Analytical Modeling of Set-Associative Caches for Optimizing Tensor Operations

Journal:	*Transactions on Architecture and Code Optimization*
Manuscript ID	TACO-2025-88
Manuscript Type:	Original Work
Date Submitted by the Author:	25-Apr-2025
Complete List of Authors:	Iooss, Guillaume; Inria Research Centre Grenoble Rhône-Alpes Guillon, Christophe; Inria Research Centre Grenoble Rhône-Alpes Rastello, Fabrice; Inria Research Centre Grenoble Rhône-Alpes Cohen, Albert; Google Inc, Sadayappan, P; The University of Utah, Computer Science & Engineering
Computing Classification Systems:	Compilers, Analytical cache modeling, Tensor Operations

Analytical Modeling of Set-Associative Caches for Optimizing Tensor Operations

GUILLAUME IOOSS, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
CHRISTOPHE GUILLON, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
FABRICE RASTELLO, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
ALBERT COHEN, Google, France
SADAY SADAYAPPAN, University of Utah, Utah, USA

Optimizing for data cache memories is a difficult problem for a compiler, and an important performance bottleneck. Indeed, the behavior of a cache is complex to model statically, due to the cache policies (associativity, eviction), and the complex interplay with the mechanisms of a superscalar microarchitecture. Thus, an analytical cache model is forced to make simplifying hypotheses, often assuming full associativity, allowing it to run in reasonable time.

In this paper, we consider tensor operations, a class of programs that includes matrix multiplication and convolution, among others. We leverage the properties of this class of programs to design and implement a novel set-associative analytical cache model. This model is much faster compared to a classical set-associative model. When optimizing a program by searching through a space of code generation configurations, we show that the model is suitable for statically selecting the configurations with the least cache misses.

We also provide some insights on the effectiveness of analytical models to select high-performing configurations. Considering a representative configuration space, we show that fully-associative analytical models perform barely better than random, which causes the performance of their selection to be unstable. This showcases the necessity of scalable, analytical set-associative models.

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

Additional Key Words and Phrases: Analytical cache modeling, operational intensity, tensor operations

ACM Reference Format:
Guillaume Iooss, Christophe Guillon, Fabrice Rastello, Albert Cohen, and Saday Sadayappan. 2025. Analytical Modeling of Set-Associative Caches for Optimizing Tensor Operations. 1, 1 (April 2025), 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

When optimizing a program, a programmer has to choose the optimizing transformations to be applied and their order. Between the nature of these transformations, their ordering and the additional required parameters (e.g., a tile size), the choices are numerous, and a single poor decision can deteriorate the performance, with no hope of recovery. Thus, the goal is to find (automatically, for an optimizing compiler) optimization strategies that find the best-performing sequence of transformations—called a *configuration*—or at least isolate high-performing configurations, inside the complex *search space* of valid implementations.

Authors' Contact Information: Guillaume Iooss, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France, guillaume.iooss@inria.fr; Christophe Guillon, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France, christophe.guillon@inria.fr; Fabrice Rastello, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France, fabrice.rastello@inria.fr; Albert Cohen, Google, Paris, France, albertcohen@google.com; Saday Sadayappan, University of Utah, Salt Lake City, Utah, USA, saday@cs.utah.edu.

One commonly accepted intuition is that a strategy should reflect the memory hierarchy of the hardware to which it is mapped. For example, if we consider BLIS [29] 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. Thus, the importance of metrics about the cache, which would guide the optimizations choices performed at these levels.

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 sequential programs on CPU. Previous works [7, 28, 29] 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, which 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 stored inside a cache. The *operational intensity* [31] 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 also be 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. A common approximation is about the associativity of the cache: approximating a set-associative cache as fully associative [8, 19, 21, 23, 25, 32] allows to compare the cardinality of the *memory footprint* of a tile with the cache size to detect if capacity misses occur. In contrast, some analytical models consider cache sets [1, 3, 6, 12, 27] which are much more precise, but are more complex and require a significant analysis time. In summary, by integrating approximations within an analytical model, one may trade analysis time with the precision of the model's prediction.
- **Simulation:** Another approach is to simulate the behavior of a cache step by step [5, 14]. 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: it 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.

Contribution. We propose a new set-associative analytical cache model, specialized for tensor operations. This model is both more reliable than a fully-associative analytical cache model, and faster than existing set-associative analytical ones. This model is especially useful as a component of any optimization process that searches through a configuration space [28]. After sampling many configurations inside this space, we want to decide *statically* the ones that are worth keeping, for further optimization or measurement. Having a reliable and fast analytical cache model that can be applied to many configurations at once is critical.

