SARCASM: Set-Associative Rotating Cache Analytical/Simulating Model

Guillaume Iooss, Christophe Guillon, Fabrice Rastello, Albert Cohen, Saday Sadayappan

SARCASM: Set-Associative Rotating Cache Analytical/Simulating Model

Guillaume Iooss* Christophe Guillon† Fabrice Rastello‡
Albert Cohen§ Saday Sadayappan¶

December 2024

Abstract

Optimizing compilation for data cache memories is made difficult by (1) the need to statically analyze the program behavior, (2) the complexity of cache hierarchies and policies, and (3) the intricate interplay with a superscalar microarchitecture. Analytical cache modeling typically focuses on the first two challenges, approximating away instruction scheduling and speculation (including data prefetching). While much progress has been made on such models, their practical usage remains very limited. We address this specific gap between analysis accuracy and effective performance. We analyze how effective cache models can be to select profitable optimizations, and study the sensitivity of these optimizations to cache modeling accuracy.

We study a representative range of tensor algebra applications, where all kernels have already been systematically vectorized and optimized for instruction-level parallelism at the register level. We consider four cache models including two analytical ones: (i) a state-of-the-art fully-associative analytical model; (ii) a novel approximate set-associative analytical cache model; (iii) a state-of-the-art cache simulator; and (iv) direct measurement of hardware counters.

We show that, while fully associative models are currently the only scalable ones available to compiler passes, they are not well correlated with performance. On the contrary, supporting some level of set associativity yields much better results, on par with the most accurate (dynamic) simulated and measured cache misses.

*Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, guillaume.iooss@inria.fr
†Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, christophe.guillon@inria.fr
‡Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, fabrice.rastello@inria.fr
§Google DeepMind, albertcohen@google.com
¶University of Utah, saday@cs.utah.edu

1 Introduction

When optimizing a program, a programmer has to choose the optimizing transformation to be applied, and in which order. Between the nature of the transformation, their ordering and the additional required parameters (e.g., a tile size), the choices are numerous, and choosing poorly even once might deteriorate the performance. Thus, the goal is to find (automatically, for an optimizing compiler) optimization strategies that at best find the best-performing sequence of transformation (called a *configuration*), or at least, isolate the good-performing configurations, inside the complex *search space* of valid implementations.

One commonly accepted intuition is that a strategy should reflect the memory hierarchy of the hardware it is mapped on. For example, if we consider BLIS [24] for matrix multiplication on CPU, their optimization strategy uses a register tile (called *microkernel*) whose footprint fits inside the number of vector register of their considered architecture. Then, the loop levels above this microkernel each correspond and saturate a different cache level.

In this paper, we focus on *tensor operations*. This class of program includes important kernels such as matrix multiplication, convolution and tensor contractions, which are of interest in many scientific domains, such as machine learning, linear algebra, or computational chemistry. These kernels have a single statement, a rectangular iteration space and manipulate multidimensional rectangular arrays (called *tensors*). This makes them simple to manipulate and analyze. For example, these kernels are fully tillable, and fully permutable, which means that we can use rectangular tiling and permute the resulting loops in any order while preserving correctness.

We consider the optimization of such programs in sequential and on CPU. Previous works [5, 24, 23] consider this problem while only focusing on the register level. We build on top of these works by assuming that an optimized microkernel has already been picked, thus that the register level performance bottlenecks are already managed. We now focus on the loop levels above the microkernel, that interacts mostly with the multiple levels of cache memories.

Optimizing at the cache level When optimizing for cache, a common tactic is to minimize the amount of cache misses, to minimize the bandwidth above the cache, and the amount of potential stalls that they could provoke. Minimizing the number of cache misses is equivalent to maximizing the reuse of the data that was stored inside a cache. The *operational intensity* [25] is a commonly used optimization metric. It is the ratio between the total amount of computation, over the number of communication required to perform it, i.e. the number of cache misses.

However, estimating precisely the amount of cache misses of a program is

a difficult problem. Indeed, modern caches implement complex and undocumented policies, and their behavior can be also impacted by other chaotic parts of the CPU, such as hardware prefetching or branch prediction. Several approaches can be used to estimate the operational intensity of a kernel:

- **Analytical modeling:** One approach is to model analytically the cache behavior and its state, in order to predict its number of cache misses. Due to the complexity of the task, approximations have to be used to make the problem trackable. The more simplifying hypothesis is made in the model, the less time its analysis will take, but the more imprecise the prediction will be. A common approximation is about the associativity of the cache: assuming that a cache is fully associative [26, 18, 6, 20, 16] allows the model to compare directly the cardinality of the *memory footprint* of a tile with the cache size, to detect if capacity misses occur. In the opposite, models that consider cache sets [22, 4, 2, 10, 1] are more complex and takes more time.

- **Simulation:** Another approach is to simulate the behavior of a cache step by step [3, 11]. The resulting prediction corresponds to our best algorithmic understanding of a cache behavior, and does not require approximations in its modeling. However, this approach is very slow, and is several order of magnitude slower than simply executing and measuring the number of cache misses of a program.

- **Hardware counters:** The last approach consists in actually executing the kernel we wish to optimize and monitor its behavior using hardware counters. The advantage of this approach is that it provides the "true" value. However, this approach can be slow if the program size is large. Also, it requires a complex iterative compiler infrastructure that allows to combine the compilation with native (partial) execution.

Contributions This paper aims at evaluating the usefulness and efficiency of the analytical modeling approaches, to find the best performing configurations. For this purpose, we assume that our cache-level optimization problem can be decoupled to the register-level one by restricting the innermost level of our loop nests to use preselected optimized micro-kernels. With this assumption, we wish to answer the following questions:

- How well does the operational intensity metric correlate to the performance inside such a search space?

- How useful are the analytical cache model predictions when we use them to identify good performing configurations? In particular, how impactful are the simplifying hypothesis used inside these models, such as full associativity?

We study these questions in the context of convolution and matrix multiplication kernels, for commonly found problem sizes, and on two CPUs (one Intel and one ARM). For each of these problem sizes and machines, we randomly draw 1000 different configurations and we measure their performance. We also obtain multiple L2 cache miss estimations from 4 different sources (in increasing order of precision):

1. *Fully associative analytical model*, based on the hypothesis made by the IOOpt [15] tool. This model heavily relies on the notion of *memory footprint*.

