Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses

Changwan Hong[1] Wenlei Bao[1] Albert Cohen[2] Sriram Krishnamoorthy[3]
Louis-Noël Pouchet[1] Fabrice Rastello[4] J. Ramanujam[5] P. Sadayappan[1]

[1] The Ohio State University, USA, {hong.589,bao.79}@osu.edu, {pouchet,saday}@cse.ohio-state.edu
[2] DI, Inria and École Normale Supérieure, France, albert.cohen@inria.fr
[3] Pacific Northwest National Laboratory, USA, sriram@pnnl.gov
[4] Inria, France, fabrice.rastello@inria.fr
[5] Louisiana State University, USA, jxr@cct.lsu.edu

Abstract

Caches are used to significantly improve performance. Even with high degrees of set associativity, the number of accessed data elements mapping to the same set in a cache can easily exceed the degree of associativity. This can cause conflict misses and lower performance, even if the working set is much smaller than cache capacity. Array padding (increasing the size of array dimensions) is a well-known optimization technique that can reduce conflict misses. In this paper, we develop the first algorithms for optimal padding of arrays aimed at a set-associative cache for arbitrary tile sizes. In addition, we develop the first solution to padding for nested tiles and multi-level caches. Experimental results with multiple benchmarks demonstrate a significant performance improvement from padding.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, compilers, optimization

General Terms Algorithms, Performance

Keywords Array padding, conflict misses, direct-mapped cache, set-associative cache, tiling

1. Introduction

Array padding is a well-known performance optimization technique widely used in practice. A common scenario for using array padding is in computations, such as multidimensional fast Fourier transform (FFT) [8, 13, 15] and alternating direction implicit (ADI) solvers [6, 19], where repeated access of data values of different directions along a multidimensional array is required. Often, the multidimensional arrays are a power of two in size, causing high power-of-two access strides in memory. In turn, this can result in occupation of only a small subset of the available sets in a set-associative cache. Even with high degrees of set associativity, the number of accessed elements mapping to the same set can easily exceed the degree of associativity, causing conflict misses and significantly reduced performance. This can occur even if the working set is much smaller than cache capacity.

Fig. 1(a) illustrates the padding issue on a simple loop nest to symmetrize a square matrix of double floating-point numbers, an operation commonly performed in quantum chemistry. The result matrix `B` is a symmetrized form of the input matrix `A`, defined as the average of `A` and its transpose: $\texttt{B}[i][j] = \texttt{B}[j][i] = (\texttt{A}[i][j] + \texttt{A}[j][i])/2$. The computation of each row of `B` requires access to the corresponding row and column of `A`.

Consider an 8-way 32KB set-associative cache with 32 KB and a line size of 64 bytes. The cache has 64 sets, each with 8 lines. Using the code from Fig. 1(a), assume the array origins are aligned to cache line boundaries. Without loss of generality, assume that `A[0][0]` maps to cache set 0 (if `A[0][0]` maps to some other set S_0, all set mappings will just shift by a fixed amount, modulo 64, and all conflict miss counts will remain identical). Fig. 1(c) shows the elements mapping to cache sets. With the row-major array linearization in `C` (because the two-dimensional (2D) array `A` has 128 elements in each row) and as each 64-byte cache line holds 8 elements, the 128 elements in the first row of `A` will map to consecutive cache sets 0, 1, ..., 15. `A[1][0]` will map to cache set 16, `A[2][0]` to cache set 32, `A[3][0]` to set 48, and `A[4][0]` back to set 0. Thus, every fourth element in a column will map to

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908123

set 0. When the entire column of 128 elements is repeatedly accessed, 32 of them will map to cache set 0. Thus, despite 8-way set associativity, cache lines containing earlier elements in A will get evicted when later elements in the column are accessed. As a result, each access along the columns of A will result in a cache miss. For example, the cache line containing A[0][0], A[0][1], ..., A[0][7] is brought in when row 0 of B is computed. However, when the next row of B is computed, element A[0][1] will no longer be in cache because the needed cache line will have been evicted earlier by conflict misses.

Fig. 1(d) shows mapping of the elements in column 0 to cache sets when A is padded by 8 dummy columns and declared A[128][136] instead of A[128][128]. Only the subset of array locations A[0 : 127][0 : 127] actually gets used, while the set of elements A[0 : 127][128 : 135] is never initialized or used. The dummy array columns' key benefit is to change the element-to-set mapping in the cache. A[1][0] now maps to cache set 17, A[2][0] to cache set 34, A[3][0] to set 51, and A[4][0] to set 4. Every adjacent pair of elements in a column now maps to sets that are 17 apart, modulo 64. Because 17 and 64 are relatively prime, each element from A[0][0] to A[0][63] maps to a distinct cache set until A[0][64] again maps to set 0. Exactly 4 elements out of the 128 elements in array column 0 map to each cache set, and no evictions occur.

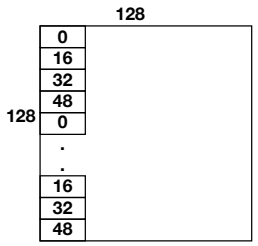
Fig. 1(b) shows the performance impact of padding for this simple example. The symmetrizer accelerates by more than 250% on two different Intel processors, while the number of L3 cache misses drops by more than 70%.