In addition to this main contribution, we use this opportunity to provide insights on the effectiveness of analytical cache models at selecting configurations, and on the shape of the space. We take advantage of the regularity of tensor operations (rectangular domains, simple access pattern) and the small size of commonly used convolution kernels (L3-resident) to run both simulations and analytical models over many configurations. In particular, we show that, in this simple case, a selection guided by a fully-associative analytical model performs only sightly better than a random selection. This emphasizes the importance of developing efficient general set-associative analytical cache models.

Outline. Section 2 defines the search space of configurations we will be using to evaluate each cache model, and introduce the core notions of a fully-associative cache model. Section 3 presents our set-associative cache model, that is based on this previous fully-associative model, and the notion of *detailed footprint*. Section 4 presents the empirical evaluation of these models. Section 5 discusses related work.

2 BACKGROUND

Let us we first introduce our benchmark suite and describe how an optimization 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 Benchmarks

This paper considers tensor operations, and studies in particular the *matrix multiplication* and the *2D NHWC_HWCF convolution* kernels. Both kernels are presented in Listing 1. We consider problem sizes extracted, respectively, from the Polybench benchmark sizes [22], and from the convolutional layers of ResNet18 [11]. For the latter sizes, we consider the convolution kernels to be in inference mode, which means that the batch size $N = 1$. 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 *full permutability* of their loops (any permutation between these loops preserve correctness), and (ii) the *full 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 represent a configuration as a list of elements [28]. 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 means that r must divide its corresponding program size.

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

Benchmark	**Problem sizes**
2D Convolution	(F, C, H/W, R/S, strides)
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)

Fig. 1. Considered problem sizes. The sizes for dimensions J and K for the Polybench sizes are slightly padded, in order to be multiples of 16. 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];
```

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. 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 [28], 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 variations of microkernels in isolation.
- The definition of configuration already forbids 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 $6 + 7 = 13$. We add 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 [24, 26], and we choose the model of Rui et al. [15], later extended by Olivry et al. [20]. 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

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

Fig. 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.

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 entire band of tiles.
- *Above saturation level:* We assume that the scenario that occurs below this loop level 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 1000 randomly sampled configuration for each of our benchmark sizes is about 7% on average.

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 occurs 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. This model reuses most of the idea of the fully associative cache model introduced in Section 2.3, while adding the concepts needed for set-associativeness.

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 fully-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).

Our set-associative model needs to assume several additional hypotheses on the program, which are:

(1) the data footprint of each reference for any subprogram must be of rectangular shape,
(2) 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,
(3) for each array accesses, the stride of each dimension of this access (except the innermost one) must be a multiple of the size of a cache line. This avoids alignment issues related to cache lines.

All of these hypotheses are valid for our considered kernels and configuration from our space (Section 2.2). Note that we could generalize our cache model algorithm to remove some of these hypotheses. However, for the sake of simplicity of explanation and formalism, we consider such extension to be outside of the scope of this paper.

Algorithm 1 Detailed footprint algorithm (for a single array access)

```
Input: ts: footprint sizes (mapping: dim_footprint ↦ size)
       access_offset: offset of each dimension in the access, in number of elements (mapping: dimension ↦ offset)
       L: size of a cache line (in number of elements)
       Ncs: number of cache sets
       align_arr: cache set of the first cell of the array
Output: dfp_acc: the detailed footprint for a given access
 1: dfp_acc ← list of size Ncs, initialized with 0s.
 2: dfp_acc[align_arr] ← 1
 3: for dim_fp dimension of the footprint do
 4:     size ← ts[dim_fp]
 5:     if dim_fp is the innermost footprint dimension then
 6:         nrot ← ⌈size/L⌉
 7:         stride ← 1
 8:     else
 9:         nrot ← size
10:         stride ← access_offset[dim_fp]/L
11:     end if
12:     dfp_acc ← Rotate_and_sum(dfp_acc, stride, nrot)
13: end for
14: return dfp_acc
```

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

Algorithm 2 Rotate and sum (grows a detailed footprint along a dimension)