2. This previous model was adapted to create a novel *set-associative analytical model*. This model relies on the notion of *detailed footprint* that tracks how many cache line are mapped to each cache sets, then apply the previous fully associative model on each of them. However, when computing the detailed footprint, it approximates the behavior of the cache by assuming that the distribution of the cache lines of the first iteration of any loop is representative of the rest of the computation. This model is also specialized for tensor operations.

3. *Simulation*, using the Dinero cache simulator [3], configured with a pseudo-LRU replacement policy.

4. *Hardware counters measurement*, using PAPI [14].

From the study of the correlation between these cache miss predictions and the performance, we reach the following conclusions.

For our chosen search space, the operational intensity metric correlates with the performance of a configuration. However, the performance stops increasing once a threshold on the operational intensity has been reached. For these high values of operational intensity, the observed configurations still have a variation of performance. This means that even if optimizing for operational intensity will likely lead to a configuration above the threshold, it will not ensure the best performance and will need to be supplemented with another metric.

Our set-associative model selects well-performing configurations, despite being sometimes very imprecise in its predictions. It emphasizes that the precision of the model might not be the most important feature for performance selection.Thus, sacrificing precision for speed opens interesting prospects to avoid costly analytical set-associative cache model. Finally, the fully-associative cache model is unreliable, even when restricted to our simple case study. It can even lead to the selection of memory-bound configurations that should be minimizing the amount of cache misses, according to their models (Figure 6). Therefore, we put into question the reliability of any fully-associative cache models for tile size selection, in the context of an optimizing compiler.

Outline Section 2 define the search space of configuration we will be using to evaluate each model, and introduce the core notions fully-associative cache model. Section 3 presents our set-associative model extension of this fully-associative cache model, based on the notion of *detailed footprint*. Section 4 presents the evaluation of each of these models. Section 5 provides an overview of the related work.

2 Background

In this section, we first introduce our benchmark and describe how a configuration is defined for these programs (Section 2.1). From this, we will define the search space of configurations (Section 2.2). Finally, we will present the fully associative analytical model that will be used as a reference (Section 2.3).

2.1 Benchmark

This paper considers tensor operations, and studies in particular the *matrix multiplication* and the *2D NHWC_HWCF convolution* kernels. Both kernels are presented in Figure 1. We consider problem sizes extracted respectively from the Polybench benchmark sizes [17], and from the convolutional layers of ResNet18 [9]. For the latter sizes, we consider the convolution kernels to be in inference mode, which means $N = 1$. We assume that the problem sizes are known at compile time.

In order to simplify the cache modeling, we want the non-inner dimensions of our array allocations to be aligned with the cache lines. Because the size of a cache line is 512 bits and we consider 32-bit floating point data type, it means that the J and K dimensions for matrix multiplication, or the F and C dimensions for convolution must be multiples of 16. This is the reason why we do not consider the first ResNet18 convolutional layer (where $C = 3$), and why we consider slightly padded versions of the Polybench sizes.

These kernels have two key properties: (i) the *fully permutability* of their loops (any permutation between these loops preserve correctness), and (ii) the *fully tilability* of their dimensions (any dimensions can be tiled using, for example, rectangular tiles while preserving correctness). Because the main transformations considered are *loop tiling* and *loop interchange*, these properties make the exploration of the optimization space much easier.

Therefore, for these programs, we can identify a configuration as a list of elements [23]. Each element $T(r, d)$ of this list corresponds to a loop level, where d is the dimension of the loop and r the number of times this loop is executed (also called *ratio*). Notice that we forbid partial tiles, which mean that r must divide its corresponding program size. We could also introduce some variations of element, such as $U(r, d)$ to denote an unrolled loop, or $V(d)$ to denote a vectorization, only for the last element of a list.

Listing 1 Studied kernels in our benchmark.

```
for(i = 0; i < I; i++)
  for(j = 0; j < J; j++)
    for(k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];
```

(a) Matrix multiplication.

```
for(n = 0; n < N; n++)
  for(h = 0; h < H; h++)
  for(w = 0; w < W; w++)
    for(r = 0; r < R; r++)
    for(s = 0; s < S; s++)
      for(c = 0; c < C; c++)
      for(f = 0; f < F; f++)
        O[n][h][w][f] +=
          I[n][h+r][w+s][c] * K[r][s][c][f];
```

(b) 2D NHWC_HWCF convolution with a unit stride.

Benchmark	**Problem sizes**
2D Convolution	(F, C, H/W, R/S, strides)
ResNet18-02	64, 64, 56, 3, 1
ResNet18-03	64, 64, 56, 1, 1
ResNet18-04	128, 64, 56, 3, 2
ResNet18-05	128, 64, 56, 1, 2
ResNet18-06	128, 128, 28, 3, 1
ResNet18-07	256, 128, 28, 3, 2
ResNet18-08	256, 128, 28, 3, 1
ResNet18-09	256, 256, 14, 3, 1
ResNet18-10	512, 512, 14, 3, 2
ResNet18-11	512, 256, 14, 1, 2
ResNet18-12	512, 512, 7, 3, 1
Matrix multiplication	(I, J, K)
Polybench-Large	(1000, 1104, 1200)
Polybench-Xlarge	(2000, 2304, 2608)

Figure 1: Considered problem sizes. All of them (except for the last) are L3-resident.

Listing 2 Running example: tiled matrix multiplication ($I = 3$, $J = 32$, $K = 16$). Its optimization configuration (Section 2.2) is $[T(4,k); T(3,i); T(4,k); T(2,j); T(16,j)]$.

```
for(k1 = 0; k1 < 4*4; k1+=4)
  for(i = 0; i < 3; i++)
    for(k = k1; k < k1+4; k++)
      for(j1 = 0; j1 < 2*16; j1+=16)
        for(j = j1; j < j1+16; j++)
          C[i][j] += A[i][k] * B[k][j];
```

However, because these properties do not impact the cache models, we will only consider $T(r,d)$ elements in our scheme to simplify the presentation. For example, the program in Listing 2 corresponds to the configuration $[T(4,k), T(3,i), T(4,k), T(2,j), T(16,j)]$.

2.2 Search space of configurations

From the definition of a configuration, we now build a search space which will be explored in the rest of this paper. We restrict the space of considered configurations by adding constraints, that are issued from expert knowledge about the shape of the best performing implementation. These constraints aim at increasing the density of good performing configurations, without excluding the best configurations.

