Falcon: A Scalable Analytical Cache Model

ARJUN PITCHANATHAN, University of Edinburgh, UK
KUNWAR GROVER[*], Advanced Micro Devices, UK
TOBIAS GROSSER[†], University of Cambridge, UK

Compilers often use performance models to decide how to optimize code. This is often preferred over using hardware performance measurements, since hardware measurements can be expensive, limited by hardware availability, and makes the output of compilation non-deterministic. Analytical models, on the other hand, serve as efficient and noise-free performance indicators. Since many optimizations focus on improving memory performance, memory cache miss rate estimations can serve as an effective and noise-free performance indicator for superoptimizers, worst-case execution time analyses, manual program optimization, and many other performance-focused use cases. Existing methods to model the cache behavior of affine programs work on small programs such as those in the Polybench benchmark but do not scale to the larger programs we would like to optimize in production, which can be orders of magnitude bigger by lines of code. These analytical approaches hand off the whole program to a Presburger solver and perform expensive mathematical operations on the huge resulting formulas. We develop a scalable cache model for affine programs that splits the computation into smaller pieces that do not trigger the worst-case asymptotic behavior of these solvers. We evaluate our approach on 46 TorchVision neural networks, finding that our model has a geomean runtime of 44.9 seconds compared to over 32 minutes for the state-of-the-art prior cache model, and the latter is actually smaller than the true value because the prior model reached our four-hour time limit on 54% of the networks, and this limit was never reached by our tool. Our model exploits parallelism effectively: running it on sixteen cores is 8.2x faster than running it single-threaded. While the state-of-the-art model takes over four hours to analyze a majority of the benchmark programs, Falcon produces results in at most 3 minutes and 3 seconds; moreover, after a local modification to the program being analyzed, our model efficiently updates the predictions in 513 ms on average (geomean). Thus, we provide the first scalable analytical cache model.

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

Additional Key Words and Phrases: static analysis, performance analysis, cache modeling

ACM Reference Format:
Arjun Pitchanathan, Kunwar Grover, and Tobias Grosser. 2024. Falcon: A Scalable Analytical Cache Model. *Proc. ACM Program. Lang.* 8, PLDI, Article 222 (June 2024), 25 pages. https://doi.org/10.1145/3656452

1 INTRODUCTION

Compilers often use performance models to evaluate and compare different optimized versions of code [Adams et al. 2019; Chen et al. 2018b; Jia et al. 2020; Kaufman et al. 2021; Narayanan et al. 2019]. This is often preferred over collecting performance measurements from a real machine as the latter can be expensive, limited by hardware availability, or infeasible (such as during ahead-of-time

[*]Work done while a student at IIIT Hyderabad and visiting the University of Edinburgh.
[†]Work primarily done while at the University of Edinburgh.

Authors' addresses: Arjun Pitchanathan, University of Edinburgh, UK, arjun.pitchanathan@ed.ac.uk; Kunwar Grover, Advanced Micro Devices, UK, KunwarShaanjeetSingh.Grover@amd.com; Tobias Grosser, University of Cambridge, UK, tobias.grosser@cst.cam.ac.uk.

ACM 2475-1421/2024/6-ART222
https://doi.org/10.1145/3656452

compilation). Moreover, hardware measurements are non-deterministic, which can result in a lack of reproducibility. Even when hardware measurements are used to compare candidate versions of the code, performance models are often used to generate a short list of potential candidates. Both LLVM's [LLVM Contributors [n. d.]] and GCC's [GCC Contributors [n. d.]] auto-vectorizers use cost models to choose which optimizations to apply or to choose the parameters of the optimization pass to apply such as the unroll factor. Many important optimizations such as operator fusion and loop tiling primarily focus on improving memory performance, and would therefore benefit from a memory performance model to decide when and how to apply these optimizations.

We present Falcon, a scalable analytical performance model for memory cache performance. Falcon outputs a predicted cache miss rate for each statement in a given affine program. We evaluate our tool on a benchmark of 46 TorchVision [Marcel and Rodriguez 2010] neural networks (Section 5.1), finding that it returns results in 44.9 seconds on average (geomean). While there has been prior work on analytical cache modeling [Gysi et al. 2019; Morelli and Reineke 2022], it focused on small single-kernel applications and did not scale to large programs; in a majority of the TorchVision benchmarks, these tools take over four *hours* per program.

Prior analytical approaches are slow because the first step they perform is to convert the entire input program into one big formula in Presburger arithmetic [Haase 2018]. Unfortunately, solving Presburger formulas does not scale well with formula size. We propose a more surgical approach that traverses a structured loop representation of the program and only uses the more expensive mathematical optimization algorithms to model relationships between specific statements in the source code. Combining an AST-based approach with the Presburger-based methods [Shirako et al. 2014] enables greater performance.

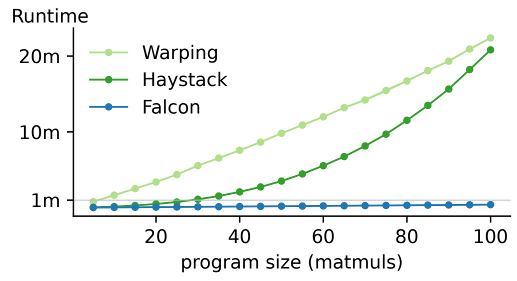


Fig. 1. Both the SOTA cache models scale poorly with the size of the program, whereas Falcon scales well. This data was collected by generating programs with a succession of matrix multiplications and running all three models on these. See Section 5.2 for details.

Moreover, for each statement, we overapproximate the region of code that can affect its cache performance, greatly reducing the number of pairs of statements that we need to consider. Thus, for practically relevant cache configurations and programs, we end up comparing each statement against a constant number of other statements instead of against the whole program. Our algorithm scales far better than existing approaches (Figure 1). In addition, our tool computes the miss rate prediction for each statement separately, and so is highly parallelizable across statements.

Our tool provides the ability to trade-off between speed and accuracy by using our partial linearization feature (Section 4.8). With this enabled, our geomean runtime becomes 3m47s, as compared to at least 32 minutes for the baselines (capped by the four-hour timeout). We obtain a correlation of 0.98 between our miss rate predictions and the real hardware measurements.

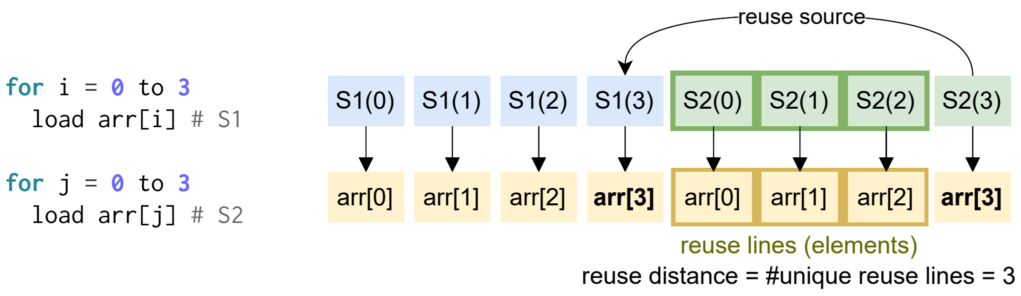
Our model supports efficiently updating the cache results after local changes to the program, which would be useful for applications in search space exploration and performance engineering. We achieve this by again leveraging a distance-based optimization, using the AST representation to detect and prune irrelevant computations. Using this, we narrow down a small set of statements whose cache performance could possibly be affected by the modification. We now use the fact that our algorithm models each statement separately to precisely recompute results for only the small set of statements that we have narrowed down to.

In summary, we contribute an analytical cache model that

- is scalable (Section 5.4),
- gives predictions well-correlated to real measurements (Section 5.3),
- can efficiently update its predictions after local program modifications (Section 5.4.2), and
- is highly parallelizable (Section 5.4.1).

2 BACKGROUND

We describe the tools we use: the theory of Presburger arithmetic and polyhedral compilation.



```
for i = 0 to 3
  load arr[i] # S1

for j = 0 to 3
  load arr[j] # S2
```

Fig. 2. Example program to illustrate terminology.

2.1 Affine Programs

We model *affine* programs, a class of programs that includes important applications like neural networks, image processing, and scientific computing. These programs have an important and useful property: they have no data-dependent branches and all branch conditions are *affine*. i.e., they are conjunctions and disjunctions of affine inequalities, of the form $a_1x_1 + \dots a_nx_n \geq c$. These are programs that can be represented as loop nests whose induction variables are incremented by a constant step size, and are constrained only by affine inequalities that depend solely on outer loop induction variables. Moreover, the bodies of loops contain only memory accesses whose index expressions are affine in the outer loop induction variables. Finally, the loop body may have branching control flow, but the branching condition must be an affine expression in the outer induction variables.[1]

Figure 2 shows two loops, each with one *statement*, $S1$ and $S2$ respectively. In general when we refer to statements in this work, we are referring to statements that access (load from or store to) memory, as all other statements (except control flow) can be ignored for the purpose of cache modeling. $S1$ is executed once for each value of i in the set $\{i \in \mathbb{Z} \mid 0 \leq i \leq 3\}$. This set is called the *iteration domain* of S1, which we denote as $\mathcal{I}_{S1}$. We will omit the requirement that i be an integer going forward as all numbers we deal with will be integers. $S2$'s iteration domain is $\{j \mid 0 \leq j \leq 7\}$. Each execution of $S1$ for a given i, denoted $S1(i)$, is called an *instance* of the statement S1.

We implement our model on the program AST of a static single-assignment (SSA) based domain-specific intermediate representation (IR) that models such affine programs at the correct level of abstraction. The language's AST preserves lexical control-flow structures like for loops and if conditions. Each memory access instruction canonically performs a single load or store, reducing the ambiguity in ordering that arises when a single line contains multiple memory accesses. The

[1]While a more general notion of affine programs supports all these depending on some *program parameters* as well, we do not deal with this case in the present work.

MLIR compiler infrastructure [Lattner et al. 2021] introduced a domain-specific intermediate representation called the Affine dialect that satisfies these criteria, which we use in our prototype; it can look quite similar to the pseudocode in the example above. We design our cache modeling algorithm to operate on such an AST.