```
Input: dfp: starting detailed footprint, of size $Ncs$ (the number of cache sets)
    stride: stride (in number of cache set), occurring when incrementing along the dimension
    nrot: number of rotations needed to be done
Output: ndfp: new detailed footprint
1: ndfp ← dfp
2: for ($i = 2; i < nrot; i += 1$) do
3:     $stride\_i \leftarrow i \times stride$
4:     for ($k = 0; k < Ncs; k += 1$) 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
```

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 reference, 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 for a single instance 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 and from the known sizes of each arrays, the memory allocation of array `A` starts at the third cache set, and the one of array `B` at the second cache set.

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]

Fig. 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.

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 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.

Figure 4 details a concrete example, for which our set-associative cache model is not precise. We assume a 4-way associative cache with 2 cache sets, and a statement with two array accesses, whose detailed footprints are $[2,0]$ and $[3,2]$ for a given loop level. When adding both footprints, we obtain $[5,2]$, which means that we must spill on the

Sizes: T=I=2, J=5, V=16
Accesses: $A[i,t,v]$ and $B[j,v]$
Scheme: [T(2,t), T(5,j), T(2,i), T(16,v)]

2 cache sets, Associativity = 4
1 cache line = 16 elements

Loop	A[i,t,v]	B[j,v]	Total
T(2,t)	[2,2]	[3,2]	[**5**,4]
T(5,j)	[2,0]	[3,2]	[**5**,2]
T(2,i)	[2,0]	[1,0]	[3,0]
T(16,v)	[1,0]	[1,0]	[2,0]

Our set-associative algorithm
Cache misses predicted: $5 \times 2 + 4 = 14$.

Iteration t=0:

Loop	A[i,t,v]	B[j,v]	Total
T(5,j)	[2,0]	[3,2]	[**5**,2]
T(2,i)	[2,0]	[1,0]	[3,0]
T(16,v)	[1,0]	[1,0]	[2,0]

Iteration t=1:

Loop	A[i,t,v]	B[j,v]	Total
T(5,j)	[0,2]	[3,2]	[3,4]
T(2,i)	[0,2]	[1,0]	[1,2]
T(16,v)	[0,1]	[1,0]	[1,1]

Cache set #0: B from iteration $(t = 0)$
reused at iteration $(t = 1)$.
$\Rightarrow$ **Only cold misses.**
Cache set #1: Only cold misses.

Cache misses predicted: $5 + 4 = 9$.

Fig. 4. Example of situation where our set-associative analytical cache model is an approximation, due to the assumption that the first iteration of a loop is representative of the rest of the iterations. This imprecision comes from the fluctuating saturation level on the first cache set.

first cache set, while the second can reuse its data. Now, let us consider the loop level above it: its iteration shifts the first detailed footprint by an offset of 1 cache set, and the second detailed footprint by an offset of 2 cache sets. This means that for the second iteration of this surrounding loop, the detailed footprints are $[0, 2]$ and $[3, 2]$ (cf right part of Figure 4). 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 considers the first iteration, it assumes that the first cache set will always spill, and that the saturation level does not move. This leads to a difference in the prediction of the cache misses.

This model could be modified to consider the saturation level, for each instance of the external loops, using a reasoning similar to Figure 4. However, even if we would regain the lost precision, we would lose in terms of speed of the model analysis time. So, the presented version of our algorithm is an interesting middle ground that maintains a reasonable analysis time, through the use of an approximation. We show in Section 4 that our set-associative cache model preserves the ordering much better than a fully associative model, which is useful when we consider many configurations from a space.

4 EFFECTIVENESS OF THE CACHE MODEL

This section evaluates the ability of cache models to select high-performance configurations inside a search space, and focuses on answering two questions:

Q1 How does these cache models compare with each other, in terms of both analysis time and quality of their selected configurations?

Q2 How useful are these cache models and the configuration selection process with respect to performance?

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).
- An ARM ThunderX2 CN99xx processor, with a 32KB 8-way L1 data 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 use the `gcc` compiler with the flags `-O3 -march=native -fno-align-loops`.

For each architecture and benchmark, we consider 1000 sequential configurations, randomly sampled from the search space described in Section 2.2 using the random selection algorithm described in [28]. Then, for each configuration, 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 [5], with a pseudo-LRU replacement policy.
(4) *Hardware counters measurement*, using **PAPI** [18].

4.1 Q1 - Quality and analysis time of the cache models

4.1.1 Quality of the cache models.

