

[32] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: https://sourceforge.net/projects/polybench/* (2012).

[33] Rabin A Sugumar and Santosh G Abraham. 1993. *Efficient simulation of caches under optimal replacement with applications to miss characterization*. Vol. 21. ACM.

[34] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[35] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2017. Ascertaining Uncertainty for Efficient Exact Cache Analysis. *CoRR* abs/1709.10008 (2017).

[36] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and Exact Analysis for LRU Caches. *Proc. ACM Program. Lang.* 3, POPL, Article 54 (Jan. 2019), 29 pages.

[37] Xavier Vera and Jingling Xue. 2002. Let's study whole-program cache behaviour analytically. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*. IEEE, 175–186.

[38] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.

[39] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[40] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48, 1 (2007), 37–66.

[41] Michael E Wolf and Monica S Lam. 1991. A data locality optimizing algorithm. In *ACM Sigplan Notices*, Vol. 26. ACM, 30–44.

[42] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 350–360.

[43] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 343–356.

[44] Jingling Xue and Xavier Vera. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput.* 53, 5 (2004), 547–566.