2.2 Terminology

Consider a single-level, fully associative cache with C cache lines. Given a statement instance $S(\mathbf{i})$ that performs a memory access, we classify it as either a cache *hit* or a *miss*.

Definition 2.1. ***Reuse source of a statement instance*** $S(\mathbf{p})$*: the most recent access to the same cache line as* $S(\mathbf{p})$*. Undefined if* $S(\mathbf{p})$ *makes the first access to that cache line.*

Say every cache line contains exactly one array element. Then in the example, the reuse source of $S2(3)$ is $S1(3)$, as $S1(3)$ accesses the same array element `arr[3]` as $S2(3)$, and moreover is the most recently executed statement instance that accesses that element. When we generalize to multiple elements per cache line, we will compare on the basis of the cache lines that statements access instead of the array elements. If the reuse source of A is undefined then A is a miss and is called a *compulsory miss.*

Definition 2.2. ***Compulsory miss****: a miss caused by the first access to a cache line. Such a miss would occur in any single-level LRU cache irrespective of cache size.*

In the example, $S1(3)$ has no reuse source as it accesses its location for the first time, so it incurs a compulsory miss. If the reuse source is defined, we need to analyze further. In such a case the number of cache lines accessed between the instance and its reuse source is called its *reuse distance.*

Definition 2.3. ***Reuse distance of a statement instance*** $S(\mathbf{p})$*: the number of unique cache lines accessed between* $S(\mathbf{p})$ *and its reuse source. Denoted by* `reuseDist`(A)*. Not defined if* $S(\mathbf{p})$ *has no reuse source.*

In the example, the reuse distance of $S2(3)$ is 3. If $S(\mathbf{p})$ does not compulsorily miss, then it incurs a miss iff `reuseDist`$(A) \geq$ `cacheSize`. Such a miss is called a *capacity miss.* was

Definition 2.4. ***Capacity miss****: a miss that is not compulsory, which would not occur if the capacity of the cache was increased to exceed the reuse distance.*

If `cacheSize` $= 3$ then $S2(3)$ incurs a capacity miss. If `cacheSize` ≥ 4 then we hit the cache. We now define some general terminology. Bold-faced variables like $\mathbf{i}, \mathbf{p}, \mathbf{q}$ denote tuples. Subscripts like $\mathbf{i}_a$ denote the a-th element in $\mathbf{i}$ and $\mathbf{i}_{a:b}$ denotes the subrange of $\mathbf{i}$ from a to b (both 1-indexed and inclusive). $\mathcal{I}$ is the set of all instances of all statements in the program under consideration, and $\mathcal{I}_S$ denotes the set of all instances of a particular statement S.

2.3 Presburger Arithmetic

We analyze affine programs with static control flow. Such programs can be modeled conveniently using Presburger arithmetic. For example, the iteration domain $\{i \in \mathbb{Z} \mid 0 \leq i \leq 3\}$ of $S1$ in Figure 2 can be expressed in this theory. The worst-case complexity of deciding Presburger formulas is at least double-exponential [Fischer and Rabin 1998], but these operations are efficient in practice for small formulas.

A Presburger set over n variables $x_1, \ldots x_n \in \mathbb{Z}$ is defined by a Presburger formula P, which is any logical formula involving affine inequalities like $\sum_i a_i x_i \geq c$. The formula may involve existential quantifiers. We can also express floor divisions and modulos of an affine expression by a constant.

Presburger relations are defined similarly, except that the variables are partitioned into domain and range variables.

We use the following operations on sets and relations:

- intersection, union, set complement, and set difference
- inverting and composing relations
- computing the domain and range of a relation
- computing the maximum and minimum value of an affine expression over elements in a Presburger set [Lovász and Scarf 1992]
- computing the lexicographically minimum and maximum elements in a set [Feautrier 1988]
- computing the parametric cardinality of a relation, obtaining a mapping from each domain element to the cardinality of the image of that element [Verdoolaege et al. 2007], represented as a piece-wise polynomial over the domain variables of the relation and floor divisions thereof

We implement our model using the MLIR Presburger library [Pitchanathan et al. 2021] to convert the Affine IR to Presburger sets and relations, and then pass these on to the isl [Verdoolaege 2010] and barvinok [Verdoolaege 2007] libraries to handle the set operations.

3 LIMITATIONS AND HARDWARE MODEL

Modern caches implement complex and usually undocumented policies that define their exact behavior. For example, the replacement policies of common Intel and AMD CPUs have not been publicly disclosed to the best of our knowledge; we model a Least-Recently Used replacement policy [Patterson and Hennessy 2013]. We do not support multi-threading or shared caches due to interference from other cores; the noise and non-determinism inherent in the multi-threaded setting makes performance estimation less viable. We do not model prefetchers. We ignore registers and register spilling and support a write-through write-allocate write policy. Finally, we model fully associative cache hierarchies with support for both inclusive and exclusive hierarchical caches. As such, we model compulsory and capacity misses, and do not model conflict misses. In this work, we show that a model of such caches is capable of giving results that are accurate and well-correlated to measurements on real hardware (Section 5.3).

We assume that arrays do not alias; this is typically the case in code generated from domain-specific compilation frameworks to accelerate performance-critical applications. The presence of aliasing arrays would likely prevent most code transformations, so information about cache performance has less utility in such settings. Finally, our cache model is designed to support affine programs as defined in Section 2.1.

4 ALGORITHM

We first describe our algorithm for a single-level cache where the size of each array element is equal to the size of a cache line. We then show how to extend this to cache lines containing multiple elements and multi-level cache hierarchies. Finally, we describe further optimizations to the algorithm.

To keep the paper self-contained, we explain all parts of the algorithm that are necessary to put our contributions into context. The elements novel to the present work are:

- the idea of breaking down the problem into smaller sub-parts, rather than operating on one big Presburger formula, in particular by leveraging dependence analysis techniques (Section 4.3), thus improving performance and enabling parallelism (Section 4.2),
- a number of distance-based optimizations (Section 4.4), including adding support for incremental recomputation of predictions after local modifications to the program (Section 4.4.3),

Algorithm 1 High-level outline of the algorithm for single-level caches. We then describe an efficient implementation of these steps (Section 4.1), generalization of the model (Section 4.8), and further optimizations (Section 4.5, Section 4.4).

```
1: function ComputeCacheMissCount(program $P$, cache size $C$)
2:     compulsoryMisses ⟵ 0, capacityMisses ⟵ 0
3:     for statement Sink ∈ $P$ do
4:         deps ⟵ ComputeCacheDependences(Sink)
5:         compulsoryMisses += #{Sink($\mathbf{p}$) ∈ $\mathcal{I}_{\texttt{Sink}}$ | deps(Sink($\mathbf{p}$)) is undefined}
6:         reuseInstances ⟵ {(Sink($\mathbf{p}$), $U(\mathbf{r})$) ∈ domain(deps) × $\mathcal{I}$ | Sink($\mathbf{p}$) ≺ $U(\mathbf{r})$ ≺ deps(Sink($\mathbf{p}$))}
7:         reuseLines ⟵ reuseInstances ∘ access
8:         capacityMisses += #{Sink($\mathbf{p}$) ∈ domain(deps) | #reuseLines(Sink($\mathbf{p}$)) ≥ $C$}
9:     return compulsoryMisses + capacityMisses
```

- two optimizations to the threshold counting component (Section 4.5), and
- the partial linearization feature to improve accuracy (Section 4.8).

4.1 Algorithm for Single-Level Cache

The algorithm analytically obtains the same results that a simulation would, but handles all accesses performed by a single memory access statement `Sink` in a loop nest at the same time. Thus, the runtime depends on the program size (number of memory access *statements* in P) rather than the number of memory *accesses* in its execution trace.

A high-level overview of the algorithm is presented in Algorithm 1. We loop through each memory access statement `Sink` in the program and compute its cache miss count separately.

```
for i = 0 to 3
  load arr[i]     # Statement S1
for j = 0 to 7
  load arr[j]     # Statement S2
```

Listing 1. Example input for Algorithm 1.

Computing dependences. We first compute the dependence relation for the sink deps : $\mathcal{I}_{\texttt{Sink}} \to \mathcal{I}$, which is a partial function mapping each statement instance `Sink`($\mathbf{i}$) to its reuse source, when one exists (Line 4). The details of this computation are described in Section 4.3. For example, let's analyze Figure 1 with statement S2 as the sink. Its set of statement instances is $\mathcal{I}_{S2} = \{S2(j) \mid j \in \{0, 7\}\}$. We then have

$$deps(S2(j)) = \begin{cases} S1(j), & 0 \le j \le 3 \\ \text{undefined}, & 4 \le j \le 7 \end{cases}$$

Counting compulsory misses. Statement instances accessing a cache line for the first time correspond to compulsory misses, so we add their count to the compulsory misses (Line 5). In the example, we have

$$\begin{aligned} \text{compulsory miss count for } S2 &= \#\{j \in \{0, \dots 7\} \mid \texttt{deps}(S2(j)) = \text{undefined}\} \\ &= \#\{j \in \{0, \dots 7\} \mid 4 \le j \le 7\} \\ &= 4 \end{aligned}$$