Evaluation metric of the quality of a cache model. The usual metric to evaluate the precision of a cache model is the relative error with a reference value. This reference value can be obtained either from measurements or from a precise (but expensive) simulation. In the context of evaluating the quality of a selection of configurations, we argue that this metric does not consistently capture the usefulness of a model. Indeed, we wish to validate that the top-scoring configurations, according to an analytical model, do correspond to the best one in practice. For example, if a cache model consistently overestimates a number of cache misses by the same constant factor, then it would still correctly identify the configurations with the lowest cache misses.

So, our metric should evaluate the fact that the configurations which are thought to be the best (according to a model), are really the best ones when we measure their number of cache misses. Therefore, we focus on the *order* between the configurations, and the *Spearman rank correlation coefficient* is a suitable metric to evaluate the quality of our analytical models.

Evaluation of the quality of the ordering. For a given benchmark and its 1000 configurations, we consider the orderings of these configurations according to the score given by each of our cache models. Figure 5 compares these three orderings with the real ordering obtained from measurements, and reports their Spearman rank correlation coefficients.

In addition, we also focus on the best configurations identified by each models, since they are the one of interest. Figure 6 reports the normalized average ranking of the 30 top configurations according to each metric, for each program size of our benchmark, within the decreasingly sorted list of configurations according to their operational intensity (lower is better).

For both figures, we observe that the quality of the set associative model is consistently above the one from the fully associative model.

In Figure 5, we note that there are several anomalous Spearman coefficients for `ResNet18_04/05/11` on the Intel architecture. For the last two benchmarks of this list, the computation is L2-resident: only the cold cache misses are counted, and the same score is given for all 1000 configurations and for all 3 models (Dinero included). Therefore, the

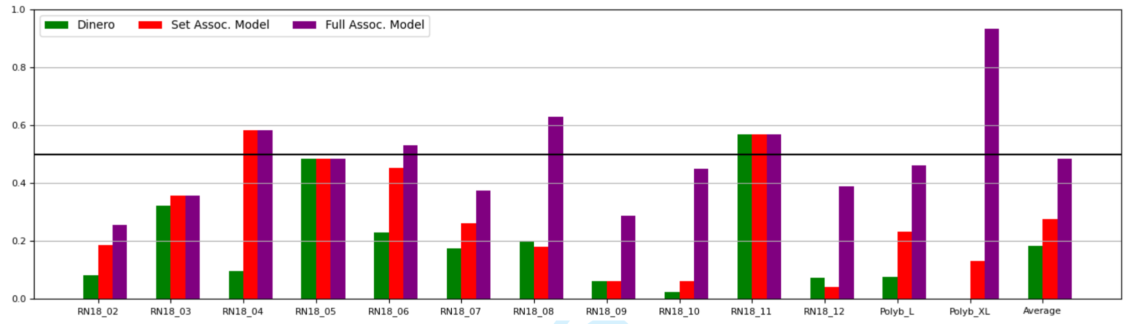
Benchmark	Dinero	ModelA	ModelFA
RN18_02	0.947	0.887	0.834
RN18_03	0.773	0.588	0.585
RN18_04	0.921	-0.011	-0.011
RN18_05	-0.013	-0.013	-0.013
RN18_06	0.469	**0.492**	-0.055
RN18_07	0.897	**0.891**	0.615
RN18_08	0.834	**0.890**	0.175
RN18_09	0.903	**0.899**	0.694
RN18_10	0.846	**0.800**	0.412
RN18_11	0.001	0.001	0.001
RN18_12	0.853	**0.827**	0.452
PolybL	0.695	0.741	0.734
PolybXL	0.998	**0.839**	-0.050

(a) Intel architecture

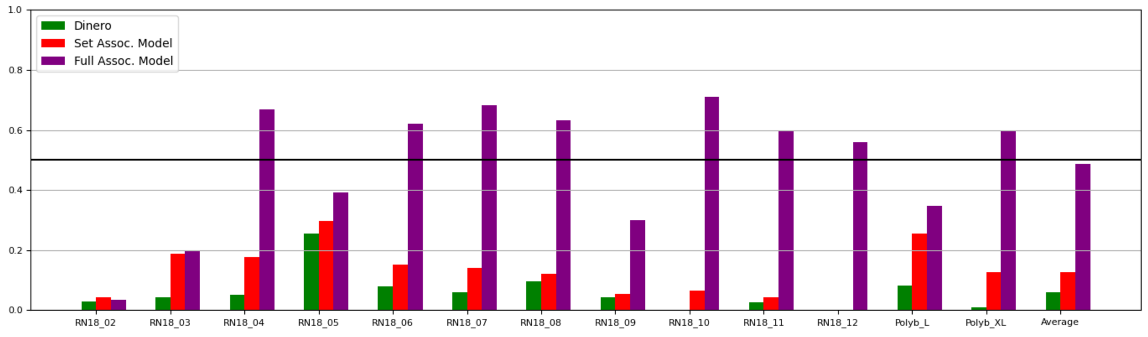
Benchmark	Dinero	ModelA	ModelFA
RN18_02	0.915	0.912	0.818
RN18_03	0.893	0.879	0.845
RN18_04	0.934	**0.852**	0.174
RN18_05	0.777	**0.723**	0.459
RN18_06	0.891	**0.863**	0.299
RN18_07	0.957	**0.918**	0.245
RN18_08	0.936	**0.915**	0.261
RN18_09	0.963	**0.893**	0.314
RN18_10	0.967	**0.659**	-0.033
RN18_11	0.949	**0.649**	-0.039
RN18_12	0.762	**0.694**	0.049
PolybL	0.747	0.671	0.645
PolybXL	0.796	**0.549**	-0.002

(b) ARM architecture

Fig. 5. Spearman's rank correlation coefficient of cache models, compared to the measured cache misses ordering. Each value are in $[-1; 1]$, 1 being the exact ordering, -1 being the reverse ordering and 0 meaning no correlation between orderings. Higher is better.



(a) Intel architecture.



(b) ARM architecture.

Fig. 6. Normalized average rank of the 30 best configurations (out of 1000) which are selected to minimize the number of cache miss according to a metric. The lower ranks are the configurations with the greatest operational intensity, according to the real measurement: lower is better. The original average rank values were normalized it to $[0, 1]$ and the horizontal bar at 0.5 corresponds to a random selection.

Method	Intel	ARM
Dinero (in C)	3.7 s PolybXL: 78.6s	7.9 s PolybXL: 242s
ModelA (in Python)	207 ms PolybXL: 477 ms	110 ms PolybXL: 307 ms
ModelFA (in Python)	0.85 ms PolybXL: 0.32 ms	0.80 ms PolybXL: 0.39 ms

Fig. 7. Average analysis time of a cache model per configuration. Because all the benchmarks have a similar volume of computation, except for Polybench-XLarge (PolybXL), we report separately the average across all the configurations of all benchmarks (minus PolybXL), and the average over all the configurations of PolybXL.

ordering is fully random and the 30 selected configurations are random. `ResNet18_04` is slightly different, since both ModelA and ModelFA are optimistic and think that this problem is L2-resident. However, Dinero detects conflict misses due to its modelisation of the cache replacement policy. This explains why Dinero model manages to obtain a good Spearman coefficient for this size, while both analytical models do not.