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  B[i][j] =
   0.5 *
   (A[i][j]+A[j][i]);
```

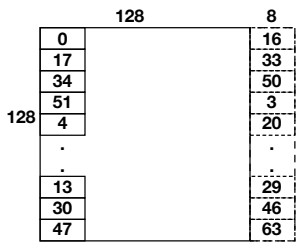
(a) Symmetrization

	Size: N	Padsize: P	Time: ms
Sandy Bridge	$N = 2k$	$P = 0$	46.33
	$N = 2k$	$P = 8$	10.80
	$N = 4k$	$P = 0$	204.60
	$N = 4k$	$P = 8$	66.12
Haswell	$N = 2k$	$P = 0$	47.50
	$N = 2k$	$P = 8$	11.33
	$N = 4k$	$P = 0$	193.90
	$N = 4k$	$P = 8$	68.45

(b) Performance Table



(c) Map without padding



(d) Map with padding

Figure 1: 2D Conflict Miss Example

In the preceding example, padding the array to hold 8 extra dummy columns (one cache line width) enables complete elimination of conflict misses for column-wise data access. In general, conflict misses can be detrimental when an application code exhibits reuse within a working set that is smaller than cache capacity, but too many elements in the working set (more than the cache associativity) map to the same cache set in a set-associative cache. A common scenario involves tiled execution, where data is reused within a tile. However, the collection of data accessed in the tile, i.e., the tile's data footprint, is not contiguous in memory. Multidimensional arrays naturally have power-of-two extents in many scientific applications, e.g., with multidimensional FFTs and adaptive mesh refinement where the coarsening/partitioning factor along each spatial dimension is typically 2. Tiled execution in such cases often results in conflict misses within the data footprint of tiles. Padding the arrays can alleviate or even completely eliminate conflict misses. The problem we address in this paper is:

> Given a set of multidimensional arrays and a multidimensional hyperrectangular data footprint for each array, can padding extents for arrays and inter-array spacing be found that completely eliminate conflict misses in a hierarchy of set-associative caches while minimizing the space overhead from the padding itself?

Until now, heuristics have been employed to determine how much array padding to use because no complete solution is known. In this paper, we develop and describe a comprehensive solution to the problem.

- We develop an analytical solution to the problem of optimal padding of arrays for a set-associative cache with necessary and sufficient conditions for avoiding conflict misses using full-capacity tiles—the tile's data footprint fully uses the entire cache capacity.
- We develop an efficient computational solution for the optimal padding of one or more arrays for an A-way set-associative cache for arbitrary tile sizes.
- We develop the first solution to padding for nested tiles and multi-level caches.
- We implement these padding algorithms in a new tool called *PAdvisor* and demonstrate its effectiveness on the co-tuning for optimal tile sizes and array padding extents.
- We present experimental results with multiple benchmarks, demonstrating significant performance improvement using *PAdvisor*.

2. Background and Related Work

Array padding is widely used for the important and commonly occurring case of data arrays with power-of-two sizes. However, this topic has only been sparsely addressed by the compiler community, and application developers resort to heuristics or experimental auto-tuning to find good values for padding. In this section, we review prior work on the padding problem.

Heuristic Approaches Bacon et al. [4] propose array padding as a method to handle conflict misses. Their work addresses intra-array padding to eliminate conflict misses

between two references in the context of a single loop (or the innermost loop in a loop nest). It does not handle tiling. In the context of embedded systems, Panda et. al. [18] handle interference misses in array tiles by enumerating different padding values and performing cache simulation for each padding value to record misses. Kowarschi et al. [16] present a review of cache optimization techniques for numerical methods. Other work [10, 14] employs padding to optimize codes. Rivera and Tseng [21] show that new transformations are needed for partial differential equations (PDEs) in three dimensions (3D) and that tile sizes must be chosen to avoid conflict misses along with the padding of arrays. They present heuristics and cost models for padding, but their solution does not guarantee elimination of conflict misses.

Using Cache Miss Equations In their work, Ghosh et al. [9] develop a general methodology for modeling cache misses (cold, capacity, and conflict misses) for affine perfectly nested loop computations. Using this framework, they have created an approach for determining padding extents for multidimensional arrays to eliminate conflict misses. Their approach involves numerical approximation to find solutions to cache miss equations. Again, it does not guarantee optimality in sizing the padded arrays.

Footprint-Based Optimal Padding In the case of direct-mapped caches, Li and Song [17] have developed a padding scheme to remove conflict misses for a tile size whose data footprint equals the cache capacity. They offer conditions under which multidimensional array tiles are conflict-free while fully utilizing the cache and find the minimum padding values that satisfy these conditions. Their solution assumes that the cache size equals the product of tiles sizes along different dimensions. In contrast, our analytical and computational solutions handle set-associative caches, and the computational solution handles arbitrary tile sizes. Furthermore, we present sufficient conditions for hierarchical tiling, which is not addressed in [17].

3. Analytical Solution: Divisible Tile Sizes

We first define the notation used in the paper and assumptions about the cache hierarchy. Whenever possible, these notations are compatible with those of Li and Song [17].

We study the optimal padding of a single d-dimensional array of some scalar element type. N_i denotes the number of elements along dimension i, for $1 \leq i \leq d$, with N_1 representing the extent along the fastest varying dimension and N_d the extent along the slowest varying dimension in the linearized layout of the array, i.e., the innermost dimension for row-major order (C and C++) and outermost dimension for column-major order (Matlab and Fortran). The padded extent $N_i = M_i + P_i$ is the sum of the number of accessible elements M_i and the amount of padding P_i at dimension i.

We consider a tiled loop nest operating over such arrays, and no restriction is made over the structure and iteration schedule at these nests. Without loss of generality, we assume the footprint of a given tile is d-dimensional in every array it accesses. Let D_i be the size of the tile footprint at dimension i. It can take any value between 1 and M_i. Because the granularity of data movement for caches is a cache line, the tile size along the fastest varying dimension D_1 is always assumed to be a multiple of the cache block size B.

We also model hierarchical tiling, aiming for the absence of conflicts at each nested tile in the corresponding caches in a multilevel hierarchy. We assume the footprints of inner nested tiles are perfectly aligned within those of outer tiles, so a collection of inner tiles precisely covers an outer tile's footprint. We show that only two levels of tiling need to be considered at a time, e.g., let D'_i denote the size of dimension i in the enclosing tiles with $D_i \leq D'_i$ for all $1 \leq i \leq d$.

It is possible to generalize this formalization to arrays and tiles of different and non-homogeneous dimensions and shapes, yet it is done without the guarantee of a consistent padding strategy across all arrays and tiling levels.

The cache hierarchy itself has multiple levels and is seen from the point of view of a single processor core. Let C_ℓ denote the capacity of the cache at level $\ell \geq 1$ following the usual top-down numbering. We assume $C_\ell \leq C_{\ell+1}$ and an identical line/block size B at every level. The latter hypothesis is not a fundamental restriction and is meant to improve readability. To simplify the notations, we also express B as a number of scalar elements rather than bytes. We write $C_\ell = S_\ell A_\ell B$, where A_ℓ and S_ℓ are the set associativity and number of sets at level ℓ, respectively.

The complexity of the analytical padding solution developed in this section does not depend on the size of the arrays. The analytical solution relies on one important restriction: for conflict-free padding at cache level j, $\prod_{1\leq i\leq d} D_i$ must divide C_ℓ. For hierarchical tiling with an additional lower cache level $j' > j$, $\prod_{1\leq i\leq d} D'_i$ must divide $C_{\ell'}$.

The restriction means that the tile footprint divides cache capacity. This apparently ad hoc constraint actually is the key to a chain of simplifications that enables an analytical solution for finding memory-optimal conflict-free padding.

In the next section, the restriction will be lifted thanks to a more expensive—nevertheless extremely efficient—computational solution to the optimal padding problem.

Note: when working on a single cache level at a time, we will drop the j subscript from these cache parameters.

3.1 Padding for Direct-mapped Caches

First, recall the case of direct-mapped caches:

Theorem 1 (Direct-mapped cache)**.** *Consider a direct-mapped cache of capacity $C = SB$. A loop nest whose tiles have a d-dimensional array footprint can fully utilize the cache and remain free of self-interference if and only if the following conditions are met:*

1. $\forall i, 1 \leq i \leq d$, *$D_i$ divides N_i.*
2. $\forall i, 1 \leq i \leq d-1, \gcd(C/\prod_{1\leq k\leq i} D_k, N_i/D_i) = 1$.

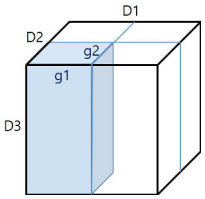
Proof. This is proven by Li and Song (see pp. 24–25 in [17]). We recall the proof argument for further generalization to the set-associative case, starting with the second condition.

For $d = 2$, the idea consists in partitioning cache lines into chunks of consecutive lines of size D_1/B. One may reason up to the tile's realignment to a chunk boundary. Each row of the tile touches exactly one chunk. There are S lines. Hence, $SB/D_1 = C/D_1$ chunks. The function mapping tile rows to chunks touches all of them if and only if N_1/D_1 is a generator of the $(\mathbb{Z}/(C/D_1)\mathbb{Z}, \cdot)$ group, i.e., if and only if $\gcd(C/D_1, N_1/D_1) = 1$.

For $d > 2$, Li and Song only state that the proof idea can be inductively applied to the general case ([17], Lemma 5.2, pp. 25). Here, we provide a proof sketch. Chunks may be reindexed by multiplying them by the inverse of N_1/D_1 modulo S/D_1. As such, the reindexed chunks associated to a tile plane are made consecutive and may form "superchunks" of size D_1D_2. The function mapping tile planes to superchunks touches all of them if and only if $N_2/D_2 \cdot N_1/D_1 \cdot (N_1/D_1)^{-1} = N_2/D_2$ generates the $(\mathbb{Z}/(C/(D_1D_2))\mathbb{Z}, \cdot)$ group. Ongoing from dimension i to $i+1$, chunks may be reindexed to make them consecutive and form higher-dimensional superchunks of size $D_1 \cdots D_{i+1}$. Those superchunks touch all cache lines if and only if the theorem's second condition holds.

Per the first condition, if D_i did not divide N_i for some i, the chunks would not span full cache lines, wasting cache capacity, which contradicts the hypothesis. Conversely, if D_i divides N_i for all i, the construction enabled by the second condition guarantees that all lines are fully used. □

3.2 Padding for Set-associative Caches



To extend this result to the set-associative case for all $i, 1 \leq i \leq d-1$, we introduce the characteristic number g_i of dimension i with respect to the cache size. Intuitively, as depicted (square at right) for $A = 4$, we will establish that if the enclosed tile $g_1 \times \cdots \times g_{d-1} \times D_d$ is free of self-interference conflicts in a direct-mapped cache, then the A-times larger tile $D_1D_2 \cdots D_d$ is free of self-interference conflicts in an A-associative cache of the same capacity.

Theorem 2 (Associative cache)**.** *Consider a set-associative cache of capacity $C = SAB$. For all $1 \leq i \leq d-1$, let $g_i = \gcd(S/\prod_{1\leq k\leq i-1} g_k, N_i)$. A loop nest whose tiles have a d-dimensional array footprint can fully utilize the cache and remain free of self-interference if and only if the following conditions are met:*

1. *$\forall i, 1 \leq i \leq d-1$, $\exists j, 1 \leq j \leq i$, $\prod_{1\leq k\leq i} g_k$ divides $D_j \prod_{1\leq i\leq j-1} g_i$.*
2. *$\exists i, 1 \leq i \leq d$, S divides $D_i \prod_{1\leq k\leq i-1} g_k$.*

Proof. We state two key observations underlying the proof, which is detailed in the appendix.

The reasoning of the direct-mapped case can be adapted to where chunks do not occupy disjoint cache sets but when, at most, A of them hit a given set instead. Such overlap will be tolerated through set associativity. The reasoning of the direct-mapped case extends to the case where exactly A chunks hit the same set, each one being aligned on a cache line boundary. This means the stride between chunks can be any integer dividing the set size (e.g., C/g_1 when $d = 2$) and greater than or equal to the set size divided by A (e.g., $C/(g_1D_1)$ when $d = 2$). This leads to the interval of possible values for a given gcd g_i in the second condition.

The case of $g_i = D_i$ hits each set exactly once on a stripe of rows in the tile footprint before hitting every set again in the next stripe. The case of $g_i = D_i/A$ matches the indexing of the direct-mapped case with each consecutive chunk in the tile footprint hitting a different set A times. Intuitively, the lower the g_i, the more associativity is "consumed" by sub-tiles of dimension i (rows, planes, etc.), leaving less conflicts to be tolerated at higher dimensions. This observation underlines the second condition. □

Note 1: the necessary condition establishes that the minimal padding satisfying the hypotheses of Theorem 2 is the optimal one that avoids self-interference conflicts in the general case of set-associative caches.

Note 2: as a side effect, the second condition eliminates degenerate cases where the tile footprint would be so small that all of its conflicts could be tolerated by associativity.

3.3 Padding for Tile Hierarchies

We now extend this result to hierarchically tiled loop nests. We focus on two nested tiles, following the notations introduced earlier in this section. We note that the previous padding approach of Li and Song [17] only models direct-mapped caches, and with that model, surprisingly, no conflict-free padding for nested tiles is feasible.

We use a simple example here to explain why. Meanwhile, a formal statement about the infeasibility of nested tiling for conflict-free padding under a direct-mapped cache model is stated and proven in the associated report [11].

Example Consider cache lines of 64 bytes. Let S_1 be 512 lines for a 32 KB L1 cache and S_2 be 4096 lines for a 256 KB L2 cache. Finally, select $M_1 = 1024$ doubles, i.e., 128 cache lines and $D_1 = 8$ and $D_2 = 32$.
For a conflict-free tile in L1, N_1 can be $1024+8$, $1024+24$, $1024+40$, $1024+56$, etc., (i.e., $1024+8(2k+1)$), because $\gcd(1024+8(2k+1), 4096) = 8$.
For a conflict-free tile in L2, N_1 can be $1024+32$, $1024+96$, $1024+160$, $1024+224$, etc., (i.e., $1024+32(2k+1)$), because $\gcd(1024+32(2k+1), 4096) = 32$.

Clearly, there are no common values for the padded array that can be conflict-free for both direct-mapped caches. However, when the caches are set-associative, we can develop padding solutions that enable interference-free access in multiple nested tiles within a cache hierarchy.