Computing `reuseInstances`. Using the dependence relation, we compute (Line 6) a relation `reuseInstances` mapping each instance `Sink`($\mathbf{p}$) to the set of statement instances of the program that are executed between `Sink`($\mathbf{p}$) and its reuse source (not including the dependence). To compute this, we use the fact that the execution ordering between statements is expressible in Presburger arithmetic (Section 4.6). The symbol $\prec$ refers to comparison under this ordering. In the example, $S1(i) \prec S2(j)$ for all valid values of i and j, $S1(i_1) \prec S1(i_2)$ iff $i_1 < i_2$, and similarly $S2(j_1) \prec S2(j_2)$ iff $j_1 < j_2$. Let's first compute the instances of $S1$ that lie in $\texttt{reuseInstances}(S2(j))$ for each instance $S2(j)$ of $S2$. Noting that the domain of `reuseInstances` is only those instances of $S2$ that have reuse sources, we obtain

$$\begin{aligned}
&\{S1(i) \in \mathcal{I}_{S1} \mid \mathsf{deps}(S2(j)) \prec S1(i) \prec S2(j)\} \\
&= \{S1(i) \in \mathcal{I}_{S1} \mid S1(j) \prec S1(i) \prec S2(j)\} && \text{(definition of } \mathsf{deps}\text{)} \\
&= \{S1(i) \in \mathcal{I}_{S1} \mid S1(j) \prec S1(i)\} && (S1 \text{ always precedes } S2) \\
&= \{S1(i) \in \mathcal{I}_{S1} \mid j < i\} && \text{(definition of } \prec\text{)} \\
&= \{S1(j+1) \ldots S1(3)\}, && \text{(definition of } \mathcal{I}_{S1}\text{; if } j = 3 \text{ then the set is empty)}
\end{aligned}$$

where the last equality is because the set of instances of $S1$ is $\mathcal{I}_{S1} = \{S1(0), S1(1), S1(2), S1(3)\}$. By doing a similar calculation for the $S2$ reuse instances, we obtain that the set of instances of $S2$ in $\texttt{reuseInstances}(S2(j))$ is $\{S2(0) \ldots S2(j-1)\}$, so that overall, we obtain

$$\texttt{reuseInstances}(S2(j)) = \{S1(j+1), \ldots S1(3)\} \cup \{S2(0) \ldots S2(j-1)\}.$$

Computing `reuseLines`. The next function we need is `access`, which maps any statement instance to the memory location it accesses. In our example, $\texttt{access}(S1(i)) = \texttt{arr[i]}$ and $\texttt{access}(S2(i)) = \texttt{arr[i]}$. Composing `reuseInstances` with `access` gives `reuseLines` (Line 7), the set of array locations accessed between a sink statement instance and the most recent access to the same cache line. In the example,

$$\begin{aligned}
\texttt{reuseLines}(S2(j)) &= \{\texttt{arr[j+1]}, \ldots \texttt{arr[3]}\} \cup \{\texttt{arr[0]}, \ldots \texttt{arr[j-1]}\} \\
&= \{\texttt{arr[i]} \mid i \in \{0, 3\} \wedge i \neq j\}
\end{aligned}$$

Computing reuse distances and counting capacity misses. For each sink instance, the number of such cache lines, the cardinality $\#\mathit{reuseLines}(\texttt{Sink}(\mathbf{p}))$, is equal to the reuse distance of $\texttt{Sink}(\mathbf{p})$. The memory access performed by the instance misses the cache iff this distance is at least the size of the cache (Section 2.2), so we count the number of such instances and add that to the number of capacity misses. In our simple example, the reuse distance is the same for all instances of $S2$:

$$\begin{aligned}
\#\texttt{reuseLines}(S2(j)) &= \{\texttt{arr[i]} \mid i \in \{0, 3\} \wedge i \neq j\} \\
&= 3.
\end{aligned}$$

In this step, if multiple statement instances had accessed the same array element, it would be counted only once. We count the number of *unique* array locations here. Let's say the cache contains two cache lines and, for now, we have assumed that each cache line corresponds to exactly one array element. Then we have

$$\begin{aligned}
\text{capacity miss count for } S2 &= \#\{S2(j) \in \mathcal{I}_{S2} \mid \#\texttt{reuseLines}(S2(j)) \geq 2\} \\
&= \{S2(j) \in \mathcal{I}_{S2} \mid 3 \geq 2\} \\
&= \#\mathcal{I}_{S2} \\
&= 4.
\end{aligned}$$

Finally, we return the total number of compulsory and capacity misses across all statements in the program – in our example, this is $4 + 4 = 8$ misses.

4.2 Parallelism

Note that each sink is handled independently, so the whole algorithm is embarrassingly parallel over the choice of the sink statement. No prior model was capable of exploiting parallelism. Running the algorithm on 16 threads gives a geomean speedup of 8.2x over a single-threaded run (Section 5.4.1).

Algorithm 2 Dependence analysis for a single instance.

```
1:  function ComputeCacheDependences(program $P$, statement instance Sink($\mathbf{p}$))
2:      deps ← {}
3:      loc ← the location that Sink($\mathbf{p}$) accesses.
4:      Let $L_1, \dots L_n$ be the loops surrounding Sink in increasing order of depth.
5:      for $i = n$ to 1 do
6:          Let $\mathcal{S}_{above}$ be the set of statements in $L_i$ lying above Sink and in fact lying above $L_{i+1}$ (if $i < n$).
7:          Let $\mathcal{I}_{above}$ be the set of instances of statements in $\mathcal{S}_{above}$ that execute in
            • the same iterations of $L_1, \dots L_i$ as Sink($\mathbf{p}$).
8:          if any instances in $\mathcal{I}_{above}$ access loc then
9:              return the last-executed such instance.
10:
11:         Let $\mathcal{I}_{prev}$ be the set of instances of statements in $L_i$ that execute in
            • the same iterations of the surrounding loops $L_1, \dots L_{i-1}$ as Sink($\mathbf{p}$), but
            • in an earlier iteration of $L_i$.
12:         if any instances in $\mathcal{I}_{prev}$ access loc then
13:             return the last-executed such instance.
14:
15:     for each statement S above $L_1$ do            ▷ (start immediately above $L_1$ and iterate upwards)
16:         if any instance of S accesses loc then
17:             return the last-executed such instance.
18:     return null
```

4.3 Value-based Dependence Analysis

The dependence algorithm finds the most recently executed statement instance accessing the same location as a given sink instance. When each cache line contains exactly one element, this is equivalent to the reuse source. We describe our extension to situations where more than one element per cache line in Section 4.8. Algorithm 2 is a high-level description of how the algorithm works for a single instance Sink($\mathbf{p}$), which provides sufficient context to explain our optimizations. The full dependence algorithm [Maslov 1994] handles all instances of the sink at once. We then optimize this further for cache modeling (Section 4.4).

Let the list of enclosing loops of the sink be $L_1, \dots L_n$, in increasing order of depth. Our sink instance occurs in iteration $\mathbf{p}_1, \dots \mathbf{p}_n$ of these loops. First, we look for dependences of the sink instance among other statements instances within the same iteration of the loops $L_1, \dots L_n$. For example, if we were looking for dependences of $S6(2)$ in Figure 2, we would at this stage only consider statements occurring in $i = 2$ of the outer loop. In this case, we immediately find a dependence to $S5(2)$.

If we don't find the dependence in the same iteration of L_n, then we look to previous iterations (Lines 11-13). If no statement in L_n accesses loc in iteration $\mathbf{p}$, then we consider prior iterations of L_n. It would be too slow to go through each iteration one at a time, so we handle all prior iterations at once. In the example, say we want to compute the dependence of $S5(3)$. It has no statements above it, so we immediately look into all previous iterations i.e., iterations with $0 \le i < 3$. We find that in iteration $i = 2$, both $S5(2)$ and $S6(2)$ access the location that $S5(3)$ accesses, so both are candidates to be the dependence. Since $S6(2)$ executes last, it is the true dependence here.

If we still don't find a dependence, then we look outside the immediately enclosing loop L_n of the sink statement; we look for dependences in the next loop L_{n-1}. If we don't find any there, then we continue further up, until we have looked at all statements in the outermost enclosing loop L_1. After this, if we still have not found a dependence, then we look for statements that appear above L_1, statements that do not share any common loops with the sink statement. When looking for the dependence of $S6(4)$ in the example, we don't find it in any statement instance inside the loop, so we look above at first statement $S4$. When we don't find it there, we look at $S3$, and do not find it there either. Note that we do not look below the loop at statement $S7$ since it executes after $S6(4)$ (and in fact after any instance of statements in $L1$). When we don't find a dependence to any statements lying above L_1 either, we return `null` indicating that no dependence exists.

```
load arr[0]       # Statement S3
load arr[1]       # Statement S4
for i = 0 to 4
  load arr[2]     # Statement S5
  load arr[i]     # Statement S6
load arr[5]       # Statement S7
```

Listing 2. Example input program for dependence analysis. To avoid confusion we do not reuse statement labels used in earlier examples.

The dependence algorithm we implement emulates the same process described above, but does so *simultaneously for all instances of the sink* using the Presburger solver. Like Algorithm 2, it iterates through all loops from L_n to L_1. At each loop L_i, for instances that haven't found a dependence yet, it looks first for (a) dependences in the same iteration of L_i as that instance, and then (b) in prior iterations of L_i. Finally, the implemented algorithm does not create a large and fragmented set containing instances of many statements in the loop like $\mathcal{I}_{above}$ and $\mathcal{I}_{prev}$. Instead, it iterates through each statement to be considered one at a time and looks for dependences in each separately, and then chooses the last-executed instance among the dependences found to each statement.

With this overview of the algorithm as context, we now describe our optimizations. The first optimization is in "inlining" Algorithm 1 into the dependence algorithm: at each step that some dependences are found, we immediately then compute the number of cache misses among the instances that found dependences. Since Presburger solvers have exponential worst-case runtime, it is much more efficient to process many small dependence relations than one big one.

4.4 Distance-based Optimizations

We start with some intuition and then give details in the subsections that follow. In the dependence algorithm, if even one sink instance is a compulsory miss, then we are forced to iterate through the whole program searching for its dependence, and never finding it. This is also the case when we have a single instance dependent on a very far away statement. In both these cases, the sink instance is a cache miss: if we go far enough above the sink then no matter where the exact dependence lies, we know that it is going to have a large or undefined reuse distance, corresponding to misses. We early exit the dependence whenever we cross this reuse distance threshold. All leftover sink instances can then be classed as misses.

4.4.1 Within-loop dependences. When we are looking at loop L_i in the dependence algorithm, we can compute the number of unique cache lines this loop accesses, parametric in the iteration of the loops $L_{1:i}$. If the piece-wise polynomial representing this count is piece-wise linear, we can use the Presburger solver to compute the minimum and maximum number of cache lines accessed in single iterations of this loop.