Figure 6 completes these observations by reporting the average "real rank" of the 30-best configurations, according to each model. This figure proves that this ranking issue does affect the best configurations (the ones estimated to have the lowest amount of cache misses), which is precisely the part on which we want a reliable selection from our model. In particular, we notice that the fully associative model is barely better than random at selecting the configurations with the least number of cache misses.

4.1.2 Analysis time of a model. We now consider the average time taken by each model, to estimate the number of cache misses of a configuration. A perfect comparison of the analysis time of these models is difficult due to several factors. First, while Dinero is written in C, both analytical models (ModelA and ModelFA) are written in Python. Second, depending on the algorithm, different factors have the greater impact on the time taken: the volume of computation (direct measurement), the size of the trace (Dinero), the number of cache sets (ModelA) or the number of cache levels (ModelA + ModelFA) can be impactful. This is why we will be reporting the analysis time of Polybench-XLarge separately, since their volume of computation is an order of magnitude greater than the rest of the benchmarks.

We report in Figure 7 the average time taken by each method per configuration, when they are run on an Intel i5-8365U CPU with 32 Go of RAM.

It also confirms that the time taken by Dinero scales with the size of the computation. The time taken by ModelA also increases, since it depends on the number of cache sets and the sum of the ratio of all the loops, but it is more manageable. In contrast, the time taken by ModelFA is constant, since it depends mainly on the number of loop levels in the considered configuration. The order of magnitude between ModelA and ModelFA are expected, since ModelA is calling ModelFA once per cache sets, and our L2 caches have 1024 cache sets. As expected, the fully-associative analytical model is the fastest, and the simulation is the slowest.

We have also evaluated the time taken by Polycache [1], which is a state-of-the-art set-associative analytical cache model based on polyhedral calculation, whose prediction matches the one from Dinero. We consider a configuration for `ResNet_02` ($[T(2,x), T(2,c), T(2,f), T(4,y), T(2,f), T(28,x), T(3,w), T(32,C), U(3,h), U(14,y), V(16,f)]$), which runs at 87% of the performance peak on our considered Intel machine. When we passed this configuration to the Polycache algorithm (reimplemented using the isl calculator [30]), it was unable to produce an answer after waiting for one hour. Even adding some simplifying assumption, such that setting the associativity of the cache to 1, was not enough to

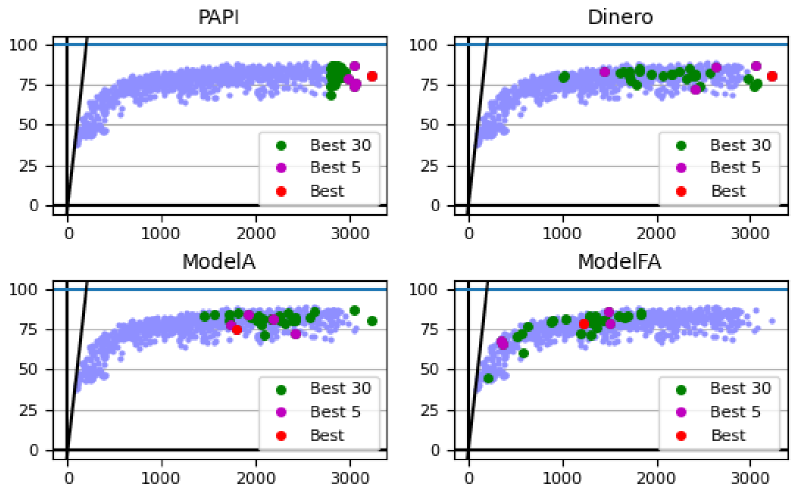


Fig. 8. Roofline chart for the 1000 configurations drawn in our configuration space, for ResNet18_08 on our Intel architecture. The X axis is the operational intensity. The Y axis is the performance of the configuration in percentage of peak performance (higher is better). The colored points are the 30/5/the best configuration(s) that are selected according to the results provided by 4 different cache miss models.

produce an answer. This is due to the complexity of the schedule of our considered configuration, which includes multiple levels of tiling, and which causes scalability issues in the polyhedral calculations.

4.2 Q2 - Link between performance and configuration selection through cache model

The previous subsection is technically enough to justify the interest of our set-associative approximate cache model for selecting configurations. So, in this subsection, we take a step back and focus on insights about the shape of the configuration space, and how efficient is the cache misses metric at sampling them.

Our main question is to check that the configurations that are found to be the best by a cache model are among the best performing ones. Indeed a classical approach, in particular for tile size selection [8, 19–21, 25], consists in using an analytical model (usually fully associative) as a cost function. In this subsection, we prove that, even in our simple configuration space, this approach is inconsistent in performance. Therefore, we need some notion of cache sets in an analytical cache model in order to ensure stable performance.

4.2.1 Shape of the configuration space on a roofline plot. We first study the correlation between the operational intensity and performance. The sizes of our benchmarks make the L2 cache one of the possible major bottleneck. Therefore, we focus on the operational intensity associated to the L2 cache, computed as the ratio between the volume of computation and the number of L2 cache misses.