Our search space is strongly inspired from the one introduced by Tollenaere et al [23], and imposes the following properties on the configurations:

- The innermost loops correspond to a register tile (called *microkernel*) that is fully unrolled and vectorized. This microkernel has to be among the best-performing ones for a given architecture. This list of good microkernel sizes is built once and for all, during an offline pass, by measuring all microkernels in isolation.

- The definition of configuration already forbid partial tiles. However, this can be too strict if a problem size has an awkward prime factor decomposition. So, we allow the sequential composition of two different microkernels. For example, if we have good-performing microkernels of sizes 6 and 7 along a dimension, we can alternate between them to form a composite tile of size $6 + 7 = 13$. This adds a new kind of element inside a configuration, that allows for the sequential composition between both variations.

- The loop above the microkernel is a loop that reuses the memory of the output array, over the C (resp. K) dimension for a convolution (resp. a matrix multiplication). This array is assumed to stay in place

in the vector registers. To ensure that this reuse is exploited, we force this loop to have at least 32 iterations, and we ensure the Load/Stores of the output array are hoisted outside this loop. We also impose that the ratio of this loop is divisible by 16, so that the scalar Load used by the microkernels iterates over the entirety of a cache line, thus improving reuse.

A microkernel plus the reuse loop directly above it already almost saturates the L1 cache, for the studied architectures. So, we focus on the L2 cache in the rest of this paper.

2.3 Fully associative cache miss modeling

Let us consider a loop level, and the subprogram composed of this loop, all the loops below it, and the statement at the innermost level. The *data footprint of an array reference for this subprogram* is the set of memory cells that are accessed during the execution of this program through this array reference. The *data footprint of a subprogram* is the union of the data footprint of its array reference. The cardinality of a data footprint is a pertinent quantity to know if the memory required by a tile of a program fits inside a cache, or if we have capacity misses. Notice that this reasoning intrinsically assumes that any element can be stored anywhere in the cache, thus it assumes a full associativity hypothesis.

We now introduce a fully associative analytical cache miss model. There are many variations on this topic [19, 21], and we choose the model of Rui et al. [12], later extended by Olivry et al. [15]. These models are able to deal with any affine program and output a parametric expression of the operational intensity, in function to the tile sizes. Then, the best tile sizes that maximize the operational intensity are found using a solver. In our paper, we only need an estimation of the number of cache misses given a non-parametric configuration. Thus, we adapt this model while keeping the same hypothesis made about the content of a cache.

To compute the number of cache misses of a program, we consider the program loops from the innermost levels to the outermost, and the corresponding subprogram starting from each of these loops. We compute the cardinality of the footprint for each subprograms, and we call *saturation level* the lowest loop level for which the footprint of its subprogram exceeds the cache capacity.

Once the footprint is computed for each subprogram, the number of cache misses is computed in the following way:

- *Below the saturation level:* There is no spilling, thus we only have cold misses. The number of cache misses is exactly the cardinality of the footprint of this tile.

- *At saturation level:* We have a band of tiles, each one fitting in cache independently, but the whole cannot fit at the same time in the cache. We assume a perfect reuse between two consecutive tiles: all the data which is used by both tiles stays in the cache. So, the number of cache misses is the sum of (i) the footprint of the first tile of the band, and (ii) the footprint of the non-reused portion of the next tiles, for every remaining tile. In our case, reuse always happens between consecutive tiles, so we can directly compute the number of cache misses as the footprint of the whole band of tiles.

- *Above saturation level:* We assume that the scenario happening below repeats itself. Thus, we just multiply the number of cache misses of the previous loop level, by the number of iteration of the current loop level.

To validate this model, we compared its prediction with the one from Dinero, configured for a fully-associative cache and with an LRU replacement policy. The average relative error between both predictions over our benchmark is about 7%.

Example Let us consider the program in Listing 2, and a fully-associative cache with a capacity of 16 cache lines. We assume a cache line size of 512 bits and 32-bits floating point elements, which means 16 elements per cache line. We also assume that the allocation of all three arrays is aligned with the start of a cache line. The cardinality of the footprint of its subprograms is shown in Figure 2, and reports the number of distinct cache lines in a footprint.

We observe that the saturation level happens above the loop $T(3, i)$. At that level, we assume that the memory accesses to array `B` are reused across these 3 iterations (corresponding to 8 cache misses). All the memory accesses to the arrays `A` and `C` are distinct, the number of cache misses corresponds to their footprint. Thus, we have 17 cache misses at that level. This scenario repeats itself 4 times (due to the surrounding $T(4, k)$ loop), which means that we predict $17 \times 4 = 68$ cache misses. This prediction is confirmed by Dinero, when configured for a fully associative cache.

3 Set-associative cache model

In this section, we present a novel set associative cache miss model. Our goal is to build a model conceptually as close as possible to the fully associative cache model introduced in Section 2.3, while being set associative. Thus, we can later use this cache model as a point of comparison to evaluate the impact of the fully associativity hypothesis on the optimization process in Section 4.

Loop	I	J	K	C[i,j]	A[i,k]	B[k,j]	Total
T(4,k)	3	32	16	6	3	32	**41**
T(3,i)	3	32	4	6	3	8	**17**
T(4,k)	1	32	4	2	1	8	11
T(2,j)	1	32	1	2	1	2	5
T(16,j)	1	16	1	1	1	1	3

Figure 2: Cardinality of the footprint of each access of the program in Listing 2, in cache lines. The "Loop" column denotes the loop level considered, from outer to inner. Assuming a fully associative cache of capacity 16 cache lines, the **bold** numbers correspond to the loop levels for which there are capacity misses. The model predicts 68 cache misses.

The main idea is to generalize the notion of footprint into a *detailed footprint* (Section 3.1), to track the amount of cache line that is mapped to each cache set. We consider the detailed footprint of the first iteration of each loop in a program and show how to compute it efficiently, using the properties of the considered program. Then, we apply the full associative cache miss analytical model on each of the cache sets to predict the number of cache misses coming from each cache set (Section 3.2).

The presented set-associative model needs to assume several additional hypotheses on the program. It can be generalized to a larger class of program, but, since its simplest limited formalization is enough for our study, we consider such extension to be outside of the scope of this paper. These additional hypotheses are: (i) the data footprint of each reference for any subprogram must be of rectangular shape, and (ii) the array references of a statement must be about different arrays (or at least have disjoint footprints), to avoid reuse across different array references of the same statement. These hypotheses are valid for our considered kernels.