If all single iterations access only a "small" number of cache lines, then all dependences between iterations that are executed "close" to each other correspond to cache hits. Formally, if all iterations access at most max cache lines, then all dependences between iterations that are at most $\lfloor \texttt{max/cacheSize} \rfloor$ apart represent cache hits, so we can remove these from the obtained dependences before the final threshold counting. Since the counting step can be expensive, it helps to eliminate dependences before that.

We still have to make sure that dependences to statement instances in these loop iterations are found; we can only skip the counting step. This is because the corresponding sink instances should be marked as having found their dependence, so that those instances do not incorrectly get mapped to some far away dependences and become wrongly classed as misses. The next two optimizations allow us to skip part of the dependence analysis entirely.

4.4.2 Top-level Statements. When dealing with programs that apply a long series of small loop kernels, the size of any given loop is typically much smaller than the entire program. This means that most of the statements we process will lie outside and above the loop nest of the sink statement, which are processed in lines 15-17 of Algorithm 2. For this scenario, we introduce a heuristic to underapproximate the reuse distance of all remaining sink instances when we are in this phase of the algorithm.

We start with an example (Figure 3). Assume that cacheSize = 8. Let's find the dependence of $S12(4)$. Clearly, it has no dependence in any iteration of its own loop nest. We then consider statement $S11$ and find no dependence there either. Now consider $S10$. Before we even look at what $S10$ accesses, we know that if we find a dependence there, reuseInstances$(S12(4))$ must necessarily contain all instances of $S11$. In other words, the reuse lines definitely include all cache lines accessed by $S11$. Thus, the reuse distance, it exists, is lower bounded by the total number of unique cache lines accessed by the $S11$, which is 4.

```
for i = 0 to 4
  load A[i]          # Statement S8
for i = 0 to 4
  load C[i]          # Statement S9
for i = 0 to 4
  load B[i]          # Statement S10
for i = 0 to 4
  load B[i]          # Statement S11
for i = 0 to 4
  load A[i]          # Statement S12
```

Listing 3. Example input program for reuse distance underapproximation among top-level statements.

In general, as we move up the program, let $\ell_1, \ldots \ell_k$ be the top-level loops lying between the sink's top-level loop and the current one. Then the reuse distance is lower bounded by the number of unique cache lines accessed in all instances of these loops. Note that the cache lines have to be unique. In the example, after processing and not finding any dependences in $S10$, the lower bound on reuse distance remains 4. But after processing $S9$ and not finding a dependence there, we can deduce that the reuse distance must be at least 8. Since we said that the cache size is 8, this means that the sink instance access was definitely a cache miss. Thus, we don't even have to iterate further up the program to find the actual dependence; we can immediately stop and classify the instance as a miss.

In summary, the algorithm tracks the set of cache lines accessed by all statements in loop nests lying between the sink loop nest and the current one. Whenever we complete searching instances in a loop nest and are about to move on to the next one, we update the set of cache lines and compute its cardinality. If the cardinality is at least the cache size, then we know that the reuse distance is at least the cache size, so the sink instance is a cache miss. We therefore mark it as such and return immediately.

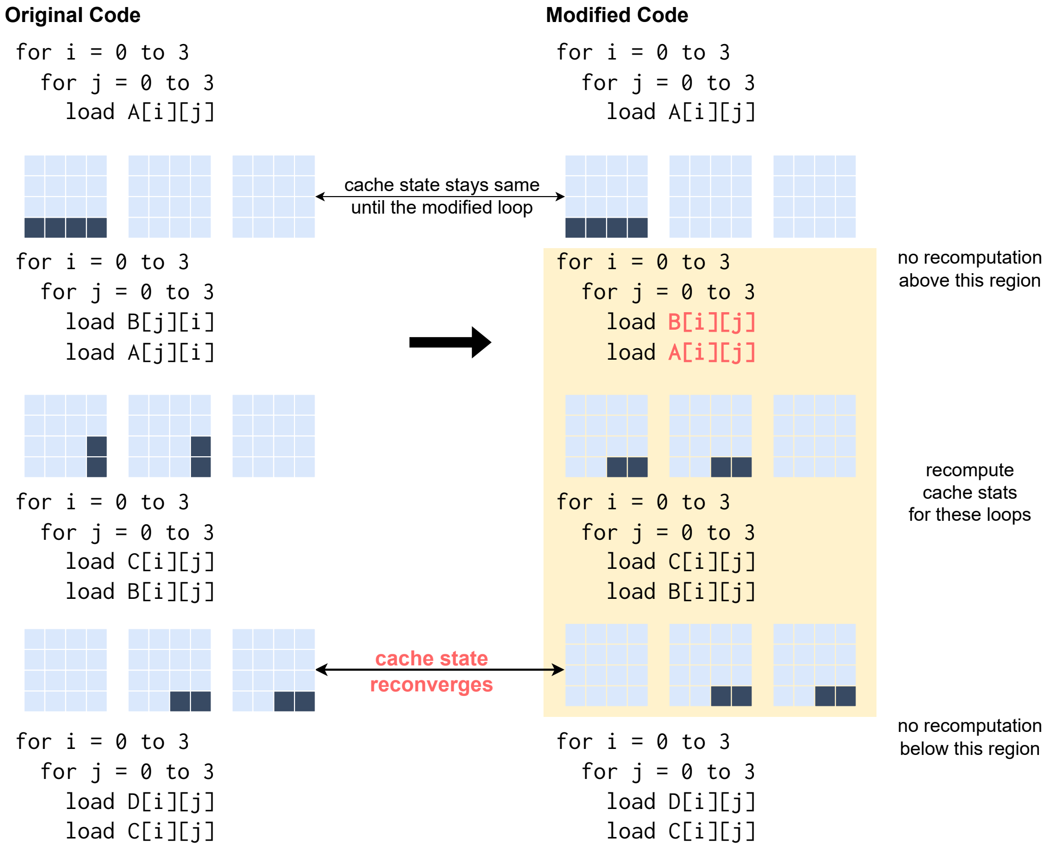


Fig. 3. Example illustrating the incremental recomputation optimization. The grids represent the arrays A, B, and C. The darker squares represent the elements that are still present in the cache after the execution of each code fragment. Cache states remain the same before the modification and are different for some time after the modification. Soon though, the effects of the modification dissipate and the cache state reconverges to that in the original program. In the example, this occurs immediately after the third loop nest is executed.

4.4.3 Incrementally Updating Predictions. Suppose we are trying to optimize our program by making changes to some part of it. Whether it is a human or an autotuner, we typically only change one local region before re-evaluating performance. Our distance-based heuristics allow us to quickly and exactly recompute the whole program's cache miss rate predictions after such a change. The user's editor or the autotuner must mark modified and new statements as "new". In our MLIR implementation, this is denoted by an attribute. A statement can be deleted by modifying it to a noop. Our algorithm will then update the predictions for each statement, by computing the results for all modified statements as well as any statements whose cache behavior is affected by these changes. However, it does not spend time recomputing results for the vast majority of statements in the program that are deduced to definitely not be affected by the changes.

The first optimization is that if any statement is only ever executed *before* any modified code, then its behavior is certainly not affected by the modification. So any statement that lies outside any modified top-level loops and above them, inherits the old predictions.

Next, we avoid excessive computations *below* modified code. This works similarly to the top-level statement heuristic. That heuristic tells us that for each sink statement, there is a specific point in the program above which there is no need to look for dependences as they do not affect the cache miss rate. This means that the rest of the program above this point does not matter at all, since our algorithm never even looks at it. If the sink statement we are analyzing is so far below the last modified top-level loop that the distance heuristic would prevent us from looking at it, then a change there has no impact on the sink, so the sink once again inherits the old miss counts. In other words, this far below the modified region, the cache states reconverge to what they were before the modification (Figure 3). We then inherit the old miss counts for all further statements until we encounter another top-level loop containing a modified statement, after which the process repeats. This optimization reduces the geomean time taken to update predictions by 133.7x (Section 5.4.2).

4.5 Threshold Counting: Counting Cache Misses

Capacity misses correspond to sink instances such that the reuse distance is at least the cache size, i.e., $\#\texttt{reuseLines}(\texttt{Sink}(\mathbf{p})) \geq \texttt{cacheSize}$, where applying the relation `reuseLines` to a domain point $\texttt{Sink}(\mathbf{p})$ produces all lines that that point maps to. We want to count the number of such instances, which is the cardinality $\#\{\texttt{Sink}(\mathbf{p})) \in \mathcal{I}_{\texttt{Sink}} \mid \texttt{reuseDist}(\texttt{Sink}(\mathbf{p})) \geq \texttt{cacheSize}\}$.

For this, we first need to obtain $\texttt{reuseDist}(\texttt{Sink}(\mathbf{p})) = \#\texttt{reuseDist}(\texttt{Sink}(\mathbf{p}))$ as a function in closed form. The parametric version of Barvinok's algorithm [Barvinok 1994; Verdoolaege et al. 2007] produces a closed-form representation of this cardinality as a piece-wise quasi-polynomial, i.e., the domain is partitioned into pieces and for each piece the function is given as a quasi-polynomial. Here a quasi-polynomial is a polynomial involving the variables $\mathbf{p}_i$ and floor divisions of affine expressions in the $\mathbf{p}_i$, where the denominator is constant.

Barvinok's algorithm only works for Presburger sets. If the produced function is linear then the set above is still a Presburger set and we can compute the outer cardinality using Barvinok's algorithm again [Gysi et al. 2019]. If the produced function is not linear, then the above is not a Presburger set, so we perform some simplifications that split the domain of the function into a number of pieces, such that the function restricted to each of these pieces can be expressed as a linear expression.

4.5.1 Symbolic Pair-Wise Enumeration. For each pair of variables in the counting polynomial, say x and y, we compute the minimum and maximum value that $x - y$ can take. This can be done efficiently in practice using linear programming. It is often the case that two variables are always, say, within 8 of each other, due to constraints on the domain. In this case, we can enumerate the possible values of x in terms of y: for each possible integer value c that $x - y$ can take, we substitute $x = y + c$ in the polynomial. For example, if, due to various domain constraints, it turns out that $0 \leq x - y \leq 1$, then the expression $x^2 - y^2$ can be rewritten as two pieces with simplified expressions $y - y = 0$ when $x - y = 0$ and $(y + 1)^2 - y^2 = 2y + 1$ when $x - y = 1$. These are much easier to count than the original multivariate quadratic expression. In our implementation, we execute the splitting if it would generate at most 16 pieces.