Theorem 3 (Hierarchical tiling, associative cache)**.** *Consider a high-level cache of capacity $C_\ell = S_\ell A_\ell B$ and a low-level cache of capacity $C_{\ell'} = S_{\ell'} A_{\ell'} B$. For all $1 \leq i \leq d-1$, let $g_i = \gcd(S_\ell / \prod_{1 \leq k \leq i-1} g_k, N_i)$ and $g'_i = \gcd(S_{\ell'} / \prod_{1 \leq k \leq i-1} g'_k, N_i)$. For both inner and enclosing tiles to fully utilize their respective caches levels and remain free of self-interference, it is sufficient that the following conditions are met:*

1. $\forall i, 1 \leq i \leq d-1$, $\exists j, 1 \leq j \leq i$, $\prod_{1 \leq k \leq i} g_k$ *divides* $D_j \prod_{1 \leq i \leq j-1} g_i$.
2. $\forall i, 1 \leq i \leq d-1$, $\exists j, 1 \leq j \leq i$, $\prod_{1 \leq k \leq i} g'_k$ *divides* $D'_j \prod_{1 \leq i \leq j-1} g'_i$.
3. $\exists k, 1 \leq k \leq d$, S_ℓ *divides* $D_k \prod_{1 \leq i \leq k-1} g_i$.
4. $\exists k, 1 \leq k \leq d$, $S_{\ell'}$ *divides* $D_k \prod_{1 \leq i \leq k-1} g'_i$.
5. $\forall i, 1 \leq i \leq d-1, \prod_{1 \leq k \leq i} D'_k$ *divides* $A_{\ell'} \prod_{1 \leq k \leq i} D_k$.

In addition, the first four conditions establish a necessary condition for both tiles to fully utilize their respective cache levels and remain free of self-interference.

Proof. The first four conditions are simply the conjunction of ones established for a single level of tiling.

The fifth condition states the footprint of the first i dimensions of the enclosing tile cannot be more than $A_{\ell'}$ times larger than the first i dimensions of the inner tile. One may then iterate along dimension $i+1$, spanning a whole $i+1$-dimensional slice of the enclosing tile without exceeding the associativity of the larger, lower-level j' cache. □

The theorem for hierarchical tiling is only a sufficient condition because specific ratios between the inner and enclosing tiles may not require the fifth condition.

Nevertheless, it is important to note that in the direct-mapped case, the fact the first four conditions alone are not sufficient proves the *impossibility of compatible paddings for two nested levels of tiling* if the lower-level cache is not sufficiently associative. This is a completely new result, inaccessible to Li and Song [17]. It also carries a concrete message for cache architects and for applying loop tiling in compilers or domain-specific frameworks: it is essential to keep the cache size [1] and tile ratios below the associativity of the lower, larger cache. This result also pushes for higher associativity as the cache hierarchy grows taller.

4. Padding For Arbitrary Tile Sizes

The previous section addressed padding for "divisible" tiles, where the cache capacity is divisible by the tile data footprint. However, this may not always be feasible. For example, consider a computation that uses three data arrays and identically sized tiles for the them. For any power-of-two cache capacity, it is impossible to satisfy the divisibility condition without making the tile unnecessarily small and wasting cache capacity. As another constraint, some tiled algorithms may be constrained to using "square" tiles, i.e., tile sizes along all dimensions must be equal. Hence, the total cache capacity may not be a perfect square or cube.

In this section, we address the more general padding problem, where the cache capacity is not constrained to be divisible by the tile data footprint. Given a 2D (resp. 3D) array of size $M_2 M_1$ (resp. $M_3 M_2 M_1$) and an arbitrarily sized data tile $D_2 D_1$ (resp. $D_3 D_2 D_1$) such that the tile data footprint is less than the cache capacity, we seek to find minimal padding extent(s) P_1 (resp. P_2, P_1) that guarantee conflict-freedom within the data tile. While the developed approach can be extended to higher dimensions, our current implementation in *PAdvisor* only handles 2D and 3D arrays. We present details for the 2D case in the paper, while details for the 3D case are provided in the associated report [11].

Before presenting the algorithms for finding optimal conflict-free padding, we first address the question: is it always feasible to achieve conflict-free padding for any arbitrary tile size as long as the total tile data footprint is no greater than cache capacity? The answer to this question is positive and is stated in the following lemma:

Lemma 1. *For an arbitrary data tile with footprint less than or equal to cache capacity, there always exists some padding that makes the tile conflict-free.*

Proof. Consider a d-dimensional tile in a d-dimensional array. The cause of conflict misses is a non-uniform mapping of tile elements to cache sets. The following constructive scheme for padded extents along the lowest $d-1$ dimensions avoids such non-uniformity: $\forall 1 \leq i \leq d-1, \overline{N_{iS}} = \overline{D_{iS}}$ ($\equiv N_i \mod S \equiv D_i \mod S$). Such a padding ensures that consecutive tile rows and tile planes map to cache sets in exactly the same way they would if the data array was the same size as the data tile, i.e., blocks in the array are mapped lexicographically to consecutive cache sets. Thus, occupancy of no cache set can exceed the associativity. □

We first present the solution for direct-mapped caches, forming the framework basis for its generalization to set-associative caches in Sec. 4.2.

4.1 Computational Scheme for Direct-mapped Caches

In contrast to the analytical approach presented in the previous section, the approach developed in this section uses an explicit enumeration process.

4.1.1 2D Data Space

Given an arbitrary 2D data tile of size $D_2 D_1$, we seek the smallest value of P_1 so that a given padded array `A` of size $M_2(M_1 + P_1)$ is conflict-free. We first explain the approach for the direct-mapped case. The essential idea is to systematically proceed to eliminate unsuitable values for P_1, i.e., values of P_1 that do not achieve freedom from conflict. Given an element $\mathtt{A}[i_2][i_1]$, in the padded array, it maps to cache

[1] Or cache slice size for shared caches with parallel access ports.

set $(N_1 i_2 + i_1)/B \mod S$. We first observe that the possible range of values to be considered for P_1 is limited to S, the number of cache sets. This is because $(N_1 i_2 + i_1)/B \equiv ((N_1 + SB)i_2 + i_1)/B \mod S$. This means that the mapping of tile elements to cache sets is exactly the same for a padded array extent N_1 and a padded array extent $N_1 + SB$. Assuming that N_1 is chosen to be a multiple of cache size, there is no need to search over all possible N_1 values to determine the existence of a conflict-free solution. Checking only S padding values $(0, B, 2B, .., (S-1)B)$ is enough to find a conflict-free padding.

To simplify the explanation of the approach, consider the cache block size to be one (the detailed algorithm provided later does not impose such a constraint). A particular choice of padding value P_1 is unsuitable if any two data tile elements $\mathtt{A}[i_2][i_1]$ and $\mathtt{A}[j_2][j_1]$ that map to the same cache set under that padding exist. The essential idea behind the computational approach developed in this section is to perform "inverse" reasoning: consider all possible pairwise tile element conflicts, find padding choices that can cause each conflict, and eliminate all such padding choices. After all possible pairwise conflicts between data tile elements are considered, any remaining padding choices are guaranteed to provide conflict-freedom for the data tile, and the choice requiring the smallest space overhead is selected..

Given a $D_2 \times D_1$ tile, there are $D_2 D_1$ distinct data blocks and, therefore $(D_2 D_1 (D_2 D_1 - 1))/2$ possible cases to consider. However, we can reduce the number of potentially conflicting pairs to be considered because of the following observation (Lemma 2): if there is a conflict (noted $\sim$) between any two elements in a tile data space A, then there necessarily also is a conflict between the first data element $\mathtt{A}[0][0]$ and some other element in A or a conflict between $\mathtt{A}[0][D_1 - 1]$ and some other element. This simplifies and focuses our reasoning on just these two particular elements.

Lemma 2. *Consider a 2D tile data space $\mathtt{A}[i_2][i_1]$ such that $0 \leq i_2 < D_2$ and $0 \leq i_1 < D_1$. For all i_2, i_1 in the data space, there are no cache conflicts $\mathtt{A}[0][0] \sim \mathtt{A}[i_2][i_1]$ and $\mathtt{A}[0][D_1 - 1] \sim \mathtt{A}[i_2][i_1]$ if and only if the entire data space is conflict-free.*

Proof. **If case**: We prove by contradiction. If the cache is not conflict-free, then $\exists (i_2, i_1), (j_2, j_1)$ such that there is a conflict $(i_2, i_1) \sim (j_2, j_1)$ with $(i_2, i_1) \prec (j_2, j_1)$. We have two possibilities:
Case 1: $i_2 \leq j_2, i_1 < j_1$, $(N_1 i_2 + i_1)/B \equiv (N_1 j_2 + j_1)/B \mod S \iff (N_1(j_2 - i_2) + (j_1 - i_1))/B \equiv ((N_1 \cdot 0) + 0)/B \mod S$. Also, $0 \leq i_1, j_1 < D_1$ and $0 \leq i_2, j_2 < D_2$. Thus, $0 \leq j_2 - i_2 < D_2, 0 < j_1 - i_1 < D_1$, which means $\mathtt{A}[j_2 - i_2][j_1 - i_1]$ is in the data space. Therefore, $\mathtt{A}[0][0] \sim \mathtt{A}[j_2 - i_2][j_1 - i_1]$, contradiction.
Case 2: $i_2 < j_2, i_1 \geq j_1$, Similar to case 1, $(N_1 i_2 + i_1)/B \equiv (N_1 j_2 + j_1)/B \mod S \iff (N_1(j_2 - i_2) + (j_1 - i_1))/B \equiv ((N_1 \cdot 0) + 0)/B \mod S \iff (N_1(j_2 - i_2) + (D_1 - 1 + j_1 - i_1))/B \equiv ((N_1 \cdot 0) + D_1 - 1)/B \mod S$. Also, $0 \leq i_2, j_2 < D_2$, and $0 \leq i_1, j_1 < D_1$. Thus, $0 \leq j_2 - i_2 < D_2, 0 \leq D_1 - 1 + j_1 - i_1 < D_1$, which means $\mathtt{A}[j_2 - i_2][D_1 - 1 + j_1 - i_1]$ is in the data space. Therefore, $\mathtt{A}[0][D_1 - 1] \sim \mathtt{A}[j_2 - i_2][D_1 - 1 + j_1 - i_1]$, contradiction.
Only if: This is true by definition, since no pair of elements can be in conflict in a conflict-free data space. □

A consequence of Lemma 2 is that checking for the absence of conflicts $\mathtt{A}[0][0] \sim \mathtt{A}[i_2][i_1]$ and $\mathtt{A}[0][D_1 - 1] \sim \mathtt{A}[i_2][i_1]$ is enough to ensure the entire tile data space is conflict-free. There is no need to check all pairs of points in the data tile.

The previous condition for checking on absence of conflicts for the top left and right corners of a 2D tile can be equivalently stated in terms of additional tests for the top left corner of the tile. Lemma 3 presents the necessary and sufficient conditions.

Lemma 3. *Given a 2D array $\mathtt{A}[*][N_1]$ with padded size N_1, the tile data space $\mathtt{A}[i_2][i_1]$, $0 \leq i_2 < D_2$, $0 \leq i_1 < D_1$, is conflict-free if and only if $(N_1 i_2 + i_1)/B \mod S \neq 0$, $\forall i_2, i_1$ such that $0 \leq i_2 < D_2, -D_1 < i_1 < D_1, i_1 \equiv 0 \mod B$ and $(i_2, i_1) \neq (0, 0)$.*

Proof. There is no cache conflict $\mathtt{A}[0][0] \sim \mathtt{A}[i_2][i_1]$ if and only if $(N_1 i_2 + i_1)/B \not\equiv 0 \mod S$ for all $0 \leq i_2 < D_2, 0 < i_1 < D_1$. There is no cache conflict $\mathtt{A}[0][D_1 - 1] \sim \mathtt{A}[i_2][i_1]$ if and only if $(N_1 i_2 + i_1 - (D_1 - 1))/B \not\equiv 0 \mod S$ for all $0 \leq i_2 < D_2, 0 \leq i_1 < D_1$. Also, $(N_1 i_2 + i_1 - (D_1 - 1))/B \not\equiv \mod S$ for all $0 < i_2 < D_2, 0 \leq i_1 < D_1 \iff (N_1 i_2 + i_1)/B \not\equiv 0 \mod S$ for all $0 < i_2 < D_2, -(D_1 - 1) \leq i_1 < 0$. So, $(N_1 i_2 + i_1)/B \not\equiv 0 \mod S$ for all $0 \leq i_2 < D_2, -D_1 < i_1 < D_1$, and by Lemma 2, this proves Lemma 3. □

Before presenting the algorithm to compute optimal padding for 2D tiles, we use a simple example to illustrate the approach. Consider a direct-mapped cache with S=10, B=1, a 2D array of size 10×10, and a 3×3 data tile. Because the array extent in the fastest varying dimension is 100, a multiple of S, every element in a column of the tile will map to the same set, causing conflict misses. As already observed, the possible padding values to be considered are from 0 to 9. By Lemma 3, for a padded size N_1 to make the data tile conflict-free, we should have:
$(N_1 i_2 + i_1) \not\equiv 0 \mod S, \forall i_2, i_1$ such that $0 \leq i_2 < 3, -3 < i_1 < 3$ and $(i_2, i_1) \neq (0, 0)$.
The preceding condition can be visualized in Fig. 2, which requires that none of the shown vectors should be "conflict vectors" with respect to (0,0), i.e., none of the target elements at the sink of the vectors should map to cache set 0. For each such vector, a Diophantine equation determines the values of N_1, if any, for which the condition is violated.

For example, considering $(i_2, i_1) = (1, 1)$, the equation $(N_1 \cdot 1) + 1 \equiv 0 \mod 10$ has solutions 9, 19, 29...etc. Any padded extent N_1 equal to 9 modulo 10 would cause a conflict between tile elements $(0, 0)$ and $(1, 1)$. As shown in Fig. 2, the value 9 is crossed off as unsuitable in the space of possible values. Similarly, considering $(i_2, i_1) = (1, -1)$, which actually corresponds to checking for a conflict with the top right corner tile element $(0, 2)$, we get the equation $(N_1 \cdot 1) - 1 \equiv 0 \mod 10$, with solutions 1, 11, 21...etc. This results in crossing off the entry for 1 in the space of possible padding values in Fig. 2. The figure shows all such "conflict vectors" evaluated and the padding value they eliminate. Some conflict vectors produce no solutions to the corresponding Diophantine equation, for example, (2,1). The equation $(N_1 \cdot 2) + 1 \equiv 0 \mod 10$ clearly has no integer solutions. Such pairs of data elements do not have conflicts for any possible padding value and, therefore, do not eliminate any options.

After eliminating all unsuitable padding values corresponding to all possible conflict vectors in the range $0 \leq i_2 < 3, -3 < i_1 < 3$ and $(i_2, i_1) \neq (0, 0)$, any remaining values are all suitable candidates for padding (modulo 10) that ensure freedom from conflicts for the data tile. For this example, the result is that a conflict-free padded extent must have a remainder of either 3 or 7 when divided by 10. The padding value that results in the least space overhead is chosen. For an array of extent 100, a padded size of 103 would be the best choice among the possible options of $100+3 \equiv 3 \mod 10$ and $100 + 7 \equiv 7 \mod 10$. If the unpadded array happened to be of size 106, the best padded choice would be $106+1 \equiv 7 \mod 10$, which is better than the other possible conflict-free choice of $106 + 7 \equiv 3 \mod 10$.

The Algorithm Alg. 1 depicts the algorithm for finding conflict-free padding. It explores a set of points (i_2, i_1) in the data space for which the modulo property is verified, per Lemma 3. Instead of formulating and solving a separate Diophantine equation for each possible conflict vector, acceleration of the execution time is achieved by a pre-computation of the inverse modulo. Given $x \in \mathbb{Z}_n^*$, there exists a unique element $y \in Z_n^*$ s.t. $xy \equiv 1 \mod n$. y is called the inverse of x, written x^{-1}, and can be computed by the extended Euclidean algorithm with a time complexity of $\mathcal{O}(n \log n)$. The algorithm proceeds by enumerating the necessary points in the data space, checking the condition of Lemma 3 to find and mark off all unsuitable padding values in the $PadOk$ array. Then, the minimal padding is obtained from this array from among those entries that have not been eliminated. As the total number of blocks in the tile data space is, at most, the number of sets in the cache, an inverse modulo operation of complexity $\log S$ is performed S times. Therefore, computational complexity is $\mathcal{O}(S \log S)$.

Algorithm 1 2D padding, single array, direct-mapped cache

Input: S (number of cache sets), D_2, D_1 (tile sizes), M_1 (unpadded array extent)
Output: Minimal Padding Size P_1