3.1 Computing the detailed footprint

Detailed footprint algorithm Let us consider a a-way associative cache which has Ncs cache sets and of capacity C. This means that $C = a \times Ncs \times L$, where L is the size of a cache line.

A data footprint considers a collection of elements or cache lines, without considering the fact that they could be mapped to different cache sets. Thus, instead of considering the cardinality of such a data footprint, we keep track of the number of cache lines per cache sets. A *detailed footprint* of an array reference is a list of positive integer of size Ncs. Its k-th element is the number of cache lines of its footprint which is mapped to the k-th cache set. Notice that the sum of all the elements of a detailed footprint is equal to the number of cache lines inside a data footprint.

The algorithm to compute the detailed footprint of a single array refer-

Algorithm 1 Detailed footprint algorithm (single access)

Input: ts: footprint sizes (map: $dim\_footprint \mapsto size$)
L: size of a cache line (in number of elements)
Ncs: number of cache sets
access_offset: offset of each dim in the access (map)
align_arr: cache set of the first cell of the array
Output: *dfp_acc*: the detailed footprint for a given access

1: $dfp\_acc \leftarrow$ list of size Ncs, initialized with 0s.
2: $dfp\_acc[align\_arr] \leftarrow 1$
3: **for all** *dim_fp* dimension of the footprint **do**
4: $size \leftarrow ts[dim\_fp]$
5: **if** *dim_fp* is the innermost footprint dim **then**
6: $nrot \leftarrow \lceil size/L \rceil$
7: $stride \leftarrow 1$
8: **else**
9: $nrot \leftarrow size$
10: $stride \leftarrow access\_offset[dim\_fp]/L$
11: **end if**
12: $dfp\_acc \leftarrow$ rot_and_sum($dfp\_acc, stride, nrot$)
13: **end for**
14: **return** *dfp_acc*

Algorithm 2 Rotate and sum

Input: *dfp*: starting detailed footprint, of size Ncs
stride: stride in the cache sets occurring when incrementing a dimension
$nrot$: number of rotations needed
Output: *ndfp*: new detailed footprint

1: $ndfp \leftarrow dfp$
2: **for** $(i = 2; i < nrot; i \mathrel{+}= 1)$ **do**
3: $stride\_i \leftarrow i \times stride$
4: **for** $(k = 0; k < Ncs; k \mathrel{+}= 1)$ **do**
5: $nk \leftarrow (k + stride\_i) \bmod Ncs$
6: $ndfp[nk] \leftarrow ndfp[nk] + dfp[k]$
7: **end for**
8: **end for**
9: **return** *ndfp*

ence, for all the loop level of a program, is presented in Algorithm 1 and 2. It heavily relies on the rectangular shape of the considered footprints, which implies a regular periodic pattern of the distribution of their cache lines across the cache sets. The alignment of the non-innermost dimension of the array alignment is also critical for this algorithm to work.

We start by building the detailed footprint of the statement (Algo 1, line 1-2), which uses a single cache line. Then, for each dimension of the data footprint set, we have to consider its size in cache line, and its stride in number of cache lines jumped (Line 9-10). This computation is different if we are currently considering the innermost dimension of the data access, which iterates over the elements of a cache line (Line 6-7).

Finally, the regularity of the cache line distribution pattern across the considered dimension is exploited in Algo 2. The key intuition is that the detailed footprint of each of these iterations are the same, modulo a rotation by an offset. By summing all of these rotated detailed footprint, we obtain a new detailed footprint that accounts for the currently considered dimension.

The complexity of Algo 1, which computes the detailed footprint for one loop level, is $O(Ncs.\sum_{dim} fp\_sizes)$. As a point of comparison, the complexity of a classical simulation, based on a trace with the read and write of a program, would have been $O\left(Ncs.\prod_{dim} sizes\right)$. However, we will later need to compute the detailed footprint of each array accesses for all loop levels. Because the tiling ratio between loop levels are integral, it is possible to directly compute the detailed footprint of a loop level, using the previous loop level and Algo 2. This means that we can keep a complexity of $O\left(Ncs.\sum_{dim} prob\_sizes\right)$ for computing the detailed footprint of all the loop levels of a program.

Finally, to find the final number of cache line per cache set, we sum the individual contribution from all the detailed footprint of each array accesses. This is valid due to the non-overlapping data footprint hypothesis.

Example Figure 3 shows an example of a detailed footprint for the program of Listing 2, and a 4-way set associative cache with 4 cache sets. In practice, we only care about the relative positioning of data, and not their absolute memory address. We assume that the memory allocation of arrays `C`, `A` and `B` are allocated contiguously, in that order, and that the memory allocation of array `C` starts at the first cache set. Under this hypothesis, the memory allocation of array `A` starts at the third cache set, and the one of array `B` at the second cache set.

Let us apply our algorithm on the array access $A[i,k]$ and on the subprogram starting at loop level $T(3,i)$. The tile sizes at that level are $(i=3, j=32, k=4)$, while the memory allocation for the array A is $(mx, my \mapsto 16.mx + my)$. The footprint of the access has 2 dimensions (in memory space, these are not the iteration space dimension), and their sizes

Loop	C[i,j]	A[i,k]	B[k,j]	Total
T(4,k)	[2,2,1,1]	[1,0,1,1]	[8,8,8,8]	[**11**,**10**,**10**,**10**]
T(3,i)	[2,2,1,1]	[1,0,1,1]	[2,2,2,2]	[**5**,4,4,4]
T(4,k)	[1,1,0,0]	[0,0,1,0]	[2,2,2,2]	[3,3,3,2]
T(2,j)	[1,1,0,0]	[0,0,1,0]	[0,1,1,0]	[1,2,2,0]
T(16,j)	[1,0,0,0]	[0,0,1,0]	[0,1,0,0]	[1,1,1,0]

Figure 3: Detailed footprint for each of the loop levels of the program in Listing 2. We consider a 4-way associative cache of size 16 cache lines, which contains 4 cache sets. The fully associative version of this example is presented in Figure 2. The **bold** text indicates for which loop level a cache set saturates. The model predicts 50 cache misses.

are $mx = i = 3$ and $my = k = 4$. The initial detailed footprint for the access A is $[0, 0, 1, 0]$. Let us consider dimension mx first: this is not the innermost dimension, so $nrot = 3$ and $shift = 16/16 = 1$. After summing 3 rotations shifted by 1, we find $dfp\_acc = [1, 0, 1, 1]$. Then, let us consider dimension my: this is the innermost dimension, so $shift = 1$ and $nrot = \lceil \frac{4}{16} \rceil = 1$. Notice that this corresponds to the case where one fourth of the cache line is actually used. Therefore, the detailed footprint of this access at this loop level is $dfp = [1, 0, 1, 1]$.