4.5.2 Enumerating Divisions. Sometimes divisions in the quasi-polynomials have large divisors and take very few possible values. For example, a division like $\lfloor (x - y)/1024 \rfloor$ where the domain constraints impose that $0 \leq x, y \leq 1500$ can only take seven values, the integers in $[-3, 2]$. As such, enumerating out all possible such values helps reduce the degree at the cost of a relatively minor blowup in the number of pieces. In our implementation, we perform this splitting if at most ten pieces would be generated for the division.

A naive implementation of this in the Presburger solver, isl, would run into problems. This is because isl normalizes all division numerator coefficients to be non-negative by adding or subtracting multiples of e.g. y as needed. The above division would then be written as $\lfloor (x + 1023y)/1024 \rfloor - y$. The division in this expression takes on many possible values for the same domain constraint; for y from 0 to 1500, it produces at least 1024 different values here, even for a fixed value of x. Therefore, the form in which the division is presented matters. For this heuristic to work well, we instead normalize all coefficients to $(-d/2, d/2]$, where d is the denominator of the division. As most of the expressions coming out of Barvinok's algorithm seem to involve small positive or negative coefficients, this modified normalization works well with our heuristic.

4.6 Representing Execution Order in Presburger Arithmetic

Let S and T be memory access statements. We want to express in Presburger arithmetic the condition that the instance $S(\mathbf{p})$ executes before the instance $T(\mathbf{q})$, denoted $S(\mathbf{p}) \prec T(\mathbf{q})$. Recall that in our Affine IR, if-conditions and loops are the only control flow. First, let's consider the case that S lies lexically before T. Then if S and T do not have any loops that surround both of them, all instances of S execute before all instances of T.

Now let's consider the case where there are $k \geq 1$ common loops surrounding S and T. Let's call these loops $L_1, \dots L_k$ in order of increasing depth. If iteration $(\mathbf{p}_1, \dots \mathbf{p}_k)$ of these loops executes before iteration $(\mathbf{q}_1, \dots \mathbf{q}_k)$, then we have $S(\mathbf{p}) \prec T(\mathbf{q})$. This happens iff $(\mathbf{p}_1, \dots \mathbf{p}_k)$ strictly precedes $(\mathbf{p}_1, \dots \mathbf{p}_k)$ in the lexicographic ordering, because in our Affine IR all loops are normalized to iterate in increasing order of induction variable, possibly with some constant stride. The lexicographic comparison can be implemented directly in Presburger arithmetic since it supports ANDs, ORs, as well as imposing equality and inequality conditions on variables. For example, (x, y) lexicographically precedes $(a, b) \iff x < a \vee (x = a \wedge y < b)$.

If $(\mathbf{p}_1, \dots \mathbf{p}_k)$ comes after $(\mathbf{q}_1, \dots \mathbf{q}_k)$ in the lexicographic ordering, then $S(\mathbf{p}) \succ T(\mathbf{q})$. On the other hand, if the two vectors are equal, then the instance of the statement that comes earlier in the *lexical* order executes first. Thus, we define $\prec$ as a Presburger relation over all instances.

4.7 Computing `reuseInstances` efficiently

We compute a relation

$$\begin{aligned} \texttt{between} : \mathcal{I} \times \mathcal{I} &\to \mathcal{I} \\ &= \{(S(\mathbf{p}), T(\mathbf{q})) \to U(\mathbf{r}) \mid S(\mathbf{p}) \prec U(\mathbf{r}) \prec T(\mathbf{q})\}. \end{aligned}$$

This maps any two instances $S(\mathbf{p}) \prec T(\mathbf{q})$ to all other instances that execute between them. For brevity, we refer to $\texttt{deps}_{\texttt{Sink}}$ as $\texttt{deps}$ here. We compute $(\texttt{deps} \times \mathcal{I}) \cap \texttt{between}$. This can be understood as

$$\big\{(\texttt{Sink}(\mathbf{p}), T(\mathbf{q})) \to U(\mathbf{r})) \mid T(\mathbf{q}) = \texttt{deps}(\texttt{Sink}(\mathbf{p})) \wedge \\ \texttt{Sink}(\mathbf{p}) \prec U(\mathbf{r}) \prec T(\mathbf{q})\big\}.$$

Projecting out $T(\mathbf{q})$ from this, we obtain

$$\big\{\texttt{Sink}(\mathbf{p}) \to U(\mathbf{r})) \mid \texttt{Sink}(\mathbf{p}) \prec U(\mathbf{r}) \prec \texttt{deps}(\texttt{Sink}(\mathbf{p}))\big\},$$

which is the set of instances executed between each statement instance $S(\mathbf{p})$ and its reuse source. In other words, this is `reuseInstances`. From the equation, we see that the only relevant parts of the $\prec$ relation are the ones involving `Sink` and statements that are dependences of `Sink`. Hence, we only write down these relevant constraints when inlining the definition of $\prec$ in the above description. Earlier approaches converted the whole program into a single big formula and would explicitly construct $\prec$ as a relation over all instances of all statements in the program, which blows up the number of constraints. By exploiting the structure of the input program throughout the algorithm we avoid this combinatorial explosion.

4.8 Supporting Cache Lines

When there is exactly one element per cache line, the dependence algorithm (Section 4.3) directly gives us the reuse source. When there are multiple elements in a cache line, we need to modify the relations mapping statement instances to the location they access, so that they instead map to the cache line they access. We can then run the same dependence algorithm on this modified access map to obtain the reuse sources.

One approach is to linearize all the array accesses, flatten all the arrays to 1D, and floor-divide by the cache line size. Thus, for a 2D array of dimension $M \times N$, $A[x][y]$ hits the cache line indexed $\lfloor (o + Nx + y)/B \rfloor$, where B is the number of array elements per cache line, which is a positive integer for typical datatypes and cache line sizes, and o is the offset of the beginning of the array A.

Unfortunately, this produces a much more complex access expression than the individual expressions for each dimension in the array, which hampers analysis performance. One workaround [Gysi et al. 2019] is to perform approximate modeling, by assuming o and the last array dimension size are multiples of B. In this case, all rows in the array start at the beginning of a cache line and we don't need to linearize. In the example, N is padded to the next multiple of B and the accessed cache line is defined by the tuple $(x, y/B)$; o becomes irrelevant when it is a multiple of B. However, this approximation results in a significant accuracy loss in some cases where the last dimension is small, which is common in some classes of neural networks.

We introduce the *partial linearization* feature, which incorporates aspects from both these approaches. When this feature is enabled, we combine an array's last k dimensions into a single one, linearizing the accesses to these dimensions into the single combined dimension. We then model the program as if this combined last dimension were padded to cache line size. This avoids issues caused by padding small dimensions. This makes involved expressions somewhat more complex, incurring some slowdown, but our model is fast enough to begin with that we can still obtain results in less than four minutes on average (Section 5.4). In our implementation, we choose k for each array such that the combined dimension size is at least 10.

4.9 Generalizing to Multi-Level Hierarchies

Real-world caches typically have multiple levels. We support exactly modeling inclusive and exclusive multi-level cache hierarchies with a write-through write-allocate policy. In this policy, all writes load the cache line into the cache and update it there. They then also write the value to the backing store. In this case, we compute the number of read misses in L1 by running the model as normal, treating writes the same as reads. Since every write loads its cache line and "uses" it (in the sense of LRU), everything works if we just include write accesses in the reuse instances relation. The number of fetches from the backing store due to writes is also computed in the same way that read misses are computed. The number of writes performed is the same for each level of the cache – it is equal to the total number of writes, which is easily computed.

The last thing left to compute is the number of read misses at L2, which is done as follows. Let L1 and L2 have C_1 and C_2 cache lines respectively. In inclusive caches, when a new cache line is loaded, it is loaded into both L1 and L2 caches. When a cache line is evicted from L2, it is evicted from L1 as well. However, in LRU caches a cache line that is being evicted from L2 would never be present in L1. This is because under LRU the set of cache lines in L1 is equal to the C_1 most recently used cache lines in L2. The line being evicted from L2 is always the C_2th most recently used and thus is not part of L1. L2 is essentially independent of L1 here; running a single-level C_2-sized cache automatically simulates the included L1 cache in its C_1 most recently used lines. The L2 miss count is thus the miss count of a single-level C_2-sized cache. Ye et al. [2017] show that

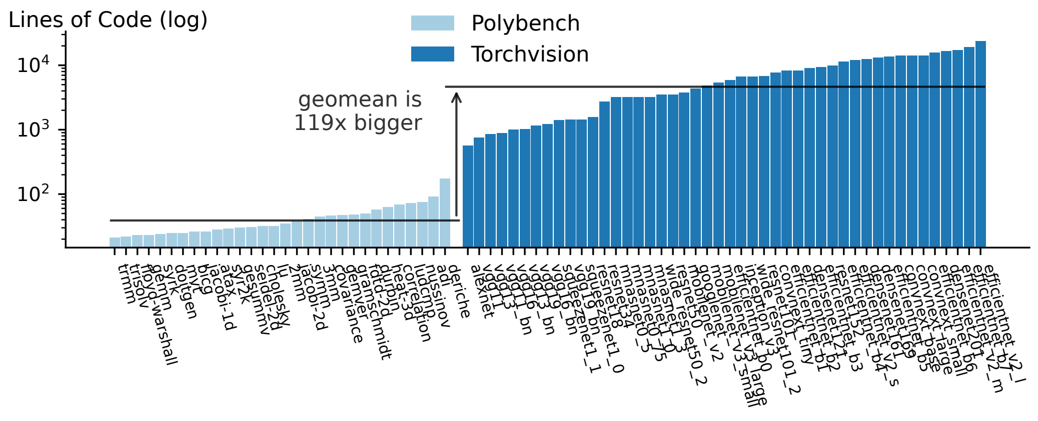


Fig. 4. We model the established but relatively small Polybench kernels (at most 175 lines) and, ranging from 566 to 23,595 lines, the considerably larger TorchVision benchmarks.

we can model exclusive cache hierarchies, by running once with size C_1 and once with size $C_1 + C_2$, obtaining miss counts for L1 and L2 respectively.