```
1:  // Initially consider all padding values as OK
2:  PadOk[S] ← 1
3:  // For each (i2, i1) clear PadOk for any padding values that create
    conflict between (0, 0) and (i2, i1)
4:  for i2 = 0 to D2 − 1 do
5:      c ← gcd(i2, S)
6:      inv ← (i2/c)^−1 mod (S/c)
7:      for i1 = −(D1 + B)/B to (D1 − B)/B do
8:          if i1 mod c = 0 then
9:              for i0 = 0 to c − 1 do
10:                 v ← (−i1 · inv) mod (S/c)
11:                 PadOk[(v + i0(S/c))] ← 0
12:             end for
13:         end if
14:     end for
15: end for
16: for i0 = 0 to S − 1 do
17:     if PadOk[(M1 + i0B) mod S] = 1 then
18:         return i0B        // Return P1
19:     end if
20: end for
21: return 0        // Return P1
```

4.1.2 3D Data Space

The extension of the previously described 2D padding algorithm to 3D data space is essentially a direct generalization. For 3D tiles, it is necessary and sufficient to analyze conflicts with respect to four corner tile elements (instead of two points for the 2D case). Lemma 4 is a generalization of Lemma 2 to 3D spaces and is proven in the associated report [11].

Lemma 4. *Let* $\mathtt{A}[i_3][i_2][i_1]$ *be a 3D tile data space, with* $0 \leq i_3 < D_3, 0 \leq i_2 < D_2, 0 \leq i_1 < D_1$*, with the additional constraint that* $i_1 \equiv 0 \mod B$*. For all* i_3, i_2, i_1 *in the data space, there is no cache conflict* $\mathtt{A}[0][0][0] \sim \mathtt{A}[i_3][i_2][i_1]$*,* $\mathtt{A}[0][0][D_1 - 1] \sim \mathtt{A}[i_3][i_2][i_1]$*,* $\mathtt{A}[0][D_2 - 1][0] \sim \mathtt{A}[i_3][i_2][i_1]$*, and* $\mathtt{A}[0][D_2-1][D_1-1] \sim \mathtt{A}[i_3][i_2][i_1]$ *if and only if the data space is conflict-free.*

Similarly, we can derive the central Lemma 5, which is the 3D analog of Lemma 3.

Lemma 5. *For* $\forall i_3, i_2, i_1$ *such that* $0 \leq i_3 < D_3, -D_2 < i_2 < D_2, -D_1 < i_1 < D_1, i_1 \equiv 0 \mod B$*,* $(i_3, i_2, i_1) \neq (0, 0, 0)$ *and given* N_2, N_1*, the data space is conflict-free if and only if* $(N_2 N_1 i_3 + N_1 i_2 + i_1)/B \not\equiv 0 \mod S$.

The proof is similar to the 2D case. It is available in the associated report [11].

The algorithm for the 3D case is similar to Alg. 1, exploring all necessary points (i_3, i_2, i_1) in the data space to eliminate unsuitable choices for conflict-free paddings. Instead of two corner tile elements in the 2D case, four corner elements in the top plane of the 3D tile must be checked for conflicts. However, there are two padding choices to be made for P_1 and P_2. For each P_2, starting with P_2=0 and incrementing

P_2 by 1, the algorithm proceeds by enumerating the necessary points in the data space to find all conflict-free padding values P_1, if any. Among the valid P_1 values for each P_2, the one requiring minimal storage overhead is identified. A globally optimal (P_2, P_1) pair is maintained and updated as different P_2 values are considered if a new conflict-free pair with lower space overhead is found.

For each value of P_2, the cost is $\mathcal{O}(S \log S)$, similar to the 2D case. There are S possible choices for P_2. Thus, the time complexity for the 3D case is $\mathcal{O}(S^2 \log S)$ applying the previously described, simple computation method. Nevertheless, additional optimizations via pre-computation can reduce the worst case complexity to $\mathcal{O}(S^2)$ and the average complexity to $\mathcal{O}(S \log S + SD_1/B)$. Details may be found in the associated report [11].

4.2 Computational Scheme for Set-associative Caches

The broad approach to computing padding for arbitrary tile sizes with set-associative caches is the same as that previously discussed for direct-mapped caches: scan the data tile space to identify padding values for which there is a conflict. Yet, there is a fundamental difference. While the existence of a conflict with any tile element is grounds for elimination of a padding choice, a more complex counting procedure must be used for set-associative caches because an A-way set-associative cache allows A conflicts at each set without needing to evict any data. Hence, we keep track of all conflict vectors for each possible padding value and only eliminate those that result in more than A conflicts.

4.2.1 2D Data Space

For clarity's sake, we start with the 2D space problem. Fig. 2 provides an intuition of the computation of conflict vectors (i.e., a data space location conflicting with either the top left or top right corner of the data space). Intuitively, the algorithm will proceed by keeping track of, for each possible padding value, the set of conflict vectors associated with it. For each possible conflict vector, a Diophantine equation's solution specifies the array padding extents (modulo S) for which such conflict vectors exist. In this example, a padding of 0 is associated with conflict vectors (1,0) and (2,0). A padding value of 3 or 7 has no conflict vectors.

As with the direct-mapped caches, we first reduce the set of data elements for which conflicts are analyzed. Whereas just the two top corner elements needed checking for the direct-mapped case, for the set-associative case all elements in the top row of a 2D tile must be checked for. This is formalized by the following lemma:

Lemma 6. *For a cache with associativity A, $\forall k$ a data point $A[0][k]$ has less than A conflicts with other points in the data space if and only if the data space is conflict-free.*

See the proof in the associated report [11].

Let us use an example to explain the computation of the optimal conflict-free padding for a set-associative cache.

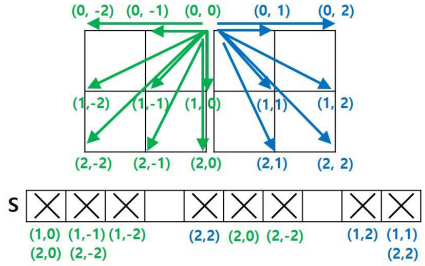


Figure 2: Conflict-free padding: direct-mapped cache

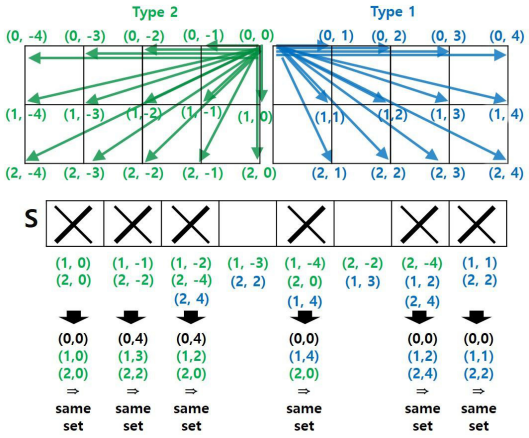


Figure 3: Conflict-free padding: set-associative cache

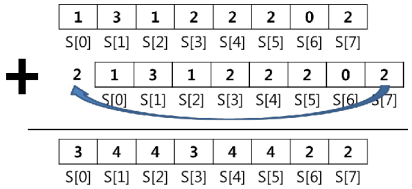


Figure 4: Inter-array padding

The example shown in Fig. 3 is similar to Fig. 2, which explains the computation of a conflict-free padding for a direct-mapped cache. Similarly in the set-associative case, the first step involves determination of padding values that cause conflicts for each point (i_2, i_1). In Fig. 3, we consider a cache with $S = 8$, $A = 2$, $B = 1$, $M_1 = 80$, $D_1 = 5$, and $D_2 = 3$. The data tile has a footprint of $3 \times 5 = 15$ blocks, and the cache capacity is 16 blocks. For each possible padding value in the range $\{0, \ldots, 7\}$, the index-pairs (i_2, i_1) that cause conflict with $(0, 0)$ are marked.

In contrast to the 2D case where simply the occurrence of a conflict eliminated a padding value from consideration, we need to count the number of conflicts for the set-associative case. Further, by Lemma 6, we must check for conflict counts w.r.t. all elements in the top row of the data tile; if any of these involve more conflicts than the cache associativity, the padding value is unsuitable.

For each of the S possible candidate values for padding, two groups of conflict vectors (i_2, i_1) are formed: Type 1 ($i_1 > 0$) and Type 2 ($i_1 \leq 0$). Type 1 conflict vectors imply that tile element $(0, 0)$ and (i_2, i_1) are mapped to the same set. Type 2 conflict vectors imply that the top right corner element in the tile $(0, 4)$ is mapped to the same set as (i_2, i_1).

For example, a padding value of 0 will yield two Type 1 conflict vectors $(1, 0)$ and $(2, 0)$. Including $(0, 0)$, there are three tile elements mapping to the same set and, therefore, cannot avoid conflict misses. Using a padding value of 1 yields two Type 2 conflict vectors $(1, -1)$ and $(2, -2)$, which means that the top right corner element $(0, 4)$ will conflict with $(0+1, 4-1)$ and $(0+2, 4-2)$, i.e., with $(1, 3)$ and $(2, 2)$. So padding by 1 cannot avoid conflict misses.

Next, consider a padding value of 3. We have a conflict vector of each type: a Type 1 vector $(2, 2)$ and a Type 2 vector $(1, -3)$. At the two corner tile points $(0, 0)$ and $(0, 4)$, only the Type 1 and Type 2 vectors, respectively, can cause conflict. However, all interior tile elements in the top row must be considered, i.e., $(0, 1)$, $(0, 2)$, and $(0, 3)$. For $(0, 1)$, we consider $(0 + 2, 1 + 2)$ and $(0 + 1, 2 - 3)$. Of these, $(2, 3)$ is within the data tile, but $(1, -1)$ is outside. Hence, there is only one conflicting data element. Similarly, it can be determined that $(0, 2)$ and $(0, 3)$ also have only one conflicting tile element. Therefore, a padding value of 3 results in conflict-free access for the data tile.

The algorithm for the 2D set-associative case is similar in structure to the previously described Alg. 1 for the direct-mapped case. The main difference is that conflict vectors are first stored for various padding values. Then, the elements of the top row of the data tile are tested for the number of conflicts (explained in the previous example). If less than A conflicts occur, the candidate padding value is valid. Finally, valid padding values are scanned to output the one with lowest space overhead. Each conflict vector (p, q) must be added to several padding candidates $S[i]$, such that $S[i]p + q = 0$. This process is repeated $\mathcal{O}(AS \log S)$ times, giving a time complexity of $\mathcal{O}(AS \log S)$. The testing of conflict counts for the top-row elements of the data tile has only an $\mathcal{O}(AS)$ cost. As such, the total complexity of the algorithm is $\mathcal{O}(AS \log S)$.

4.2.2 3D Data Space

The *PAdvisor* algorithm for 3D tiles and set-associative caches uses a combination of the approach previously described for the 3D direct-mapped case and the approach for handling associativity in 2D tiles.

4.3 Inter-array Padding

When multiple arrays are accessed in a tiled computation in an interleaved manner, the relative offsets of the array origins can clearly affect the number of cache misses due to inter-array interference in the cache. Therefore, even after padding each array to avoid cache conflicts, conflict misses could occur because of inter-array interference. Such interference may be avoided by suitably shifting array origins so that conflict misses stemming from previously interfering data elements by different arrays no longer cause conflict misses. Details are provided in the associated report [11]. Here, we use an example to explain the main idea behind the approach.

Fig. 2 illustrates an example with two arrays. No restrictions are imposed on either the array or tile sizes. First, for each array and its data tile footprint, padding analysis is performed per the algorithm presented in Sec. 4.2. In this example, we consider the same tile size for both arrays. The cache has 8 sets and a set-associativity of 4. Assume that for some choice of padding N_i, the padding analysis (identical for both arrays) shows interference counts of 1, 3, 1, 2, 2, 2, 0, and 2, for sets 0 through 7, respectively. This means that set 0 would have just one data block mapped to it while set 1 has three different data blocks in the data tile mapped to it for the chosen padding value.