Figure 8 and Figure 9 show two examples of distribution of the 1000 configurations on a roofline plot, for ResNet18_08 on our Intel architecture, and for the ResNet18_10 on our ARM architecture. Let us ignore (for now) the colored points and only focus on the position of all the points, which is the same for all 4 subfigures. The horizontal axis is the operational intensity, while the vertical axis is the percentage of peak performance (100% is the theoretical 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.

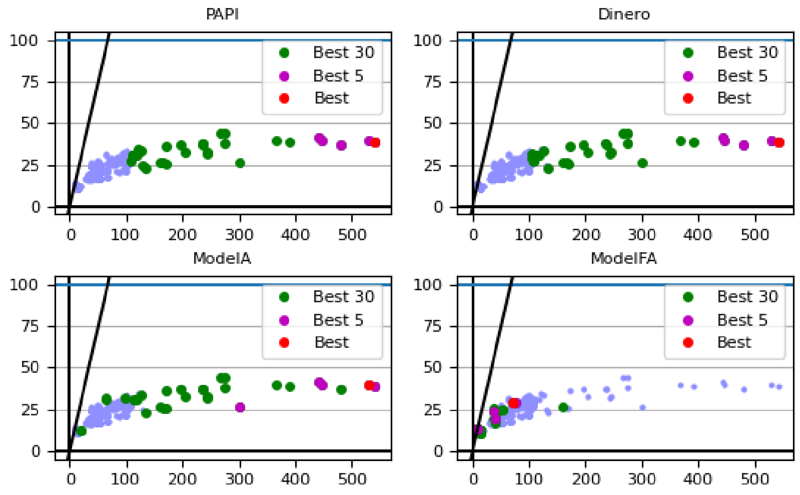


Fig. 9. Roofline chart for the 1000 configurations drawn in our configuration space, for ResNet18_10 for our ARM architecture. This figure is similar to Figure 8 and shows a situation where the fully associative model degrades the performance.

The performance of the ARM architecture is capped at 65% of the machine peak performance. This bottleneck is caused by the front-end of this architecture, which is why we were also not able to go pass this limit when running microkernels in isolation, even with ideal memory conditions.

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 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 sometimes be locally memory-bound. This is caused by the non-uniform distribution of load/store and cache misses 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% peak performance). This means that relying blindly 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 much sense, due to the high instability of its performance on the right of the rooflines. Therefore, we will consider a set of 30 top configurations according to each model, and study their performance. Such a 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.

4.2.2 Position of the best configuration, according to each model. We now consider Figure 8 and Figure 9 in their entirety, colored points included. Each subfigure corresponds to a different cache model, and their colored points to the position of the configurations with the best prediction according to their associated cache model. For example, in the top-left subfigure (PAPI), the best configurations must be the ones the most on the right, because the “prediction” made by this model is the actual measurement.

The positions of these configurations along the horizontal axis (operational intensity) are coherent with the observations made in Section 4.1. When we study their position along the vertical axis (performance), we observe that the precision of the model really matters when obtaining a compute-bound configuration is hard (or if the problem is memory-bound). In Figure 8, most of the configurations are already on the “compute bound” part of the roofline. This means that their performance are similar, and a mediocre rank will not impact the performance.

However, if we look at Figure 9, which shows the distribution of the space for `ResNet18_10` on our considered ARM architecture, very few configurations are compute-bound. This space is harder to optimize, and the rank of the selected configuration becomes critical to ensure no loss of performance. Indeed, we can see that the fully-associative model mostly select memory-bound configurations, which translates into a drop of performance.

We conclude that the fully associative model has accuracy and stability issues that penalize performance. It cannot be relied upon across all program sizes.

4.2.3 Discussion about the quality and scalability of analytical cache model. The conclusion we made on the fully-associative model can be generalized to all fully associative cache models. Indeed, when we compare the prediction of our fully associative cache model with the one from a Dinero simulation configured for a fully-associative LRU cache, we found an average relative error of 7%. This gap is low enough to assume that the selection made using a fully-associative Dinero will be similar.

The issue is that the prediction of this fully-associative Dinero is the target of any fully-associative analytical cache model. Thus, any fully-associative analytical cache model that aims at being precise will encounter the same instability issues when we apply them to a set-associative cache, over our (extremely simplified) configuration space.

This proves that one **must** use a set-associative model for a set-associative cache for the prediction to be consistent over multiple problem sizes. The issue is that set-associative analytical cache models usually do not scale in complexity, which prevents their practical use inside a compiler infrastructure.

In this paper, we have shown that some well-chosen approximations could yield a scalable set-associative cache model, with only a small degradation of its quality. This opens a new approach that consists on exploring approximations in the design of a set-associative cache model, in order to build one which is both scalable, usable and applicable to a broader class of program.

5 RELATED WORK

Fully-associative analytical cache modeling. Gysi et al [8] presented Haystack, a cache model for fully-associative cache with LRU cache replacement policy. It is based on the notion of reuse distance [2], which is computed symbolically through the Barvinok algorithm. Shah et al. [25] improved the scalability of this technique with BullsEye, using approximation and advanced linerization techniques, allowing them to achieve a significant speed-up while retaining their accuracy.