We do not need to run the whole cache model for L1 and L2 separately. The reuse distance is the same irrespective of the cache size; only the threshold counting depends on the cache size. However, our distance-based optimizations (Section 4.4) depend on the cache size, so we need to decide what size of cache they should use when computing reuse distance expressions to be used in multiple threshold counts. When pruning sure *misses*, we use the size of the largest cache in the hierarchy to ensure that we only consider sure misses that miss even in the biggest cache considered. When pruning sure *hits* (Section 4.4.1), we use the size of the smallest cache in the hierarchy, to ensure that we only consider sure hits that hit even in the smallest cache.

5 EVALUATION

We evaluate our tool on two benchmark sets: a collection of deep neural networks and a well-established set of loop kernels. We evaluate our ability to scale by evaluating on 46 TorchVision networks (inference mode). In addition, we evaluate on Polybench [Pouchet 2012], a set of 30 loop kernels previously used in the cache modeling literature [Gysi et al. 2019; Morelli and Reineke 2022; Shah et al. 2022]. We compare our model's performance against the state-of-the-art existing models, Haystack [Gysi et al. 2019] and Warping [Morelli and Reineke 2022]. The TorchVision networks are two orders of magnitude larger than the Polybench kernels on average (Figure 4), so good performance on this benchmark indicates scalability.

On the TorchVision networks, our single-threaded performance is at least 40x faster than the state-of-the-art models HayStack and Warping. While Haystack's geomean runtime for it is over half an hour, Falcon's geomean runtime is just 1 minute and 8 seconds, and even on the slowest benchmark Falcon takes less than 90 seconds. We are also at least twice as fast as Haystack and Warping on Polybench. Running in parallel on 16 cores yields an additional 8.2x geomean speedup on TorchVision. Updating the prediction after a local change to the program takes just 513 ms on average (geomean).

Our implementation returns the same results as Haystack on Polybench and on all programs in the TorchVision benchmark where Haystack terminates within our 4-hour timeout. With our partial linearization technique, our Pearson correlation of $R = 0.98$ with hardware measurements for TorchVision is close to a perfect correlation ($R = 1$).

5.1 Benchmarks

We evaluate Falcon on a benchmark consisting of neural networks from TorchVision, including popular architectures such as AlexNet [Krizhevsky et al. 2012], ConvNext [Liu et al. 2022], GoogLeNet [Szegedy et al. 2015], Inception v3 [Szegedy et al. 2016], MobileNet [Howard et al. 2017], ResNet [He et al. 2016], VGG [Simonyan and Zisserman 2015]. To run our model, we lower the programs to the structured Affine IR (Section 2.1) using an existing third-party front-end, Torch-MLIR[2]. Six of the architectures were not supported by Torch-MLIR and had to be excluded. For each architecture, we take all versions that are available on TorchVision.

```
%zero = arith.constant 0.000000e+00 : f32
%A = memref.alloc() {alignment = 64 : i64} : memref<64x128xf32>
affine.for %i = 0 to 64 {
   affine.for %j = 0 to 128 {
     affine.store %zero, %A[%i, %j] : memref<64x128xf32>
   }
}
```

(a) Original MLIR Affine IR.

```
volatile int A[64][128];
for (int i = 0; i <= 63; i += 1)
  for (int j = 0; j <= 127; j += 1)
    A[i][j] = 0;
```

(b) Converted C code.

Fig. 5. Example conversion from MLIR Affine IR to C for running baseline models.

To run the baseline cache models, we convert the programs to a C representation (Figure 5). The C representation is obtained by creating a C program that accesses the same memory locations as the MLIR Affine IR. Since prior models only operate on the memory access trace of the given program, ignoring scalar accesses, running them on this C representation is equivalent to running them on the program with compute operations.

In the TorchVision benchmark, all arrays always have datatype `f32` (32-bit float), alignment to 64 bytes whenever that parameter is present, and contain an even number of elements. Due to the even element counts, 64-byte alignment occurs by default. Due to the datatype always being `f32`, we can always use 32-bit integer in the converted C code (only datatype width matters for cache models). Finally, the `volatile` keyword does not have any effect on cache modeling but prevents accesses being optimized out when compiling for hardware measurement of cache misses.

Prior works like Haystack and Warping primarily evaluated their work on Polybench [Pouchet 2012]. We also evaluate on this to show that we are still competitive on this benchmark which has been traditionally used in this area of work. We run Haystack and Warping on Polybench's C benchmark. We then raise the C benchmark to MLIR Affine IR using Polygeist [Moses et al. 2021] and run Falcon on the raised representation.

[2]https://github.com/llvm/torch-mlir

5.2 Methodology

Our development machine has an AMD Ryzen 9 5950X 16-core system with 64 GB of RAM. Each CPU core has a 32 KiB L1 data cache and 512 KiB L2 cache (inclusive of L1). Each 8-core complex has a 32 MiB L3 cache (exclusive of L1 and L2). All caches use a write-allocate write-back policy. The L1 and L2 caches are 8-way while the L3 cache is 16-way set associative.

Our experiments model the L1 and L2 cache hierarchy of this system. We approximate its undocumented replacement policy as LRU and model fully associative versions of these caches. We are able to show that this is sufficient to achieve very good correlation with hardware measurements when the partial linearization feature is enabled.

We run all models with a 4-hour time limit as the baselines that we compare against sometimes run for a long time. Even within this time limit, Haystack sometimes exhausts the 64 GB RAM of the system and crashes. Because of this, all experiments were *run* on another machine, having an Intel(R) Xeon(R) Gold 6226 CPU and 187 GiB of memory.

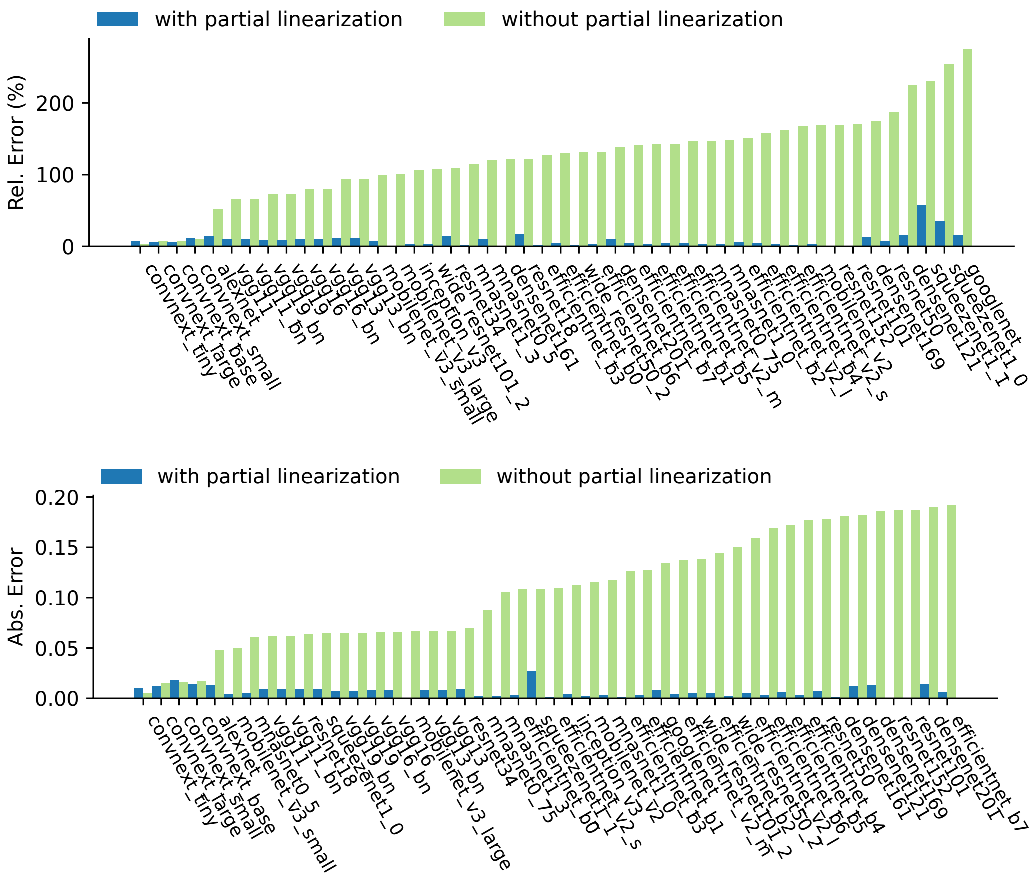


Fig. 6. Enabling partial linearization reduces the mean relative error in predicting L1 miss rates from 124.53 to 8.53, and the mean *absolute* error from 0.11 to 0.01. (The absolute error can be at most 1 in the worst case.)

5.3 Accuracy

We find that our model without partial linearization produces outputs that exactly match Haystack on all 120 Polybench kernels and on all the TorchVision models where it terminated in time. This makes it especially interesting to compare the accuracy of Falcon with and without partial linearization. We check accuracy by comparing the predicted miss rates with hardware measurements performed using the `perf_event_open` syscall. This syscall gets information from the hardware performance monitoring unit (PMU). Partial linearization brings down the mean absolute prediction error from 0.11 to 0.01 and the relative error from 124.53% to 8.53% (Figure 6). Finally, enabling partial linearization takes the Pearson correlation of our miss rate predictions from 0.56 to 0.98 (Figure 7), which is close to the optimal value of 1.