If no inter-array padding is utilized, the total interference count from both arrays combined will double the single-array interference counts, resulting in interference counts of 2, 6, 2, 4, 4, 4, 0, and 4. This would be unsatisfactory as set 1 has 6 data blocks mapping to it but only 4 lines. If we shift the second array's origin by 1 cache line, the set interference counts for different cache sets would shift from the previous case (Fig. 2). After the inter-array shift, the accumulated interference counts for sets $\{0, \ldots, 7\}$ are 3, 4, 4, 3, 4, 4, 2, and 2. Now, no sets exceed their capacity of 4 cache lines, so the inter-array shift results in conflict-free mapping of the data tiles for both arrays.

The approach generalizes to multiple arrays without any restriction on the data footprints or array extents. Details are provided in the associated report [11].

4.4 Computational Complexity

When a divisibility relationship between the tile and cache sizes can be enforced, the very efficient analytical reasoning introduced in Sec. 3 applies. Table 1 summarizes the computational complexity of the padding algorithms for various other cases.

Table 1: Computational complexity: B=line size, S=number of sets, A=associativity

Type	Worst-case	Average
2D direct-mapped	$\mathcal{O}(S \log S)$	$\mathcal{O}(S \log S)$
2D set-associative	$\mathcal{O}(AS \log S)$	$\mathcal{O}(AS \log S)$
3D direct-mapped	$\mathcal{O}(S^2)$	$\theta(S \log S + SD_1/B)$
3D set-associative	$\mathcal{O}((AS)^2)$	$\theta(S \log S + SD_1/B + AS)$

The table shows worst- and average-case complexity for the algorithms. In practice, because of acceleration techniques, the average complexity can be lower than the worst-case complexity (shown in Table 1). Details for all algorithms and the complexity analysis are provided in the associated report [11]. The actual runtime of the most com-

plicated algorithm (3D set-associative) is reported later in Sec. 5, showing *PAdvisor* runs on the order of a few milliseconds for all benchmarks.

5. Experimental Evaluation

Padding is an essential optimization to avoid conflict misses, including when data are accessed along different directions of a multidimensional array. For example, the Intel Math Kernel Library Fast Fourier Transform routine, Intel MKL FFT [13], explicitly encourages padding by the user for best performance by separating out the description of the data layout from the FFT problem size in its interface, and it provides a tool to iteratively try various padding sizes for best performance [12]. Our work affords analytical solutions to the padding size search problem, and we now illustrate the impact of padding on several representative problems. They have been chosen to highlight the performance impact of padding in codes traversing data in different directions of a multidimensional array (Intel's MKL FFT and ADI), as well as the role of padding in tile size selection and its performance impact in multi-level tiling schemes on well-optimized codes (HPGMG, DGEMM, and Stencils).

Benchmarks We evaluate on six benchmarks, four of which can be tiled. For those, we also perform extensive tile size exploration.

ADI is an alternating direction implicit solver from PolyBench/C 4.1 [20] typically used to solve PDEs, we evaluate 20+ 2D problem sizes. For both MKL-FFT and ADI, the data access pattern combines row-first and column-first traversals of the data space, a stress case for conflict misses. We evaluate 13 different 2D problem sizes.

HPGMG is a High Performance Geometric Multigrid benchmark from DOE [25] to proxy full applications using adaptive mesh refinement. Multigrid solvers typically imply a division/multiplication by 2 of the box size (data space) computed on a processor. As such, domain decomposition into boxes typically uses power-of-two box sizes. We evaluate on the most time-consuming part of the application, a Chebychev smoother implementing a 3D stencil with four time iterations on which we implemented parametric time-tiling.

DGEMM is a classical BLAS3 routine implemented in C using parametric tiling and code massaging to ensure good AVX/AVX2 vectorization by the compiler. Tiles are scanned in the classical i, j, k order, but within a tile, we permuted the loops to k, i, j for efficient vectorization and data reuse. In our experiments, we cannot use the equivalent BLAS functions from Intel MKL: the tiling / tile size implemented within MKL is not exposed to the user, preventing the ability to compute a meaningful padding for out-of-cache problems.

Finally, Stencil-2D and Stencil-3D are two highly tuned codes we have developed to compute iterative Jacobi stencils (typical in image processing), PDE solving, or function smoothing. Each implements a cross stencil (i.e., computes the average of all neighbors along each orthogonal direction) with fixed coefficients. We made a particular effort to achieve high performance using explicit SIMD vectorization, register tiling, etc.

5.1 Experimental Setup

Experimental Protocol We evaluated the performance of a variety of problem and tile sizes (when applicable) on two machines. SB is a Sandy Bridge single-socket 4-core Intel Core i7-2600K CPU running at 3.40 GHz, and HSW is a Haswell single-socket 4-core Intel Core i7-4770K CPU running at 3.50 GHz. Each runs Linux and has L1 of 32 KB (8-way associativity, $S = 64$), L2 of 256 KB (8-way associativity, $S = 512$), and L3 of 8192 KB (16-way associativity, $S = 8192$). At all levels, the cache line size is 64 B, and we used double-precision floating point, meaning 8 elements per cache line. For each problem/tile size explored, we timed the program's execution with and without padding. Five runs were performed and averaged for each case. Programs were compiled with GCC 4.9.2, using flags `-Ofast -fstrict-aliasing -march=native` and `-fopenmp` for multicore experiments.

We used huge pages (2 MB, with explicit mmap), running RedHat Linux with kernel 2.6.32. TLB misses are negligible in these experiments. In addition, as the computed padding sizes are typically small, there is only a marginal increase in TLB accesses, and TLB misses actually decrease due to reduced cache misses. We also conducted full evaluation using small pages [11], and observed very similar trends and improvements as the one detailed below.

Padding Computation To compute the padding value, we calculated the hot reuse space footprint (e.g., a column of data for MKL-FFT and ADI; a tile of data for Stencil-xx) for each problem size/tile size by manual analysis and computed padding for the smallest cache level fully enclosing this data space. That is, we did not pad systematically for the largest cache but instead padded for the smallest cache containing the data space. The benefit is that a smaller conflict-free padding can be found (having less space overhead) while still ensuring (by definition) a conflict-free space at a higher cache level. Note the reverse is not true: a conflict-free padding for the largest cache does not ensure the data tile is conflict-free for a smaller cache with fewer sets. Automatically computing the data space footprint is out of this paper's scope. Notably, there are numerous techniques to compute this data space, exactly or by over-approximation, such as the distinct line (DL) model [7].

5.2 Experimental Results

MKL-FFT Tables 2-3 show the performance impact of padding for a variety of 2D FFT problem sizes, ran on both machines and in either single- or multi-core settings. Performance is reported in pseudo GF/s, and the padding improvement Imp is shown. We observe that the impact of padding is

greater on Sandy Bridge, higher with larger problem sizes, and usually higher in the parallel case—all expected results. It can reach 40% or more for sizes exceeding 2048 on Sandy Bridge, demonstrating the need for effective padding. In all cases, a padding of 8 elements (i.e., one line size of 64 bytes), the smallest padding producible by our scheme, was the smallest (optimal) padding needed to ensure a lack of conflicts between a row and column of data.

Table 2: MKL-FFT on SB

	1 core			4 cores		
N	no pad	pad	Imp.	no pad	pad	Imp.
512	8.94	9.14	2.3%	18.72	21.25	13.5 %
640	7.80	7.95	1.9%	21.21	21.94	3.4%
768	7.78	7.92	1.7%	22.43	23.27	3.7%
896	7.49	7.56	0.9%	21.69	21.87	0.8%
1024	8.46	9.08	7.3%	18.02	22.26	23.6%
1280	6.90	7.40	7.1%	16.78	18.85	12.4%
1536	6.27	6.91	10.3%	16.42	17.56	7.0%
1792	6.55	7.24	10.5%	16.46	17.94	9.0%
2048	6.20	8.14	31.1%	13.65	19.37	41.9%
2560	5.87	6.86	16.7%	12.92	19.07	47.6%
3072	5.78	7.04	21.6%	11.54	16.99	47.3%
3584	5.54	6.53	17.8%	12.18	17.58	44.3%
4096	6.40	7.98	24.6%	14.55	19.61	34.7%

Table 3: MKL-FFT on HSW

	1 core			4 cores		
N	no pad	pad	Imp.	no pad	pad	Imp.
512	10.04	11.13	11.0%	24.73	24.76	0.1%
640	8.47	9.23	9.0%	23.53	24.55	4.3%
768	8.49	9.31	9.6%	17.88	26.07	45.8%
896	8.67	9.04	4.3%	22.05	26.84	21.7%
1024	9.90	11.33	14.5%	24.86	28.62	15.1%
1280	8.50	8.62	1.5%	21.18	23.33	10.1%
1536	7.88	8.10	2.8%	20.19	22.37	10.8%
1792	8.46	8.49	0.4%	22.00	23.66	7.5%
2048	7.40	10.38	40.3%	15.62	25.70	64.6%
2560	7.36	8.10	10.1%	18.80	22.68	20.6%
3072	7.35	8.09	10.0%	18.57	22.96	23.6%
3584	7.05	7.77	10.2%	18.46	22.69	22.9%
4096	8.41	9.63	14.4%	21.07	25.23	19.7%

ADI Figures 5-6 summarize the performance improvement of padding versus no padding for the ADI benchmark. Performance is reported in GF/s. ADI reflects the impact of padding amplified compared to MKL-FFT, an effect particularly exacerbated when running on multi-core architectures. This stems from the inherent repeated row-first and column-first data access pattern of ADI, where even when the spatial reuse space (N rows each of 1 cache line worth of data) fits in cache L_x, cache L_{x+1} does not contain enough sets to act like a victim cache and ensure evictions from conflict misses in L_x are kept in L_{x+1}, incurring in high miss penalty. Using padding, spatial reuse can be implemented in the smallest cache whose capacity is larger or equal to the reused data footprint because no conflict miss will occur. Therefore, maximal cache utilization is realized. Similar to MKL-FFT, the padding used for each case was 8 elements, the minimal padding in our framework.

DGEMM Fig. 7 reports the results of tile size exploration for the DGEMM benchmark. For clarity, the focus is on 1-core data on HSW, and only a selection of 20+ tile sizes we found to perform best after more extensive exploration. Each

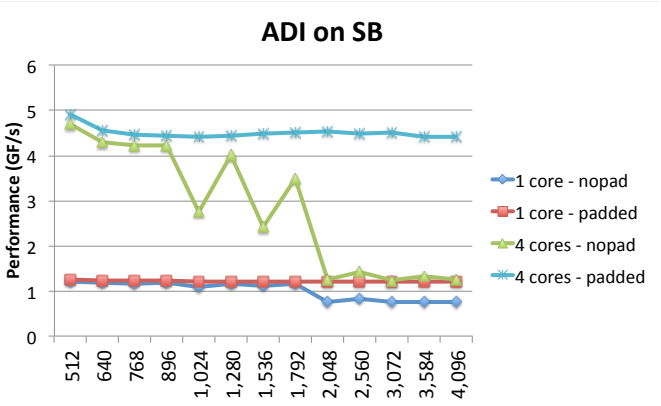


Figure 5: ADI: Impact of padding on SB

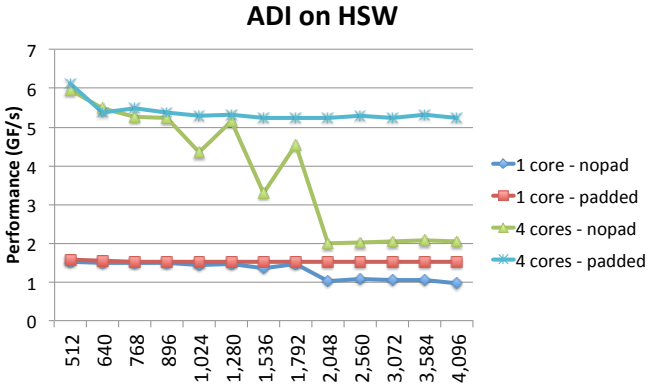


Figure 6: ADI: Impact of padding on HSW