Recently, Pitchanathan et al. [21] proposed Falcon, a fully-associative analytical cache model that was optimized for performance. In particular, it runs efficiently for neural network computations.

Hsu and Kremer [13] compare different state-of-the-art analytical cost models for tile size selection and padding factor selection for matrix multiplication and LU decomposition. All the considered tile size selection analytical models assume that the cache is fully associative.

Sarkar et al. [24] compute a symbolic expression of the number of cache misses for perfected affine loop nests with a small number of loop levels, a single level of tiling, then they introduce a technique to solve it. It is based on the notion of distinct cache lines accessed by a single tile, which can be viewed as a data footprint at the granularity of cache lines. This approach was later used by Shirako et al. [26] as a pessimistic analytical cache model (called DL). They uses it in conjuction to an optimistic analytical cache model (ML, which assumes that the cache replacement policy between successive tiles is ideal) to restrict the size of the configuration search space by orders of magnitude. We remark that the ML model makes the same hypothesis about intra-tile temporal reuse as our fully-associative model introduced in Section 2.3.

Li et al. [15] and Olivry et al. [20] push this approach even further. Their goal is to find a configuration (tile sizes and tiled loop order) which maximises the operational intensity. By considering reuse, they drastically prune the number of loop permutations to be considered, and they use a fully-associative cache model reasonning to output a symbolic expression of the number of cache misses, parametrized by the tile sizes. Then, they use a solver to find the permutation and sizes that optimize this expression. In this paper, we have replicated the core hypothesis used in these papers to build our fully-associative analytical cache model.

Narashimhan et al. [19] introduced a tile size selection model for fully-associative cache, which focuses on temporal and spatial reuse along each dimension.

Set-associative analytical cache modeling. Ghosh et al. [6] 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. [3] 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 occur, 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 as 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. [27], then Harper et al. [9, 10] consider 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. [12] 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. [16] in the context of iterative stencil operations.

Recently, De Albuquerque Silva et al. [4] computes the number of cache misses given a set-associative cache, for a GotoBLAS-style matrix multiplication. The tile sizes and problem sizes are symbolic, but the optimization scheme is fixed. Morelli et al. [17] also combines simulating and analytical cache modeling, by using the regularity of polyhedral computation and the symmetries in cache behavior to cover large portions of a simulation at once.

6 CONCLUSION

In this paper, we introduced a new set-associative analytical cache model which is specialized for tensor operations. This cache model is faster than usual set-associative models, while being more precise than fully-associative cache models. It is based on the notion of *detailed footprint*, and relies on an approximation when combining the contributions of all array accesses. We show that this approximation does not degrade much the ordering of configurations, which makes this model suitable for statically selecting configurations over a set.

From our observations, we do not recommend the use of a fully-associative analytical cache to model a set-associative cache model inside an optimizing compiler. Even if their analysis time scales much better, theirs results cannot be fully relied on. Thus, we have to use an analytical set-associative cache model, which is also an issue due to their analysis time. We believe that a promizing approach to solve this situation is the use of some well-chosen approximations in the design of a set-associative cache model, that does not degrade much the quality of the results while gaining in simplicity and analysis time. Our paper presents such an approximation, but is also restricted over a conceptually simple (and interesting) class of programs. A future goal would be to build a similar approximate set-associative analytical cache model for a larger class of programs.

ACKNOWLEDGMENTS

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 32 (dec 2017), 26 pages. https://doi.org/10.1145/3158120

[2] 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