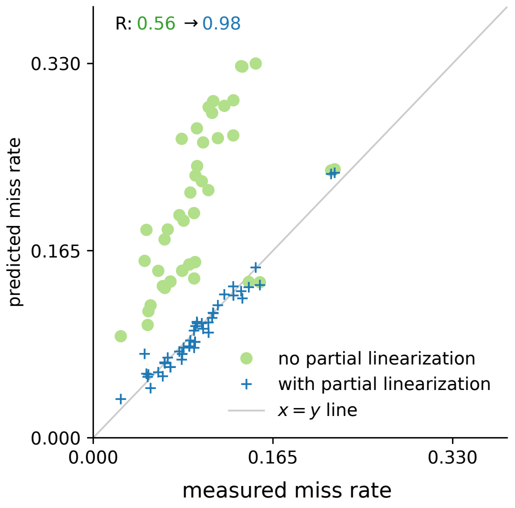


Fig. 7. Partial linearization makes our model much more accurate: the Pearson correlation R for L1 miss rate predictions goes from 0.56 to 0.98 (1 being perfect correlation).

The accuracy of these predictions indicates that few conflict misses occur in this workload. This matches the findings of prior work such as Haystack [Gysi et al. 2019], which showed that out of thirty different kernels in the Polybench benchmark, only one (doitgen) showed a significant difference between the fully associative and set-associative cache performance.

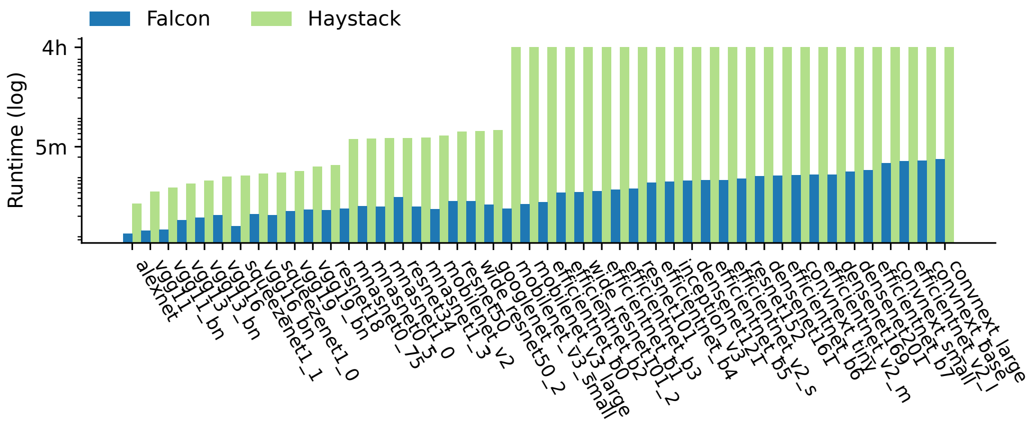


Fig. 9. Falcon takes seconds to minutes to model the programs in the TorchVision benchmark whereas Haystack is slower or times out.

5.4 Performance

We compare the performance of our model against Haystack and Warping. We perform all comparisons on equal terms by disabling partial linearization, and separately report the performance impact of partial linearization. For programs in the TorchVision benchmark, Falcon runs in seconds to minutes whereas Haystack often times out after running for four hours (Figure 9). Warping always times out.

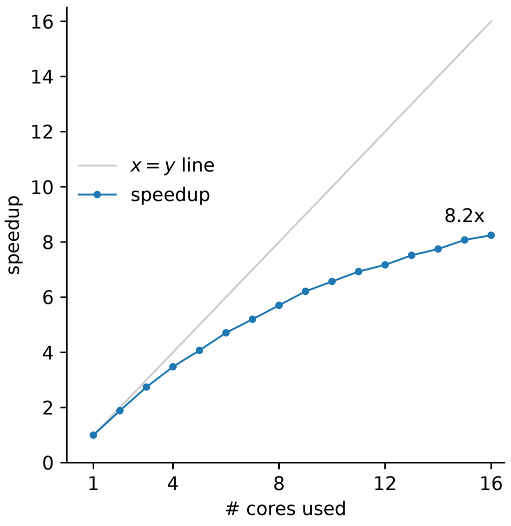


Fig. 8. Running on 16 cores gives Falcon an 8.2x geomean speedup on the TorchVision benchmark over single-threaded runs.

Finally, we compare with Polybench (XL), where XL refers to larger array sizes, loop trip counts, and number of memory accesses, but not more program statements. We find Falcon has a geomean runtime of 952 ms, as compared to 2.06 s for Haystack, and at least 4 minutes for Warping. (Warping sometimes hits the four-hour time limit, so the true average would be higher.) Thus, we are not only much faster on the large programs in the TorchVision benchmark but also competitive on the smaller benchmarks that the previous cache models were evaluated on.

Enabling the partial linearization method results in a 5.05x slowdown in our model's runtime, bringing the geomean runtime to 3 minutes and 46 seconds, which is still significantly faster than the baselines, both of whose geomean runtimes are greater than 32 minutes.

5.4.1 Parallelism. Our algorithm is embarrassingly parallel across sinks. Thus, we obtain notable speedups with increasing parallelism (Figure 8). This would be useful in latency-sensitive situations, such as a cache performance LSP that gives feedback to a performance programmer doing manual performance tuning. Using 16 cores, we obtain an 8.2x speedup over our single-threaded runtime. The maximum attainable speedup here mostly depends on the length of the longest-running thread, i.e., the time taken to predict the performance of the single most difficult program statement.

5.4.2 Incrementally Updating Predictions. We evaluate the performance of our algorithm to incrementally update predictions after local changes (Section 4.4.3). The evaluation methodology here must be chosen with some care. If the local modifications we perform add huge amounts of code to the program, or even highly complex code that is difficult to model, then this will be the overwhelming factor determining the runtime of the incremental compute. On the other hand, if we evaluate by always simplifying a local section of code, then that would make incrementally updating look especially performant.

A fair evaluation will check how much faster incrementally updating is than a full recomputation of the analysis, all else being equal – in particular, with the size and complexity of the code not changing significantly. To this end, we evaluate by marking a statement as being modified without changing the content of the statement. We run the incremental update algorithm once for each statement in the program, by marking that statement as modified.

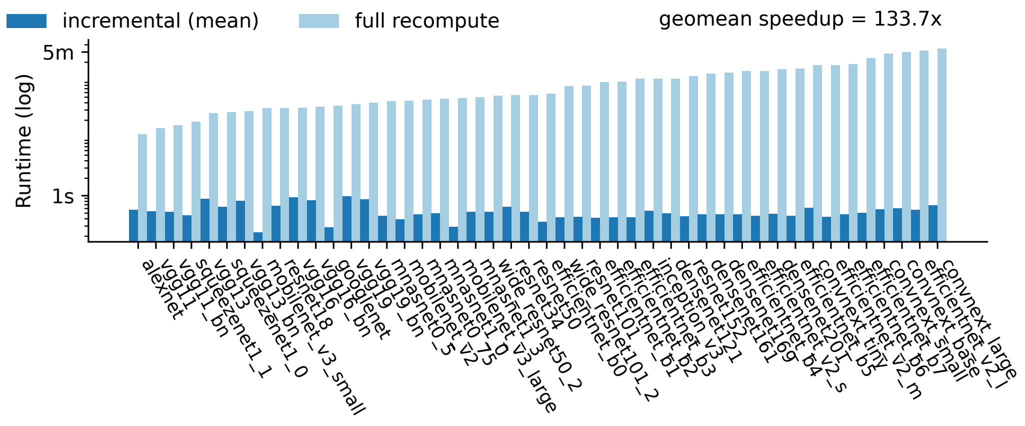


Fig. 10. Incremental updates after local changes take Falcon less than a second instead of seconds to minutes for the full analysis.

Taking an average over the statements in each program gives a good estimate of how local the impact of the average statement in that program is, and how much our algorithm can exploit this locality to cut down on recomputation time. We find that the mean incremental update takes less than a second (Figure 10), as compared to tens of seconds to minutes for a full recomputation.

6 RELATED WORK

Simulators. Cache simulators like Dinero [Edler and Hill 1999] and CASPER [Iyer 2003] can precisely model the cache misses for a variety of real-world cache policies. However, they do so by explicitly iterating through a trace of all the memory accesses in the program and so scale poorly.

Hybrid approach. Warping [Morelli and Reineke 2022] runs a simulation and tries to fast-forward with polyhedral techniques whenever possible. Like a simulator, it can support a variety of cache configurations. However, it turns out to be slow for the large TorchVision programs.

Analytical models. Analytical cache models try to provide a more scalable solution. There is a long history of work on these [Bao et al. 2017; Beyls and D'Hollander 2005; Chatterjee et al. 2001; Gysi et al. 2019]. Chatterjee et al. [2001] developed the first analytical cache model for arbitrary affine programs by describing the set of cache misses by a Presburger formula. Rather than computing the reuse sources for a statement instance $\mathcal{S}(\mathbf{p})$ accessing a cache line L in an A-way associative cache explicitly, they write down a formula with $A + 1$ existentially quantified variables corresponding to statement instances accessing unique cache lines that are not L, with the constraint that there should be no access to L between these and $\mathcal{S}(\mathbf{p})$. If this formula is satisfiable for a particular $\mathbf{p}$, then the reuse distance for $\mathcal{S}(\mathbf{p})$ exceeds the associativity and that instance incurs a miss. Thus, counting the number of solutions to this formula gives the number of misses. This approach, however, does not scale to larger associativities as it produces high-dimensional formulas that are hard to solve.

Beyls and D'Hollander [2005] explicitly calculate the most recent access to a location with a formula similar to that of Chatterjee et al. [2001]: they consider every pair of statements and compute the set of pairs of statement instances such that both references access the same location and there do not exist any intervening accesses to the same location. They use this to compute the reuse lines relation and compute the parametric cardinality to obtain the reuse distance polynomial. They noted that the miss count is the number of **p**'s such that this polynomial exceeds the associativity, but did not present an approach to actually compute this. Instead, they use the calculated polynomial at runtime to insert cache prefetch instructions. We focus on statically and efficiently estimating cache performance. They were also the ones that introduced the concepts of *backward* and *forward* data reuses; the notion of *reuse source* that we use is the source of the backward reuse of a statement instance (if it exists).