implements a hierarchical tiling (2-level), and we report the achieved GF/s for a problem size $N = 2048$. HT/HPad is hierarchical tiling that uses the nested padding scheme presented in the previous section. We also report the performance achieved for the exact same tiling in: HT/PadL3, padding only for the outer tiles / L3; HT/PadL1, padding only for the inner tiles / L1; and HT/nopad with no padding.

We make several key observations. First, the nested padding approach consistently outperforms all other padding schemes. This clearly motivates the need for hierarchical padding, e.g., applying Song and Li's padding scheme to only one of the two tiling levels would lead to decreased performance. Second, *the performance ordering of different tile sizes without padding is not the same as with padding*. This is a crucial aspect for the tile size exploration framework: the problem of tile size exploration and padding cannot be decoupled, i.e., first explore to find the best tile then pad for it. Based on these experiments, it would lead to selecting a tile size that is about 10% slower after padding than the optimal padded tile size. We argue this is an essential observation for auto-tuning frameworks, motivating the need to have very fast and automated solutions for computing the (hierarchical) padding values such as the method proposed in this paper. Indeed, contrary to the previous benchmarks, here, the computed padding differs between tile sizes, ranging from 8 to 128. Frameworks like ATLAS [3, 24] that

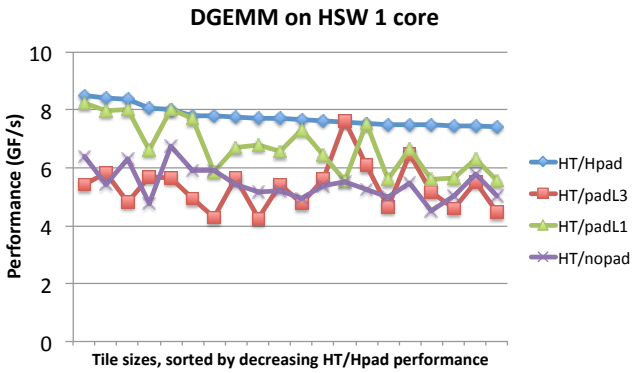


Figure 7: DGEMM: Impact of nested padding

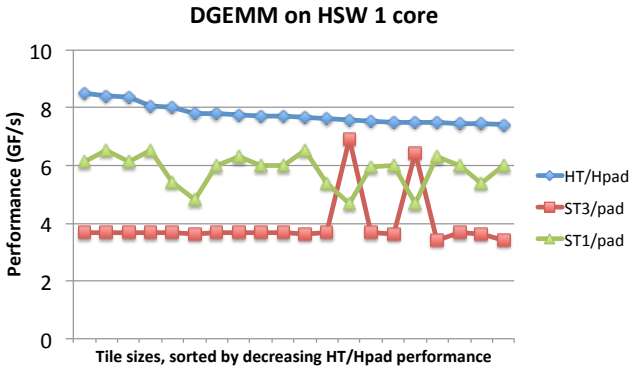


Figure 8: DGEMM: Impact of hierarchical tiling

perform a moderate level of auto-tuning on the target machine are perfect candidate users of our *PAdvisor* tool. We will show this observation holds not only for DGEMM but also for iterative stencils.

Fig. 8 demonstrates the benefit of hierarchical tiling in our experiments. We compare HT/HPad to single-level tiling only for the L3 cache ST/L3 and single-level tiling only for the L1 cache ST/L1. In all cases, the optimal padding is applied. Using AVX2 FMAs, the single-core DP peak performance of this machine is 56 GF/s, and our best performance in this plot is 10 GF/s, indicating that while there is room for improvement, our code achieves solid performance. Of note, the impact of padding relates to the quality of the optimized code. For inefficient codes where conflict misses are not the dominant bottleneck, padding does not provide much improvement. This is not the case in our examples, given the strong improvements via padding only.

HPGMG Fig. 9 provides a comparative plot for a tile size exploration on HPGMG, running on a core of HSW mirroring the Message Passing Interface (MPI)-based distribution of the full HPGMG code. We display a larger number of tile sizes to show the impact of intra-array padding only ST/pad versus intra- and inter-array padding ST/pad+inter, against no padding ST/nopad.

As with GEMM, we see the performance ordering of tiles is not the same whether or not padding is applied and that padding significantly improves performance. We also observe the high impact of inter-array padding in this case. HPGMG uses 5 different arrays, and inter-array conflicts occur frequently, even if the data tile is conflict-free for one array. Indeed, when computing padding for set-associative caches individually for each array, we aim to find the smallest padding that ensures no conflicts, assuming only this array occupies the cache. In contrast, considering both intra- and inter-array padding as depicted in the previous section is key for performance in this situation—with up to 4× improvements for certain tile sizes over intra-padding alone. Similarly to DGEMM, the padding (both intra and inter) values computed by our approach differ between tile sizes with inter-array padding values ranging from 573 KB to 2.6 MB.

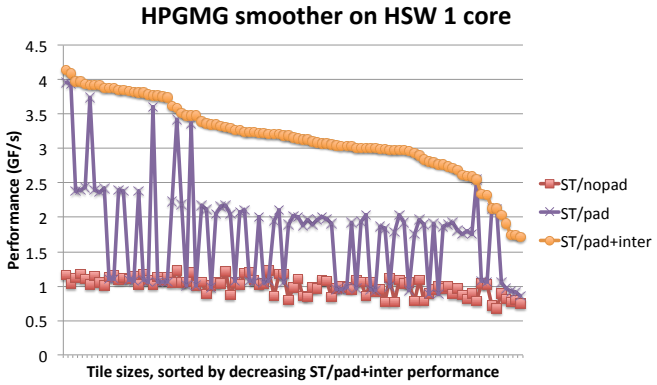


Figure 9: HPGMG: Impact of intra/inter array padding

Stencils Table 4 summarizes the best performance that can be achieved after an extensive auto-tuning of tile sizes on two stencil computations. We display the best performance achieved in GF/s (the higher, the better), to emphasize the high-performance nature of our customized implementations. We integrated *PAdvisor* in the tile size selection process and report the performance for the best tile found. For each case, the tile achieving the best performance in the nopad case is not the same as the one for the intra case, representing intra-array padding only. We also show the impact of inter-array padding on performance in the *intra+inter* columns.

Table 4: Stencil-2D (top three entries) and Stencil-3D (bottom three entries), in GFlop/s using 4 cores

	SB			HSW		
N	no pad	intra	intra+inter	no pad	intra	intra+inter
1024	10.22	21.54	25.22	20.23	32.16	32.30
1536	13.03	27.44	33.52	26.23	39.23	39.89
2048	13.06	27.66	32.02	26.19	37.97	38.54
256	13.74	22.42	24.76	20.19	27.94	30.45
384	18.63	21.86	22.11	24.06	27.00	27.29
512	17.74	20.29	20.31	22.28	27.08	27.08

PAdvisor *Running Time* We conclude our experimental study with a display of the execution time of our *PAdvisor* implementation using scenarios requiring the most computation: 3D data space, non-power-of-two data tiles using L3 16-way set-associative cache. Fig. 10 shows the time, in milliseconds, for a variety of tile sizes. Each series depicts a

different inner-most tile size (varying other tile sizes), empirically illustrating that our algorithm's complexity is not driven by the data space size but by the size of the inner-most tile dimension. In any case, the our implementation's execution time is in the milliseconds range, making it suitable for integration both in production compilers and auto-tuning frameworks.

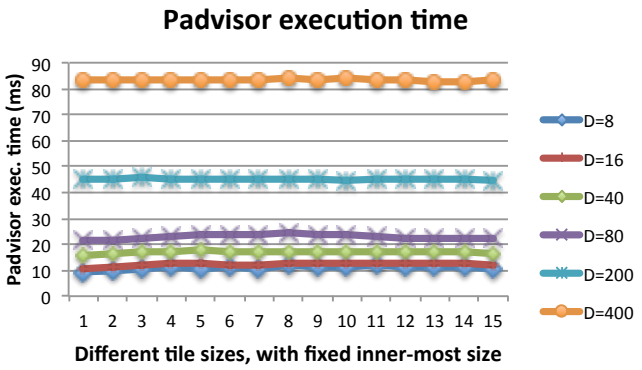


Figure 10: *PAdvisor* execution time

5.3 Discussion

Padding Versus Data Copying An alternative to padding is data copying, where data to be loaded in the cache is explicitly copied into contiguous smaller temporary arrays, improving conflict misses. This approach has trade-offs. The cost of copying data has to be amortized by making a high number of uses of the copied block, which would otherwise have suffered a high number of conflict misses. Copying is used in Intel MKL, for instance, for their BLAS3 DGEMM implementation, but not for code with a lower reuse factor, such as MKL FFT routines. Assessing the profitability of data copying is difficult and implementation-specific, but for codes with low arithmetic intensity, such as FFTs, ADIs, and simple stencils, the copy cost is unlikely to be amortized. In contrast, padding does not incur any copy, but it does require the user to pad the data structure across complete functions or programs, and does increase the amount of virtual memory needed for the padded data structures.

Replacement Policy While the cache replacement policy can clearly affect the number of cache misses for an application, it is not expected to make much of a difference when repeated accesses to a data tile occur in a padded array—as long as the replacement policy is some approximation of the Least Recently Used (LRU) (as is the case in practice). This is because the padding guarantees that all accessed data can fit without conflict in the cache. The only possibility of encountering misses via conflict among tile data is if pre-existing data in some cache lines are retained, and more recently accessed tile data are replaced instead. This scenario is possible with a random replacement policy. Regardless, even with such a policy, the probability of conflict misses among tile data will decrease asymptotically with repeated accesses.

Our analysis does not account for non-tile data accessed, e.g., due to register spill code introduced by the compiler, or access to data on the stack. In this case, conflict-freedom cannot be guaranteed by padding. Therefore the replacement policy may have an impact on performance. However, in such a scenario, padding to utilize a capacity of (A-1)S instead of the full capacity of AS should be able to guarantee freedom from conflict misses, assuming that all data accessed on the stack are less than SB bytes.

6. Conclusion

Array padding is a well-known technique for application developers, especially for the rather commonly encountered scenario in scientific/engineering computing where natural extents of multidimensional dense arrays are powers-of-two. For example, Intel provides an "FFT Length and Layout Advisor" [12] to use in conjunction with the Intel MKL FFT library. This tool uses heuristics to determine suitable padding for the user-specified array size for multidimensional FFTs. Rather than a heuristic, *PAdvisor* provides conflict-free padding solutions with minimal padding space overhead for arbitrary multidimensional tile data footprints, and nested hierarchical tiles.

In this work, we have made several contributions, including: 1) developed optimal analytical solutions for the array padding problem for set-associative caches when tile sizes divide the number of cache sets, 2) developed efficient computational solutions for the general case of arbitrary-sized tiles and multiple arrays with set-associative caches, 3) presented a first solution for interference-free padding of hierarchical tiles in a multi-level cache hierarchy, 4) integrated these new developments in a tool called *PAdvisor*, and 5) provided an experimental evaluation with a variety of benchmarks to demonstrate the impact of conflict misses and the effectiveness of *PAdvisor*.

Experimental data clearly showed many cases with a tight coupling between tile size optimization and padding. If tile size selection is done first without padding and optimal padding is performed for that tile size, the achieved performance is not as high as with co-tuning, where optimal padding is done for each tile size in the auto-tuning run. *PAdvisor* is very fast and can be effectively used for such co-tuning of padded data layout and tile size optimization in auto-tuning environments, such as OpenTuner [1, 2], CHiLL [23], Active Harmony [22], and a number of other auto-tuning frameworks [5, 8, 24].

Acknowledgements

We are grateful to the PLDI'16 reviewers for their very detailed feedback and suggestions, which helped improve the paper. This work was supported in part by NSF through award ACI-1440749, and DOE's Office of Science, Office of Advanced Scientific Computing Research, under DOE award DE-SC0008844 and Early Career award 63823. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] J. Ansel. *Autotuning programs with algorithmic choice*. PhD thesis, Massachusetts Institute of Technology, 2014.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *PACT'14*, pages 303–316. ACM, 2014.

[3] ATLAS. ATLAS homepage. `http://math-atlas.sourceforge.net`.

[4] D. F. Bacon, J.-H. Chow, D.-c. R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. In *CASCON'94*. IBM Press, 1994.