3.2 Set-associative cache model

Algorithm First, we use the algorithm of Section 3.1 to compute the detailed footprint of all loop level of a program. Our set-associative cache model algorithm simply considers each cache set independently and apply a fully associative cache miss modeling (presented in Section 2.3) to predict the number of cache miss, for each cache set. Finally, we sum all the cache misses predictions over all cache set to obtain the global cache miss prediction.

Example As an example, let us consider Figure 3. The saturation level of the first cache set happens at loop level $T(3, i)$, while the over levels are one level above. Thus, we have $5 \times 4 = 20$ cache misses on the first cache set, and 10 of each of the remaining cache sets. The total number of predicted cache misses is $20 + 10 + 10 + 10 = 50$. As seen in Section 2.3, the fully associative model considers that the saturation level happens at $T(3, i)$, which is the reason why it predicted 68 cache misses. When using Dinero, configured with a pseudo-LRU replacement policy, the number of cache misses reported is 62.

Approximation Technically, this set-associative cache algorithm is inexact, and not only due to the intrinsic approximations made about the cache

behavior. The issue is that the detailed footprint of a subprogram represents the distribution of cache line for **the first iteration of each loops above it**.

This is not an issue if we have only one array access: if we consider another instance of this subprogram, the only impact of its detailed footprint would be a rotation on its elements. However, it becomes an issue when we combine the detailed footprint of different array accesses, because this rotation speed for each array accesses is different. This means that the detailed footprint of the whole subprogram changes across the iterations of the surrounding loops, which might change the saturation level of a cache line, thus the whole computation performed.

For example, assuming a 4-way associative cache with 2 cache sets, let us consider a statement with two array accesses, whose detailed footprints are $[2, 0]$ and $[3, 2]$ for a given loop level. When summing both footprint, we obtain $[5, 2]$, which means that we must spill on the first cache set, while the second can reuse its data. Now, let us consider an extra loop level above it, whose iteration shifts the first detailed footprint by an offset of 1 and the second detailed footprint by an offset of 2. This means that for the second iteration of this surrounding loop, the detailed footprints are $[0, 2]$ and $[3, 2]$. When summing both contributions, both cache sets do not have to spill. So, the saturation level of the first cache set changes, depending on the instance of the subprogram we consider. In comparison, because our model only consider the first iteration, it assumes that the first cache set will always spill, and that the saturation level does not move.

Even if we have a periodicity along the different instances, it would be expensive to consider them all and estimate their cache misses. So, despite its imprecision, we claim that our model is a usable middle ground that sacrifices precision for the speed of its evaluation. Moreover, we will show in Section 4 that it is exploitable to select performing configurations, and that it even performs better than the fully associative model.

4 Performance selection with cache models

In this section, we evaluate the ability of models to select high-performing configurations inside a search space. We consider 1000 randomly sampled sequential configurations from the search space described in Section 2.2 using the random selection algorithm described in [23].

We consider 2 CPU architectures:

- An Intel Xeon Gold 6230R CPU, with a 32KB 8-way L1 data cache (64 cache sets), a 1024KB 16-way L2 cache (1024 cache sets), a 35.8MB 11-way shared L3 cache and 2 AVX512 vector units (512-bits wide).
- A ARM ThunderX2 CN99xx processor, with a 32KB 8-way L1 data

Method	Intel	ARM
Dinero	9.5 s	25.9 s
ModelA	228 ms	125 ms
ModelFA	0.8 ms	0.7 ms

Figure 4: Average time per configuration taken by a method.

cache (64 cache sets), a 256KB 8-way L2 cache (512 cache sets), a 32MB 32-way shared L3 cache and 2 Neon vector units (128-bits wide).

We consider the L2 cache misses estimation from 4 different sources:

1. *Fully associative analytical model* (**ModelFA** - Section 2.3).
2. *Set associative analytical model* (**ModelA** - Section 3).
3. *Simulation* (**Dinero**), using the Dinero cache simulator [3], with a pseudo-LRU replacement policy.
4. *Hardware counters measurement*, using **PAPI** [14].

Analysis time of a model Figure 4 shows the average time each method took per configurations. Both models are implemented in Python, while Dinero is implemented in C. Note that the time taken by Dinero increases with the size of the trace of the application, which is why the `Polybench-Xlarge` sizes take one order of magnitude more time (78.6s, against 3.7s in average for the other sizes for the Intel CPU). On the contrary, the time taken by the analytical models remains stable across our benchmark.

Notice that our approximate set-associative model is much faster than the state-of-the-art set-associative cache models. Polycache [1] reports a 6 seconds analysis time on a single-level tiled `Polybench-large` matrix multiplication. This is an order of magnitude slower, while being applied to a much simpler code, and having a C implementation.

In Section 4.1, we study the relationship between performance and operational intensity within our search space, and where the best configurations are placed, according to each cache miss model. In particular, this study shows that fully associative cache models are unstable. To support this claim, we highlight a situation where this instability directly impacts the resulting performance. Then, in Section 4.2, we study a state-of-the-art tile size optimizer [15]. It uses a fully-associative model as a part of its cost function to choose tile sizes that maximizes the operational intensity. We show that random selection within our search space performs better than this optimizer.