[3] 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

[4] Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, Victor Jegu, and Claire Pagetti. 2024. A Predictable SIMD Library for GEMM Routines. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, IEEE, Hong Kong, Hong Kong SAR China, 55–67. https://doi.org/10.1109/RTAS61025.2024.00013

[5] Jan Edler and Mark D. Hill. 1999. Dinero IV Trace-Driven Uniprocessor Cache Simulator. https://pages.cs.wisc.edu/~markhill/DineroIV/ Last accessed 26 February 2024.

[6] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.* 21, 4 (jul 1999), 703–746. https://doi.org/10.1145/325478.325479

[7] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[8] 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* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 816–829. https://doi.org/10.1145/3314221.3314606

[9] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. 1997. *Predicting the cache miss ratio of loop-nested array references.* Report 336. University of Warwick. Department of Computer Science. https://wrap.warwick.ac.uk/61022/

[10] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. 1999. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Transaction on Computers* 48, 10 (Oct 1999), 1009–1024. https://doi.org/10.1109/12.805152

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

[12] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2016. Effective padding of multidimensional arrays to avoid cache conflict misses. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming*

Language Design and Implementation (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 129–144. https://doi.org/10.1145/2908080.2908123

[13] Chung-Hsing Hsu and Ulrich Kremer. 2004. A Quantitative Analysis of Tile Size Selection Algorithms. *The Journal of Supercomputing* 27 (03 2004), 279–294. https://doi.org/10.1023/B:SUPE.0000011388.54204.8e

[14] 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 (MASCOTS)*. 182–187. https://doi.org/10.1109/MASCOT.2003.1240655

[15] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P. Sadayappan. 2019. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, New York, NY, USA, 13 pages.

[16] Zhiyuan Li and Yonghong Song. 2004. Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems* 26, 6 (nov 2004), 975–1028. https://doi.org/10.1145/1034774.1034777

[17] 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/10.1145/3519939.3523714

[18] Phil Mucci, Shirley Moore, Christine Deane, and George Ho. 1999. PAPI: A Portable Interface to Hardware Performance Counters. , 8 pages. https://icl.utk.edu/papi/.

[19] Kumudha Narasimhan, Aravind Acharya, Abhinav Baid, and Uday Bondhugula. 2021. A practical tile size selection model for affine loop nests. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 27–39. https://doi.org/10.1145/3447818.3462213

[20] Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. 2021. 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3453483.3454103

[21] Arjun Pitchanathan, Kunwar Grover, and Tobias Grosser. 2024. Falcon: A Scalable Analytical Cache Model. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 222 (June 2024), 25 pages. https://doi.org/10.1145/3656452

[22] Louis-Noël Pouchet and Tomofumi Yuki. 2016. PolyBench/C: The polyhedral benchmark suite, version 4.2. http://polybench.sf.net.

[23] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. 2008. Positivity, posynomials and tile size selection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas) *(SC '08)*. IEEE Press, Article 55, 12 pages.

[24] V. Sarkar and N. Megiddo. 2000. 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)*. IEEE Computer Society, USA, 146–153.

[25] Nilesh Rajendra Shah, Ashitabh Misra, Antoine Miné, Rakesh Venkat, and Ramakrishna Upadrasta. 2022. BullsEye: Scalable and Accurate Approximation Framework for Cache Miss Calculation. *ACM Transactions on Architecture and Code Optimization* 20, 1, Article 2 (Nov. 2022), 28 pages. https://doi.org/10.1145/3558003

[26] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction* (Tallinn, Estonia) *(CC'12)*. Springer-Verlag, Berlin, Heidelberg, 101–121. https://doi.org/10.1007/978-3-642-28652-0_6

[27] O. Temam, C. Fricker, and W. Jalby. 1994. Cache interference phenomena. *SIGMETRICS Performance Evaluation Review* 22, 1 (may 1994), 261–271. https://doi.org/10.1145/183019.183047

[28] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. 2023. Autotuning Convolutions Is Easier Than You Think. *ACM Transactions on Architecture and Code Optimization* 20, 2, Article 20 (mar 2023), 24 pages. https://doi.org/10.1145/3570641

[29] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3, Article 14 (June 2015), 33 pages.

[30] Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*. Springer, Kobe, Japan, 299–302.

[31] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communication of the ACM* 52, 4 (2009), 65–76.

[32] Jingling Xue. 2000. *Loop tiling for parallelism*. Kluwer Academic Publishers, USA.