[5] J. Bilmes. PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS'97*. ACM, 1997.

[6] J. Douglas. Alternating direction methods for three space variables. *Numerische Mathematik*, 4(1):41–63, 1962.

[7] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. *LCPC'92*, pages 328–343, 1992.

[8] M. Frigo. A fast Fourier transform compiler. In *PLDI'99*, pages 169–180. ACM, May 1999.

[9] 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, July 1999.

[10] E. Herruzo, O. Plata, and E. L. Zapata. Using padding to optimize locality in scientific applications. In *ICCS'08*, pages 863–872. Springer, 2008.

[11] C. Hong, W. Bao, A. Cohen, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. Effective padding of multi-dimensional arrays to avoid cache conflict misses. Technical Report OSU-CISRC-4/16-TR2, Ohio State University, 2016.

[12] Intel. Intel FFT length and layout advisor. `https://software.intel.com/en-us/articles/fft-length-and-layout-advisor`, .

[13] Intel. Intel Math Kernel Library. `https://software.intel.com/en-us/intel-mkl`, .

[14] K. Ishizaka, M. Obata, and H. Kasahara. Cache optimization for coarse grain task parallel processing using inter-array padding. In *LCPC'04*, pages 64–76. Springer, 2004.

[15] S. G. Johnson and M. Frigo. Implementing FFTs in practice. In C. S. Burrus, editor, *Fast Fourier Transforms*, chapter 11. Connexions, Rice University, Houston TX, September 2008.

[16] M. Kowarschik and C. Weiss. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, pages 213–232. Springer, 2003.

[17] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, Nov. 2004.

[18] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Trans. on Computers*, 48(2):142–149, 1999.

[19] D. W. Peaceman and H. H. Rachford, Jr. The numerical solution of parabolic and elliptic differential equations. *J. of the Society for Industrial and Applied Mathematics*, 3(1):28–41, 1955.

[20] L.-N. Pouchet and T. Yuki. PolyBench/C 4.1. `http://polybench.sourceforge.net`.

[21] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *SC'00*, page 32. IEEE, 2000.

[22] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth, et al. Active harmony: Towards automated performance tuning. In *SC'02*, pages 1–11. IEEE, 2002.

[23] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS'09.*, pages 1–12. IEEE, 2009.

[24] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[25] S. Williams. High-Performance Geometric MultiGrid. `https://hpgmg.org/`.

A. Appendix

We first prove the main result of Sec. 3: Theorem 2. We use the following notation: N_i is the padded size of an array along dimension i, D_i is the tile size along dimension i, S is the number of sets in the cache, A is the associativity, and B is the cache block size. For the case of "divisible tiles" assumed in Sec. 3, N_1 and D_1 are multiples of B. We will simplify the notation without loss of generality and assume that N_1 and D_1 have been normalized by dividing by B, i.e., consider $B = 1$. Let $g_i = \left(S/\prod_{1\le k<i} g_k\right) \wedge N_i$, for all $1 \le i < d$. Finally, let $n_i = N_i/g_i$ and $s_i = S/\prod_{1\le k\le i} g_k$ for $1 \le i < d$, and $\sigma = s_{d-1}$.

Lemma 7. *Consider a set-associative cache of capacity C. If the following conditions are met, then a loop nest whose tiles have a d-dimensional array footprint can fully utilize the cache and remain free of self-interference:*

1. $\forall i, 1 \le i < d$, g_i *divides* D_i.
2. S *divides* $D_d \prod_{1\le i<d} g_i$.

Proof. Let $\upsilon = \left(\prod_{1\le i<d} g_i\right) D_d/S$. We have $\sigma = D_d/\upsilon$. Partition the $D_d \times D_{d-1} \times \ldots \times D_1$ tile into sub-tiles of size $\sigma \times g_{d-1} \times \ldots \times g_1$. We show that no two memory blocks within a sub-tile can map to the same cache set. The total number of sub-tiles is exactly the cache associativity A because $\frac{\prod_{1\le i\le d} D_i}{D_d/\upsilon \prod_{1\le i<d} g_i} = \frac{\upsilon \prod_{1\le i\le d} D_i}{D_d \prod_{1\le i<d} g_i} = \frac{\upsilon \prod_{1\le i\le d} D_i}{\upsilon S} = A$. Consider two blocks of the same sub-tile with respective index $(i_d, \ldots, i_2, i_1)$ and $(j_d, \ldots, j_2, j_1)$. Supposing they map to the same cache set, then:

$$\overline{\left(i_1 + \sum_{2\le\ell\le d} i_\ell \prod_{1\le k<\ell} N_k\right)}_S \equiv_S \overline{\left(j_1 + \sum_{2\le\ell\le d} j_\ell \prod_{1\le k<\ell} N_k\right)}_S$$

Denoting $\delta_k = i_k - j_k$ for each $1 \leq k \leq d$, as $S = g_1 s_1$,

$$\overline{\left(\delta_1 + N_1 \left(\delta_2 + \cdots + \delta_d \prod_{2 \leq k < d} N_k\right)\right)}_{g_1} \equiv_{g_1} \overline{0}_{g_1}$$

Since $N_1 = g_1 n_1$ and $|i_1 - j_1| < g_1$, necessarily $i_1 = j_1$. The previous equation becomes

$$\overline{\left(N_1 \left(\delta_2 + \delta_3 N_2 + \cdots + \delta_d \prod_{2 \leq k < d} N_k\right)\right)}_{S} \equiv_S \overline{0}_S$$

i.e., (as $N_1 = n_1 g_1$ and $S = s_1 g_1$),

$$\overline{\left(n_1 \left(\delta_2 + \delta_3 N_2 + \cdots + \delta_d \prod_{2 \leq k < d} N_k\right)\right)}_{s_1} \equiv_{s_1} \overline{0}_{s_1}$$

which reduces (as $g_1 = S \wedge N_1$, i.e., $1 = s_1 \wedge n_1$) to,

$$\overline{\left(\delta_2 + \delta_3 N_2 + \cdots + \delta_d \prod_{2 \leq k < d} N_k\right)}_{s_1} \equiv_{s_1} \overline{0}_{s_1}$$

Applying the same reasoning iteratively on all dimensions, we conclude that for all $1 \leq i \leq d,\ \delta_i = 0$. □

Lemma 8. *Consider a set-associative cache of capacity C. Let $g_d = S/\prod_{1 \leq k \leq d-1} g_k$, i.e., $S = \prod_{1 \leq k \leq d} g_k$. If the following condition is met, then a loop nest whose tiles have a d-dimensional array footprint can fully utilize the cache and remain free of self-interference:*

1. $\forall q, 1 \leq q \leq d,\ \exists p, 1 \leq p \leq q,\ \prod_{1 \leq k \leq q} g_k$ divides $D_p \prod_{1 \leq k < p} g_k$.

Proof. We observe that the above condition implies (for all q) the existence of p such that $\prod_{p \leq k \leq q} g_k$ divides D_p. This allows to partition the interval $[1 : d]$ into consecutive intervals $[1 : q_1]$, $[p_2 : q_2]$, …, $[p_e : d]$ (we have $\forall l, q_l + 1 = p_{l+1}$), where $\prod_{p_l \leq k \leq q_l} g_k$ divides D_{p_l}. For each interval, we define $g'_{p_l} = \prod_{p_l \leq k \leq q_l} g_k$, and for all other $p_l < i \leq q_l$, $g'_i = 1$. We have $\prod_{p_l \leq k \leq q_l} g_k = \prod_{p_l \leq k \leq q_l} g'_k$.

By definition of p_e, $\prod_{1 \leq k \leq d} g_k$ (also equal to S) divides $D_{p_e} \prod_{1 \leq k < p_e} g_k = D_{p_e} \prod_{1 \leq k < p_e} g'_k$.

Let $\upsilon = \left(D_{p_e} \prod_{1 \leq i < p_e} g'_i\right)/S$ and partition the $D_d \times D_{d-1} \times \ldots \times D_1$ data tile into sub-tiles of size $1 \times \ldots 1 \times (D_{p_e}/\upsilon) \times g'_{p_e - 1} \times \ldots \times g'_1$. Similar to Lemma 8, we show that no two memory blocks within a sub-tile can map to the same cache set. The total number of sub-tiles is the cache associativity A because $\frac{\prod_{1 \leq i \leq d} D_i}{1^{d-p_e} (D_{p_e}/\upsilon) \prod_{1 \leq i < p_e} g'_i} = \frac{\upsilon \prod_{1 \leq i \leq d} D_i}{D_{p_e} \prod_{1 \leq i < p_e} g'_i} = \frac{\upsilon \prod_{1 \leq i \leq d} D_i}{\upsilon S} = A$. Consider two blocks of the same sub-tile with index $(i_d, \ldots, i_2, i_1)$ and $(j_d, \ldots, j_2, j_1)$. Supposing they map to the same cache set, and denoting $\delta_k = i_k - j_k$ for each $1 \leq k \leq d$, then:

$$\overline{\left(\delta_1 + N_1 \left(\delta_2 + \cdots + \delta_d \prod_{2 \leq k < d} N_k\right)\right)}_{S} \equiv_S \overline{0}_S \qquad (1)$$

We have $|\delta_{p_e}| < D_{p_e}/\upsilon$, $|\delta_{p_l}| < g'_{p_l}$ for $0 \leq l < e$, and $\delta_i = 0$ otherwise. Eq. 1 can be rewritten, merging terms interval by interval, as follows:

$$\overline{\left(\delta_1 + \prod_{1 \leq k \leq q_1} N_k \left(\delta_{p_2} + \cdots + \delta_{p_e} \prod_{p_2 \leq k < d} N_k\right)\right)}_{S} \equiv_S \overline{0}_S$$

As $\prod_{1 \leq k \leq q_1} N_k = g'_1 \prod_{1 \leq k \leq q_1} n_k$, and because the previous equation also holds modulo g'_1, necessarily $\delta_1 = 0$. Similar to the proof of Lemma 8 (now iterating on l), we conclude that for all $1 \leq i \leq d,\ \delta_i = 0$. □

Lemma 9. *Assume that $n \wedge S = 1$, and let us consider $\overline{k}_S \in \mathbb{Z}/S\mathbb{Z}$, such that $k \mid S$. Let for some $\overline{p}_S \in \mathbb{Z}/S\mathbb{Z}$, define in $\mathbb{Z}/S\mathbb{Z}$: $P = \bigcup_{\alpha \in \mathbb{Z}} \overline{((p + \alpha k)n)}_S$, and $P' = \bigcup_{\alpha \in \mathbb{Z}} \overline{(pn + \alpha k)}_S$. Then, $P = P'$*

Proof. We have that $n \wedge S = 1$, and therefore $n \wedge (S/k) = 1$. P rewrites as $\overline{(pn)}_S + \bigcup_{\alpha \in \mathbb{Z}} \overline{(\alpha nk)}_S$, and P' as $\overline{(pn)}_S + \bigcup_{\alpha \in \mathbb{Z}} \overline{(\alpha k)}_S$. We need to prove that $\bigcup_{\alpha \in \mathbb{Z}} \overline{(\alpha nk)}_S = \bigcup_{\alpha \in \mathbb{Z}} \overline{(\alpha k)}_S$. As $k \mid S$, this is equivalent to prove that $\bigcup_{\alpha \in \mathbb{Z}} \overline{(\alpha n)}_{S/k} = \bigcup_{\alpha \in \mathbb{Z}} \overline{(\alpha)}_{S/k}$, which is true as $n \wedge (S/k) = 1$. □

Lemma 10. *Define the occupancy of $\overline{i}_S \in \mathbb{Z}/S\mathbb{Z}$ as $\mathsf{occ}_{yx}(\overline{i}_S) = \left|\{(y, x) : \overline{(N_1 y + x)}_S \equiv \overline{i}_S \wedge 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$. Suppose that $g_1 \nmid D_1$ or ($S \nmid D_1$, and $S \nmid g_1 D_2$). Then, this occupancy is not uniform. In other words, there exists $\overline{i}_S \not\equiv \overline{j}_S$ such that $\mathsf{occ}_{yx}(\overline{i}_S) \neq \mathsf{occ}_{yx}(\overline{j}_S)$.*

Proof. First, suppose that $g_1 \nmid D_1$. Define $\mathsf{CC}_{g_1}(\overline{i}_{g_1}) = \{(y, x) : \overline{(N_1 y + x)}_{g_1} \equiv \overline{i}_{g_1} \wedge 0 \leq y < D_2,\ 0 \leq x < D_1\}$. For $\overline{i}_{g_1} \not\equiv \overline{i'}_{g_1}$, $\mathsf{CC}_{g_1}(\overline{i}_{g_1}) \cap \mathsf{CC}_{g_1}(\overline{i'}_{g_1}) = \emptyset$. Assuming (by contradiction) the occupancy ($\mathsf{occ}_{yx}()$) of cache sets to be uniform, then the occupancy of CC's set also must be uniform. In other words, $\forall \overline{i}_S \not\equiv \overline{j}_S,\ \mathsf{CC}_{g_1}(\overline{i}_S) = \mathsf{CC}_{g_1}(\overline{j}_S)$. Because $g_1 \mid N_1$, if $(y, x) \in \mathsf{CC}_{g_1}(\overline{i}_{g_1})$, then $\forall 0 \leq y' < D_2,\ (y', x) \in \mathsf{CC}_{g_1}(\overline{i}_{g_1})$. In other words, $|\mathsf{CC}_{g_1}(\overline{i}_{g_1})| = D_2 |\{\overline{x}_{g_1} \equiv \overline{i}_{g_1} \wedge 0 \leq x < D_1\}| = D_2 \left\lfloor (D_1 - \overline{i}_{g_1})/g_1 \right\rfloor$. Hence, for this to be equal for any value of $\overline{i}_{g_1}$, we must have $g_1 \mid D_1$, which is a contradiction.