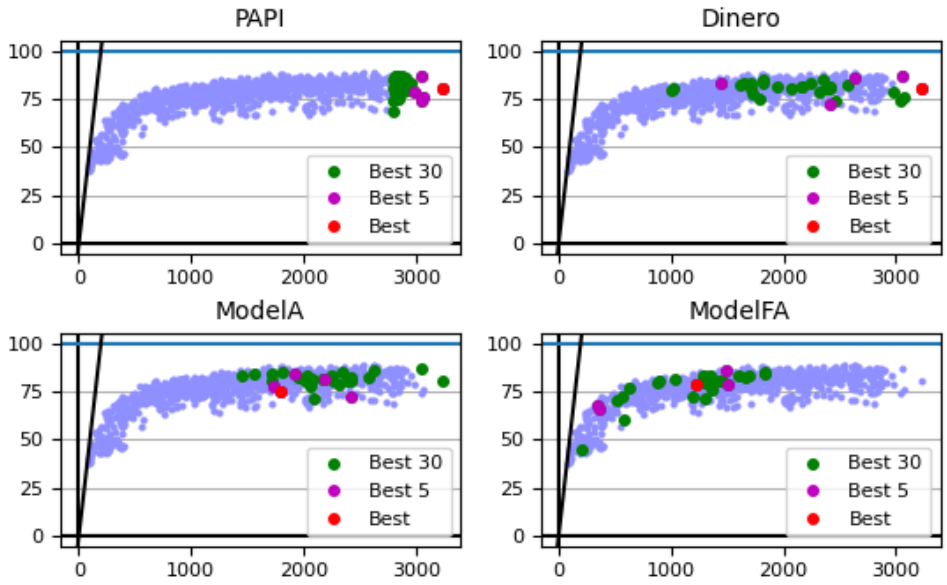


Figure 5: Roofline chart for the 1000 configurations drawn in our search space, for ResNet18_08 on our Intel architecture. The X axis is the operational intensity. The Y axis is the performance of the configuration in percentage of peak performance (higher is better). The colored points are the 30/5/the best configuration(s) that are selected according to the results provided by 4 different cache miss models.

4.1 Performance selection

Distribution of the space on a roofline plot We first study the correlation between the operational intensity and performance. We focus on the L2 cache, so we compute the operational intensity as the ratio between the volume of computation and the number of L2 cache misses.

Figure 5 shows the distribution of the 1000 configuration on a roofline plot, for the ResNet18_08 kernel, on our Intel architecture. The horizontal axis is the operational intensity, while the vertical axis is the percentage of peak performance (100% is the ceiling, higher is better). The diagonal black line on the left of each figure is the performance ceiling caused by the bandwidth of a cache.

As expected, the worst performing configurations are memory bound. We also notice that the plot shape is a band of points that follows a curve, which is passes below the corner of the roofline ceiling before slowly flattening. This shape is caused by the fact that the operational intensity is an average over the whole execution of a configuration: even if a configuration is globally compute-bound, it can be sometimes locally memory-bound. This is caused by the non-uniform distribution of load/store and cache misses

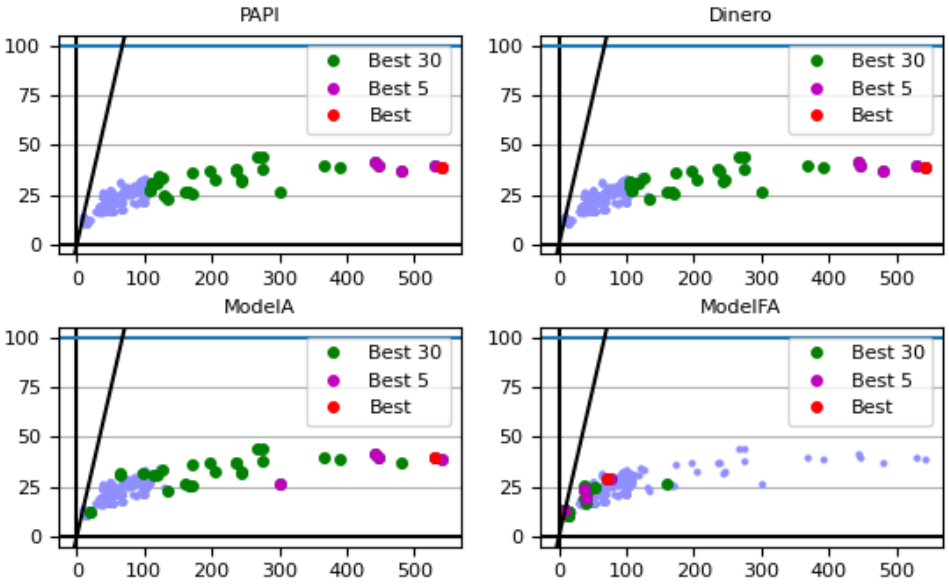


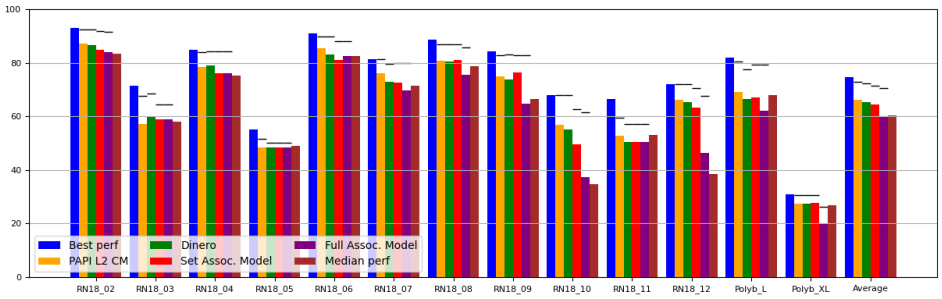
Figure 6: Roofline chart for the 1000 configurations drawn in our search space, for ResNet18_10 for our ARM architecture. This figure is similar to Figure 5 and shows a situation where the fully associative model degrades the performance.

in the program. So, a compute-bound configuration close to the memory bound area would encounter statically more memory bottleneck than a configuration on the far-right of a roofline plot.

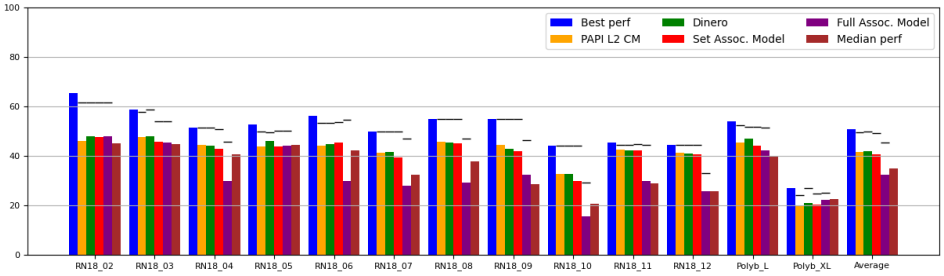
We also notice that the right part of the distribution is thick along the vertical axis (from 65% to 85% of peak performance). This means that relying only on the operational intensity metric is not enough to identify the best performing configuration, and it needs to be complemented with another metric or optimization process. Because of this, comparing the single best configuration of each model does not make sense, due to the high instability of its performance. This is why we consider a set of 30 top configurations for each model, and study its distribution of performance. Such set could be later used as the entry of a later configuration selection analytical pass, and have a good chance of having at least a point from the upper part of the roofline distribution.