Bao et al. [2017] instead use parametric integer programming (PIP) [Feautrier 1988] to directly compute the most recent access. Their approach runs within tens of seconds for most Polybench kernels on the standard problem size. They also use an approach based on the reuse lines relation , but solve the threshold counting problem differently. Instead of computing the parametric cardinality, they once again exploit PIP. The goal is to find at least *associativity* unique *parametric points* in the intervening access relation. PIP can provide the domain of a relation as well as a parametric point, mapping each domain element to a point in its image. Thus, by repeatedly finding a parametric point and subtracting it from the relation $associativity - 1$ times, they obtain a relation whose domain only contains instances that map to at least A accesses. Computing the cardinality of this domain gives the number of misses. They also developed extensions to support multi-level caches, and a similar repeated-PIP approach to compute the set of cache lines in the cache at the end of a program fragment. These repeated-PIP methods are likely to blow up at larger associativities, as PIP sometimes generates a large number of disjuncts and subtracting by many disjuncts typically produces even more disjuncts, and their approach repeats this whole process *associativity* times.

Haystack [Gysi et al. 2019] introduced a fully associative model that can handle larger problem sizes by computing the number of cache misses from the polynomial-based formulation introduced by Beyls and D'Hollander [2005]. Their contribution is to solve the threshold counting problem efficiently in practice. When the polynomial is affine, they use another execution of Barvinok's algorithm to compute the number of misses. They also introduced two division simplification techniques to try and reduce the degree of the polynomial. Finally, they introduced partial enumeration to handle the case when the polynomial cannot be made affine. Their approach was the first of these models robust enough to compute all Polybench results on larger array sizes within a minute. Their model assumes that the last dimension of all arrays is a multiple of the cache line size; if the input does not satisfy this, it operates as though the last dimension had been padded to satisfy this constraint. While this did not significantly hamper accuracy on Polybench, it does cause issues in the TorchVision benchmark. In this case, the last dimension often refers to the number of channels, which could be three for an RGB input image. Here, we significantly improve accuracy by introducing partial linearization (Section 4.8).

BullsEye [Shah et al. 2022] further improved on Haystack by contributing novel probabilistic and approximate methods to perform threshold counting. Our main contributions are orthogonal and complementary to those of Haystack and BullsEye. These existing works as well as future analytical models can be plugged into our framework, which will improve our performance at threshold counting. In turn, our work can produce simpler reuse distance polynomials, which these threshold counting methods would have an easier time counting.

Polyhedral cache modeling has thus far mostly focused on single-kernel benchmarks. Ours is the first model that scales to modeling cache hierarchies on the actual full programs we would like to optimize today. As we have seen, all previous approaches have operated on the whole program represented as abstract Presburger formulas and did not exploit the program structure at all.

The threshold counting problem that Gysi et al. [2019] and Shah et al. [2022] focus on is not currently the bottleneck when it comes to such large programs. Rather, it's the reuse distance computation. Our approach, for the first time, exploits the program AST structure, allowing us to optimize the reuse distance computation. As a result, we introduce the first analytical cache model that scales to large programs and can efficiently update results after local modifications to the program.

Estimation by random sampling. Some works estimate cache performance by random sampling. One disadvantage of such approaches is that compilers that use them to guide optimization do not produce deterministic artifacts. Chen et al. [2018a] generate a random sample of instances of each statement and compute the reuse *time* for each sampled instance, which is the number of total accesses (not necessarily unique) performed between the instance and its reuse source; this is then used to estimate cache miss rates. Their tool generates a program-specific binary for each input program and samples a constant fraction of instances from each statement. Compilation may be quick for the Polybench programs they evaluate on, but for larger programs this time can be substantial, especially in an autotuning setting. Our incremental recomputation feature could be combined with this to pick an appropriate fragment of the program that is sufficient to capture the cache behavior of a given modified statement. They report a speedup of 20.97x over tracing. In comparison, Haystack reports a speedup of 370x over the Dinero [Edler and Hill 1999] simulator, while Falcon is in turn much faster than Haystack.

Xue and Vera [2004] explore approximate and randomized approaches. They start by defining the reuse source using a Presburger formula. When trying to find the most recent prior statement instance of a statement S1 that accesses the same cache line as a specific instance of a statement S2, they use an approximation to avoid the usage of Parametric Integer Programming in cases where the two statements have index expressions that, for each array dimension, only differ from each other by a constant, but have the same coefficients for all induction variables. This approximation is exact if the array is at most 2-dimensional or if the lowest dimension's size is a multiple of the cache line size. These assumptions do not hold in the TorchVision benchmark, where most arrays have more than two dimensions and a small last dimension. In our work, we show that partial linearization obtains a significant accuracy boost over predictions made under the cache line padding assumption. Their sampling approach can be plugged in as a replacement for threshold counting in our tool as well. Finally, our approach enables fast incremental recomputation which they do not support.

Polyhedral dependence algorithms. Our algorithm primarily operates on the AST representation of the program, unlike prior cache models. For this, we use a dependence algorithm based on the implementation in isl, which is in turn based on the algorithm given by Maslov [1994]. This algorithm is generic; one of our other contributions is the observation that this generic algorithm works well for the cache modeling use case. On top of this, we use several domain-specific optimizations to accelerate the model on large-scale programs and support efficiently updating the results after program modifications.

7 CONCLUSION

We presented Falcon, the first analytical cache model that scales to large-scale programs like neural networks. This is accomplished by taking advantage of the program AST's control-flow structure instead of operating on programs abstractly represented by Presburger formulas. Our model runs in 44.9 seconds on average on our neural network benchmark as compared to over 32 minutes for the prior state-of-the-art. Falcon updates predictions after local modifications in 513 ms on average. Thus, we provide a scalable, accurate, and efficiently updateable analytical cache model.

DATA-AVAILABILITY STATEMENT

We have made available an artifact [Pitchanathan et al. 2024] that includes a Docker image with the necessary toolchains, benchmarks, sources, and scripts to reproduce the main results from our evaluation (Section 5).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This project has received funding from the European Union's Horizon EUROPE research and innovation program under grant agreement no. 101070374 (CONVOLVE).

REFERENCES

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2017. Analytical Modeling of Cache Behavior for Affine Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 32 (dec 2017), 26 pages. https://doi.org/10.1145/3158120

Alexander I. Barvinok. 1994. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension Is Fixed. *Mathematics of Operations Research* 19, 4 (1994), 769–779. http://www.jstor.org/stable/3690312

Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250. https://doi.org/10.1016/j.sysarc.2004.09.004

Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. 2001. Exact Analysis of the Cache Behavior of Nested Loops. *SIGPLAN Not.* 36, 5 (may 2001), 286–297. https://doi.org/10.1145/381694.378859

Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018a. Locality analysis through static parallel sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 557–570. https://doi.org/10.1145/3192366.3192402

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://dl.acm.org/doi/10.5555/3327144.3327258

Jan Edler and Mark D. Hill. 1999. Dinero IV Trace-Driven Uniprocessor Cache Simulator. (1999). https://pages.cs.wisc.edu/~markhill/DineroIV/

Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268. https://doi.org/10.1051/ro/1988220302431

Michael J. Fischer and Michael O. Rabin. 1998. Super-Exponential Complexity of Presburger Arithmetic. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Bob F. Caviness and Jeremy R. Johnson (Eds.). Springer Vienna, Vienna, 122–135. https://link.springer.com/chapter/10.1007/978-3-7091-9459-1_5

GCC Contributors. [n. d.]. Auto-Vectorization in GCC. ([n. d.]). https://gcc.gnu.org/projects/tree-ssa/vectorization.html Accessed: 2023-11-12.

Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 816–829. https://doi.org/10.1145/3314221.3314606

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

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

Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/300.pdf

Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://arxiv.org/abs/2008.01040

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105. https://dl.acm.org/doi/10.5555/2999134.2999257

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*. IEEE Press, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Z. Liu, H. Mao, C. Wu, C. Feichtenhofer, T. Darrell, and S. Xie. 2022. A ConvNet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 11966–11976. https://doi.org/10.1109/CVPR52688.2022.01167

LLVM Contributors. [n. d.]. Auto-Vectorization in LLVM. ([n. d.]). https://llvm.org/docs/Vectorizers.html Accessed: 2023-11-12.

László Lovász and Herbert E. Scarf. 1992. The Generalized Basis Reduction Algorithm. *Mathematics of Operations Research* 17, 3 (1992), 751–764. http://www.jstor.org/stable/3689761

Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia (MM '10)*. Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

Vadim Maslov. 1994. Lazy Array Data-Flow Dependence Analysis. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. Association for Computing Machinery, New York, NY, USA, 311–325. https://doi.org/10.1145/174675.177911

Canberk Morelli and Jan Reineke. 2022. Warping Cache Simulation of Polyhedral Programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/10.1145/3519939.3523714

William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

David A. Patterson and John L. Hennessy. 2013. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Arjun Pitchanathan, Kunwar Grover, and Tobias Grosser. 2024. Artifact for "Falcon: A Scalable Analytical Cache Model". (2024). https://doi.org/10.5281/zenodo.10972076

Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. 2021. FPL: Fast Presburger Arithmetic through Transprecision. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 162 (oct 2021), 26 pages. https://doi.org/10.1145/3485539

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

Nilesh Rajendra Shah, Ashitabh Misra, Antoine Miné, Rakesh Venkat, and Ramakrishna Upadrasta. 2022. BullsEye: Scalable and Accurate Approximation Framework for Cache Miss Calculation. *ACM Trans. Archit. Code Optim.* 20, 1, Article 2 (nov 2022), 28 pages. https://doi.org/10.1145/3558003

Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2014. Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, 287–298. https://doi.org/10.1109/SC.2014.29

Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.1556

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

Sven Verdoolaege. 2007. `barvinok` library. (2007). https://barvinok.sourceforge.io/

Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *ICMS*, Vol. 6327. 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

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

J. Xue and X. Vera. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput.* 53, 5 (2004), 547–566. https://doi.org/10.1109/TC.2004.1275296

Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache Exclusivity and Sharing: Theory and Optimization. *ACM Trans. Archit. Code Optim.* 14, 4, Article 34 (nov 2017), 26 pages. https://doi.org/10.1145/3134437

Received 2023-11-16; accepted 2024-03-31