Now, suppose $g_1 \mid D_1$, $S \nmid D_1$, and $S \nmid g_1 D_2$. Because $g_1 = S \wedge N_1$ divides both S, N_1 and D_1, we have that $\mathsf{occ}_{yx}(\overline{i})_{S/g_1} = \left|\{(y, x') : \overline{(yN_1/g_1 + x')}_{S/g_1} \equiv \overline{i}_{S/g_1} \wedge 0 \leq y < D_2,\ 0 \leq x' < D_1/g_1\}\right| = g_1 \mathsf{occ}_{yx}(\overline{iS})$. The consequence is that without loss of generality, we can essentially consider that $g_1 = S \wedge N_1 = 1$. Thus, the last $(D_1 - (D_1 \mod S))$ columns uniformly occupy the cache sets. Observe that $D_1 \mod S \neq 0$. We can assume, without loss of generality, that $0 < D_1 < S$. Also, as N_1 is a generator of $\mathbb{Z}/S\mathbb{Z}$, the last $(D_2 - (D_2 \mod S))$ rows uniformly occupy the cache sets. Similarly, we assume that $0 < D_2 < S$. Let n' denote the inverse of $\overline{(N_1)}_S$ in $\mathbb{Z}/S\mathbb{Z}$. We have that $\mathsf{occ}_{yx}(\overline{i}_S) = \sum_{0 \leq x < D_1} \left|\{(y, x) : \overline{y}_S \equiv \overline{(i - x)}_S n' \wedge 0 \leq y < D_2\}\right|$. Let us define $\delta(\overline{y}_S)$ as 1 if $0 \leq y < D_2$ and 0 if $D_2 \leq y < S$: $\forall \overline{i}_S,\ \mathsf{occ}_{yx}(\overline{i}_S) = \sum_{0 \leq x < D_1} \delta\left(\overline{(i - x)}_S n'\right)$. We get, $\forall i, \mathsf{occ}_{yx}(\overline{i}_S) - \mathsf{occ}_{yx}(\overline{(i-1)}_S) = \delta\left(\overline{i}_S n'\right) - \delta\left(\overline{(i - D_1)}_S n'\right)$. Suppose now that $\mathsf{occ}_{yx}()$ is uniform. We have $\mathsf{occ}_{yx}(\overline{i}_S) = \mathsf{occ}_{yx}(\overline{(i-1)}_S)$, i.e., $\forall \overline{i}_S,\ \delta\left(\overline{i}_S n'\right) = \delta\left(\overline{(i - D_1)}_S n'\right)$. Setting $k = D_1 \wedge S$ (observe that $k \neq S$ and $k \mid S$), we have that

for any $\overline{i}_S \in \mathbb{Z}/S\mathbb{Z}$, $\forall \alpha \in \mathbb{Z}, \delta\left(\overline{i}_S n'\right) = \delta\left(\left(\overline{i}_S + \alpha k\right) n'\right)$. By Lemma 9, $\forall \alpha \in \mathbb{Z}, \delta\left(\overline{i}_S n'\right) = \delta\left(\overline{i}_S n' + \alpha k\right)$. By definition of $\delta()$, we have $\delta\left(\overline{(S-1)}_S\right) = \delta\left(\overline{-1}_S\right) = 0$, which leads to $\delta\left(\overline{(-1+k)}_S\right) = 0$ i.e. $k - 1 \geq D_2$. We also have $\delta\left(\overline{0}_S\right) = 1$, leading to $\delta\left(\overline{k}_S\right) = 1$. This implies $k < D_2$, which is absurd. So $\mathsf{occ}_{yx}()$ cannot be uniform. □

Lemma 11. *Let $\mathsf{occ'}_x(\overline{i}_{g_1}) = \left|\{(x) : \overline{x}_{g_1} \equiv \overline{i}_{g_1},\ 0 \leq x < D_1\}\right|$, $\mathsf{occ'}_{zyx}(\overline{i}_{g_1}) = \left|\{(z,y,x) : \overline{(N_2N_1z + N_1y + x)}_{g_1} \equiv \overline{i}_{g_1} \wedge 0 \leq z < D_3,\ 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$, and $\mathsf{occ}_{zyx}(\overline{i}_S) = \left|\{(z,y,x) : \overline{(N_2N_1z + N_1y + x)}_S \equiv \overline{i}_S \wedge 0 \leq z < D_3,\ 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$. If the occupancy $\mathsf{occ'}_x()$ is not uniform, then the occupancies $\mathsf{occ'}_{zyx}()$ and $\mathsf{occ}_{zyx}()$ also are not uniform.*

Proof. Because $g_1 \mid N_1$, $\mathsf{occ'}_{zyx}(\overline{i}_{g_1}) = \left|\{(z,y,x) : \overline{x}_{g_1} \equiv \overline{i}_{g_1} \wedge 0 \leq z < D_3,\ 0 \leq y < D_2,\ 0 \leq x < D_1\}\right| = D_3D_2\mathsf{occ'}_x(\overline{i}_{g_1})$. In other words, if $\mathsf{occ'}_x()$ is not uniform, then $\mathsf{occ'}_{zyx}()$ cannot be uniform. Moreover, S is a multiple of g_1. So, $\mathsf{occ}_{zyx}()$ is not uniform. □

Lemma 12. *Let $\mathsf{occ'}_{yx}(\overline{i}_{g_1g_2}) = \left|\{(y,x) : \overline{(N_1y + x)}_{g_1g_2} \equiv \overline{i}_{g_1g_2} \wedge 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$, $\mathsf{occ'}_{zyx}(\overline{i}_{g_1g_2}) = \left|\{(z,y,x) : \overline{(N_2N_1z + N_1y + x)}_{g_1g_2} \equiv \overline{i}_{g_1g_2} \wedge 0 \leq z < D_3,\ 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$, and $\mathsf{occ}_{zyx}(\overline{i}_S) = \left|\{(z,y,x) : \overline{(N_2N_1z + N_1y + x)}_S \equiv \overline{i}_S \wedge 0 \leq z < D_3,\ 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$. If the occupancy $\mathsf{occ'}_x()$ is not uniform, then the occupancies $\mathsf{occ'}_{zyx}()$ and $\mathsf{occ}_{zyx}()$ also are not uniform.*

Proof. Similar to the proof for Lemma 11. □

Lemma 13. *Suppose that $g_1|D_1$ and $D_2 < S/g_1$. Let $\mathsf{occ}_{yx}(\overline{i}_S) = \left|\{(y,x) : \overline{(N_1y + x)}_S \equiv \overline{i}_S \wedge 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$. Suppose this occupancy to be non-uniform ($\mathsf{occ}_{yx}(\overline{i}_S)$ is not constant over $\mathbb{Z}/S\mathbb{Z}$). Then, $\forall \overline{k}_S \not\equiv \overline{0}_S \in \mathbb{Z}/S\mathbb{Z}$, $\exists \overline{j}_S \in \mathbb{Z}/S\mathbb{Z}$ s.t. $\mathsf{occ}_{yx}(\overline{j}_S) \neq \mathsf{occ}_{yx}(\overline{(j+k)}_S)$.*

Proof. Because $g_1 = S \wedge N_1$ divides both S, N_1 and D_1, we have that $\mathsf{occ}_{yx}(\overline{i})_{S/g_1} = \left|\{(y,x') : \overline{(yN_1/g_1 + x')}_{S/g_1} \equiv \overline{i}_{S/g_1} \wedge 0 \leq y < D_2,\ 0 \leq x' < D_1/g_1\}\right| = g_1\mathsf{occ}_{yx}(\overline{i})_S$. The consequence is that without loss of generality, we can consider that $S \wedge N_1 = 1$ and $D_2 < S$. We also can assume (because of non-uniform occupancy) that $D_2 \not\equiv 0$. Also, $\mathsf{occ}_{yx}(\overline{i}_S)$ can be rewritten as $\sum_{0 \leq x < D_1} \left|\{(y,x) : \overline{(N_1y + x)}_S \equiv \overline{i}_S \wedge 0 \leq y < D_2\}\right|$. Denoting n' as the inverse of $\overline{(N_1)}_S$ in $\mathbb{Z}/S\mathbb{Z}$, it then can be rewritten as $\mathsf{occ}_{yx}(\overline{i}_S) = \sum_{0 \leq x < D_1} \left|\{(y,x) : \overline{y}_S \equiv \overline{(i-x)}_S n' \wedge 0 \leq y < D_2\}\right|$. For the rest of this proof, *all* variables (but α) belong to $\mathbb{Z}/S\mathbb{Z}$. To simplify the notations, modulo arithmetic—overline and S-subscript—is left implicit below. Let us define $\delta(y)$ as 1 if $0 \leq y < D_2$ and 0 otherwise: $\forall i,\ \mathsf{occ}_{yx}(i) = \sum_{0 \leq x < D_1} \delta((i-x)n')$. We get, $\forall i, \mathsf{occ}_{yx}(i) - \mathsf{occ}_{yx}(i-1) = \delta(in') - \delta((i-D_1)n')$. Because $\delta(y) \in \{0,1\}$, $|\mathsf{occ}_{yx}(i) - \mathsf{occ}_{yx}((i-1))| \leq 1$. Also, $\sum_{i \in \mathbb{Z}/S\mathbb{Z}} \mathsf{occ}_{yx}(i) - \mathsf{occ}_{yx}((i-1)) = 0$. As a consequence, because we considered non-uniform occupancy in the hypothesis, there exists $p \in \mathbb{Z}/S\mathbb{Z}$ such that $\mathsf{occ}_{yx}(p) - \mathsf{occ}_{yx}((p-1)) = 1$. To prove our lemma, we assume by contradiction that $\exists k \not\equiv 0$ s.t. $\forall j,\ \mathsf{occ}_{yx}(j) = \mathsf{occ}_{yx}(j+k)$. We have that $\forall \alpha \in \mathbb{Z}, \mathsf{occ}_{yx}(j) = \mathsf{occ}_{yx}(j + \alpha(k \wedge S))$. We can assume, without loss of generality, that $k \wedge S = k$, i.e., $k \mid S$. Recall that $\forall i, \delta(in') - \delta((i-D_1)n') = \mathsf{occ}_{yx}(i) - \mathsf{occ}_{yx}(i-1)$. In particular, $\forall \alpha, \delta((p+\alpha k)n') - \delta((p-D_1+\alpha k)n') = \mathsf{occ}_{yx}(p+\alpha k) - \mathsf{occ}_{yx}(p+\alpha k-1) = \mathsf{occ}_{yx}(p) - \mathsf{occ}_{yx}(p-1) = 0$. Because $\delta(y) \in \{0,1\}$, $\forall \alpha, \delta((p+\alpha k)n') = 1$, and $\delta((p-D_1+\alpha k)n') = 0$. Applying Lemma 9, we get that $\forall \alpha, \delta(pn' + \alpha k) = 1$, and $\delta((p-D_1)n' + \alpha k) = 0$. This means that there exists $0 \leq y < k$ such that $\delta(y) = 0$ and $k \leq y' < 2k$ (recall that $k \mid S$ and $k \not\equiv \overline{0}_S$) such that $\delta(y') = 1$. By definition of $\delta()$, this means that $y \geq D_2$ and $y' < D_2$, which contradicts the fact that $y < k \leq y'$. □

Lemma 14. *Define the occupancy of $\overline{i}_S \in \mathbb{Z}/S\mathbb{Z}$ as $\mathsf{occ}_{zyx}(\overline{i}_S) = \left|\{(z,y,x) : \overline{(N_2N_1z + N_1y + x)}_S \equiv \overline{i}_S \wedge 0 \leq z < D_3,\ 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$. Suppose that $S \nmid D_1$, and $S \nmid g_1D_2$, and $S \nmid g_1g_2D_3$. Then, this occupancy is not uniform. In other words, there exists $\overline{i}_S \not\equiv \overline{j}_S$ such that $\mathsf{occ}_{zyx}(\overline{i}_S) \neq \mathsf{occ}_{zyx}(\overline{j}_S)$.*

Proof. First, we will assume that $g_1|D_1$. Indeed, if this occupancy is uniform, then similarly to the proof for Lemma 10, we can prove that $g_1|D_1$. Let $\mathsf{occ}_{yx}(\overline{i}_S) = \left|\{(y,x) : \overline{(N_1y + x)}_S \equiv \overline{i}_S \wedge 0 \leq y < D_2,\ 0 \leq x < D_1\}\right|$. First, observe that for any $\overline{i}_S \in \mathbb{Z}/S\mathbb{Z}$, $\mathsf{occ}_{zyx}(\overline{i}_S) = \sum_{0 \leq z < D_3} \mathsf{occ}_{yx}(\overline{(i - zN_1N_2)}_S)$. So, for any $\overline{i}_S \in \mathbb{Z}/S\mathbb{Z}$, $\mathsf{occ}_{zyx}(\overline{i}_S) - \mathsf{occ}_{zyx}(\overline{(i-N_1N_2)}_S)$ is equal to $\mathsf{occ}_{yx}(\overline{i}_S) - \mathsf{occ}_{yx}(\overline{(i-D_3N_1N_2)}_S)$. Suppose by contradiction that $\mathsf{occ}_{zyx}(\overline{i}_S)$ is constant (uniform occupancy). A direct consequence is that $\mathsf{occ}_{yx}(\overline{i}_S) = \mathsf{occ}_{yx}(\overline{(i-D_3N_1N_2)}_S)$. In other words, this means that there exists $\overline{k}_S$ (equal to $\overline{(-D_3N_1N_2)}_S \in \mathbb{Z}/S\mathbb{Z}$) such that for all $\overline{i}_S \in \mathbb{Z}/S\mathbb{Z}$, $\mathsf{occ}_{yx}(\overline{i}_S) = \mathsf{occ}_{yx}(\overline{(i+k)}_S)$. Now, the hypothesis that $S \nmid D_1$, and $S \nmid g_1D_2$ implies (from Lemma 10) that $\mathsf{occ}_{yx}(\overline{i}_S)$ is not constant (non-uniform occupancy). In order to apply Lemma 13 to prove the contradiction, we need to prove that $\overline{k}_S \not\equiv \overline{0}_S$. By definition of g_1, $(S/g_1) \wedge (N_1/g_1) = 1$, and in particular, $(S/(g_1g_2)) \wedge (N_1/g_1) = 1$. Also, $(S/(g_1g_2)) \wedge (N_2/g_2) = 1$. By hypothesis $(S/(g_1g_2)) \nmid D_3$. Thus, $S/(g_1g_2) \nmid (N_1/g_1)(N_2/g_2)D_3$. In other words, $\overline{(D_3N_1N_2)}_S \not\equiv \overline{0}_S$. □

Theorem 2 (Set-associative cache)**.** *Consider a set-associative cache of capacity $C = SAB$. For all $1 \leq i \leq d-1$, let $g_i = S/\prod_{1 \leq k \leq i-1} g_k \wedge N_i$. A loop nest whose tiles have a d-dimensional array footprint can fully utilize the cache and remain free of self-interference if and only if the following conditions are met:*

1. *$\forall i, 1 \leq i \leq d-1$, $\exists j, 1 \leq j \leq i$, $\prod_{1 \leq k \leq i} g_k$ divides $D_j \prod_{1 \leq i \leq j-1} g_i$.*
2. *$\exists i, 1 \leq i \leq d$, S divides $D_i \prod_{1 \leq k \leq i-1} g_k$.*

Proof. Lemma 8 proved the sufficient condition, and the necessary condition follows from Lemma 14. □