Figure 5 also shows (in color) the position of the 30 top points according to each model. We observe that, contrary to the other models, the points provided by the fully associative cache model (bottom-right plot) are positioned more toward the left, and even includes some memory-bound points.

Effectiveness of performance selection In Figure 7, we estimate the quality of the selection of each model by reporting the average and the best



(a) Intel architecture.



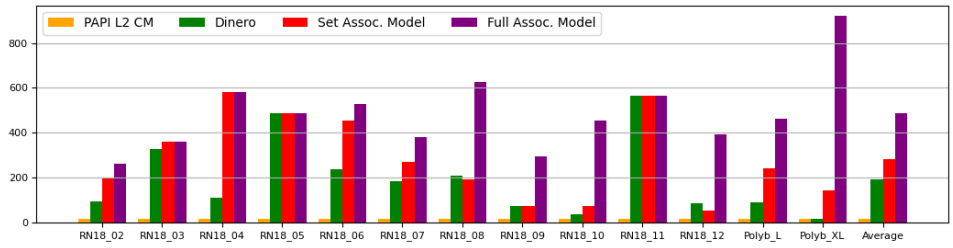
(b) ARM architecture.

Figure 7: Average (colored box) and best (black bar) performance of the 30 best configurations (out of 1000) which are selected to minimize the number of cache misses according to a metric. We also report the best and the median (500th) performance.

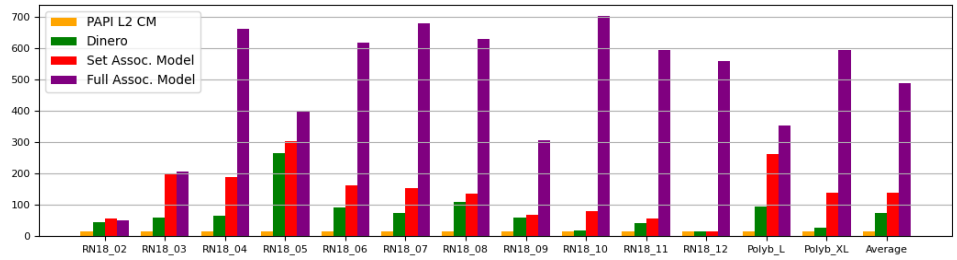
performance of the 30 top configurations according to each metric, for every program sizes of our benchmark. We also report the best performance and the median over our 1000 drawn configurations. Figure 8 complements this analysis by reporting the average ranking of these 30 top configurations, inside the decreasingly sorted list of configuration according to their operational intensity (lower is better).

1) Set associative model and imprecision: We observe that the set associative model outperforms consistently the fully associative model. This is surprising, taking into account that the precision of the set-associative model can be bad, especially for the later `ResNet18` layers. For `ResNet18_10` on the Intel architecture, if we compare the average relative error with the measured cache misses across all 1000 configurations, the fully associative model has an error of 30% while the set associative model has an error of 54% (ratio of 1.8 in average imprecision). Despite this loss of precision, the set associative model outperforms the fully associative model in the performance selection task. This demonstrates that precision of performance might not be the right metric to evaluate the efficiency of a cache model.

2) Instability of the fully associative model: From Figure 8, we observe



(a) Intel architecture.



(b) ARM architecture.

Figure 8: Average rank of the 30 best configurations (out of 1000) which are selected to minimize the number of cache miss according to a metric. Rank 1 is the configuration which has the greatest operational intensity, according to the real measurement. Lower is better, the minimum being 15.5 (which is always the value for PAPI L2 CM).

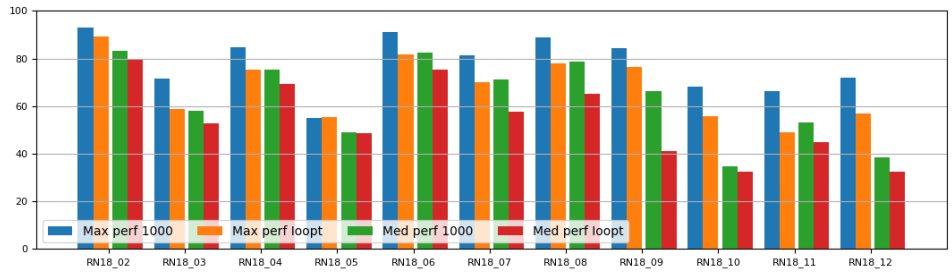


Figure 9: Maximum and median performance of the 1000 randomly drawn and the IOOpt configurations.

that the fully associative model consistently selects configurations of worse operational intensity, compared to the other models. Notice that this does not always translate into a loss of performance. For example, in Figure 5, lots of configurations can already be on the "compute bound" part of the roofline, which means that their performances are similar. So, selecting configurations of worse rank has no effect on the performance.

The opposite situation might happen: Figure 6 shows a situation for `ResNet18_10` on the ARM architecture, where a gap of performance appears between each models. This is caused by the fact that very few configurations are compute bound for this situation. Thus, the fully-associative model will mostly select memory bound configurations. Therefore, the fully associative model has some stability issues, and cannot be relied on blindly across all program sizes.

Our observation on fully associative model can be generalized to any fully associative cache models. When we compare the prediction of our fully associative cache model with the one from Dinero configured for a fully-associative LRU cache, we found an average relative error of 7%. Thus, even the predictions done by a fully-associative Dinero simulation, which is the algorithmic ideal for a fully-associative model, would encounter the same observed issues. Therefore, *any fully associative cache model (i) that tries to be precise (i.e., to match Dinero prediction) and (ii) that is applied to a set-associative cache will encounter the same instability problem when trying to select performing configurations.*

4.2 Fully associative tile size selector evaluation

In this section, we evaluate the efficiency of IOOpt [15] for finding a performing configuration. This tool searches for a loop permutation and tile sizes (i) by computing the operational intensity as a close-form formula using symbolic tile sizes; (ii) by using this expression as the cost function of an optimization problem; then (iii) by deducing from the result of this solver the optimal loop permutation and tile sizes that theoretically maximizes the operational intensity. The operational intensity formula is constructed by using hypothesis from a fully-associative cache model (Section 2.3). Notice that IOOpt allows partial tiles, which means that its configurations are mostly outside of our search space.

IOOpt is not suitable to optimize for register-level, but we can compensate by providing it with a good performing microkernel to be used as the innermost loops of its configuration, then ask it to build its surrounding loops. So, we consider a list of performing microkernels (the same used in Section 2.2) whose sizes divide the program sizes. For each of these microkernels, we call IOOpt to complete the scheme.

Figure 9 compares the set of configurations synthesized by IOOpt (one per valid microkernel), with the 1000 randomly drawn configurations in our

search space. Notice that we do not have a way to predict the final performance of a configuration completed by IOOpt, from only its microkernel. So, if we pick a microkernel randomly, the median performance of this figure shows the expected performance of such candidate. Therefore, the figure shows that, in average, using IOOpt is worse than using a random selection of a configuration within our space.

5 Related work

Set-associative analytical cache modeling Ghosh et al. [4] introduce the Cache Miss Equations model, for affine programs. They model dependencies with reuse vectors. Their main idea is to generate a collection of linear Diophantine equations for each reuse vector, such that each solution corresponds to the cache lines of two accesses using the same cache set. Then, they count the amount of cache lines between the producer and consumer of a dependency and to compare it with the cache associativity.

Chatterjee et al. [2] propose an alternative model, based on Presburger formula. They also focus on the composability of their cache modeling analysis, by differentiating the cache miss that must occurs, and those that occurs only depending on the initial state of the cache. Bao et al. [1] have adopted a similar approach for all polyhedral programs, and have integrated it inside the PolyCache tool. They report the same prediction through their static algorithm than a Dinero simulation using an LRU policy, but with a significantly reduced time. The main limitation of such approach is the complexity of the Presburger set manipulations, and the computation of their cardinality, which can be prohibitive in time.

A suite of papers by Temam et al. [22], then Harper et al. [7, 8] considers an approach close to ours. They consider perfectly nested loop programs with a rectangular loop domain, and they consider a notion close to the detailed footprint of each reference inside such program. Their algorithm computes directly the footprint using complex close form formula, but needs to average the footprint over the granularity of an interval of cache set. In comparison, our algorithm does not approximate along cache sets and is closer to a simulation at each loop level, while exploiting the regularity of the memory accesses to minimize its complexity.

Hong et al. [10] consider the issue of finding the best padding in order to eliminate conflict misses, i.e., to have a perfect balance of usage between cache sets. They also consider programs whose array references have a hyper-rectangular footprint. The same approach is considered by Li et al. [13] in the context of iterative stencil operations.

6 Conclusion

From our observation, we do not recommend the use of fully associative analytical cache modeling inside an optimizing compiler that consistently aims for the best performances. We propose a simple approximate set-associative cache model for tensor operations that is usable for configuration selection.

We are not aware of state-of-the-art set-associative analytical cache models that scales and can be applied to a general class of program. However, sacrificing precision for scalability could be a valid approach for an approximate set-associative cache model, to identify reliably the best configurations of a space.

References

[1] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, and P. Sadayappan. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages*, 2(POPL), dec 2017.

[2] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. *SIGPLAN Not.*, 36(5):286–297, May 2001.

[3] J. Edler and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator, 1999. Last accessed 26 February 2024.

[4] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, jul 1999.

[5] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3), May 2008.

[6] T. Gysi, T. Grosser, L. Brandner, and T. Hoefler. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 816–829, New York, NY, USA, 2019. Association for Computing Machinery.

[7] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Predicting the cache miss ratio of loop-nested array references. Report 336, University of Warwick. Department of Computer Science, Dec. 1997.

[8] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transaction on Computers*, 48(10):1009–1024, Oct 1999.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE Computer Society, June 2016.

[10] C. Hong, W. Bao, A. Cohen, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. Effective padding of multidimensional arrays to avoid cache conflict misses. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 129–144, New York, NY, USA, 2016. Association for Computing Machinery.

[11] R. Iyer. On modeling and analyzing cache hierarchies using CASPER. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pages 182–187, 2003.

[12] R. Li, A. Sukumaran-Rajam, R. Veras, T. M. Low, F. Rastello, A. Rountev, and P. Sadayappan. Analytical cache modeling and tile-size optimization for tensor contractions. In M. Taufer, P. Balaji, and A. J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2019. ACM.

[13] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems*, 26(6):975–1028, nov 2004.

[14] P. Mucci, S. Moore, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters, 01 1999. `https://icl.utk.edu/papi/`.

[15] A. Olivry, G. Iooss, N. Tollenaere, A. Rountev, P. Sadayappan, and F. Rastello. IOOpt: Automatic derivation of I/O complexity bounds for affine programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1187–1202, New York, NY, USA, 2021. Association for Computing Machinery.

[16] A. Pitchanathan, K. Grover, and T. Grosser. Falcon: A scalable analytical cache model. *Proceedings of the ACM on Programming Languages*, 8(PLDI), June 2024.

[17] L.-N. Pouchet and T. Yuki. PolyBench/C: The polyhedral benchmark suite, version 4.2, 2016. `http://polybench.sf.net`.

[18] L. Renganarayana and S. Rajopadhye. Positivity, posynomials and tile size selection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008.

[19] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '00, page 146–153, USA, 2000. IEEE Computer Society.

[20] N. R. Shah, A. Misra, A. Miné, R. Venkat, and R. Upadrasta. Bullseye: Scalable and accurate approximation framework for cache miss calculation. *ACM Transactions on Architecture and Code Optimization*, 20(1), Nov. 2022.

[21] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, page 101–121, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. *SIGMETRICS Performance Evaluation Review*, 22(1):261–271, may 1994.

[23] N. Tollenaere, G. Iooss, S. Pouget, H. Brunie, C. Guillon, A. Cohen, P. Sadayappan, and F. Rastello. Autotuning convolutions is easier than you think. *ACM Transactions on Architecture and Code Optimization*, 20(2), mar 2023.

[24] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3), June 2015.

[25] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communication of the ACM*, 52(4):65–76, 2009.

[26] J. Xue. *Loop tiling for parallelism.* Kluwer Academic Publishers, USA, 2000.