An analytical model for loop tiling transformation

Vasilios Kelefouras[1,4] v.kelefouras@plymouth.ac.uk, Karim Djemame[2], Georgios Keramidas[3,4], and Nikolaos Voros[4]

[1] University of Plymouth (UK)
[2] University of Leeds (UK)
[3] Aristotle University (Greece)
[4] University of Peloponnese (Greece)

Abstract. Loop tiling is a well-known loop transformation that enhances data locality in memory hierarchy. In this paper, we initially reveal two important inefficiencies of current analytical loop tiling models and we provide the theoretical background on how current analytical models can address these inefficiencies. To this end, we propose a new analytical model which is more accurate that the existing ones. We showcase, both theoretically and experimentally, that the proposed model can accurately estimate the number of cache misses for every generated tile size and as a result more efficient tile sizes are opted. Our evaluation results provide high cache misses gains and significant performance gains over gcc compiler and Pluto tool on an x86 platform.

Keywords: loop tiling · data cache · data reuse · analytical model · cache misses

1 Introduction

Loop tiling is a loop transformation that exploits locality of data accesses in loop nests; the reused data stay in the cache and thus the number of cache misses is reduced. Although loop tiling does not always align with performance, it is one of the key optimizations for memory-bound loop kernels. The selection of an efficient tile size is of paramount importance as tiles of different sizes can lead to significant variations in performance. In this paper, we define a tile size as efficient if it achieves a reduced number of cache misses.

The two main strategies to address the tile size selection problem are analytical [16] and empirical [24]. The former refers to static approaches in which the tile size is selected based on static code analysis of the loop kernel and the memory configuration (number of caches, cache sizes, associativity, line size). Typically, the analytical model outputs the cache misses as a function of tile sizes, input size (of the executed kernel), and cache characteristics. The second strategy refers to empirical (experimental-based) approaches that rely on auto-tuning. In auto-tuning, the input program is executed multiple times assuming different tile sizes, until the best solution is found. The input program is considered as a black-box and no information of the source code is extracted.

In this paper, we first demonstrate two important inefficiencies of current analytical models and provide the theoretical background on how current models can address these inefficiencies. Second, we propose a new more accurate analytical model for loop tiling, for single-threaded programs.

The first drawback of current analytical models is that they do not accurately calculate the tiles sizes and as a consequence additional unforeseen cache misses occur (not captured by the model). The second drawback is that the tiles cannot remain in the cache in most cases due to the cache modulo effect. This is because the cache line size, cache associativity and data reuse of tiles, are not efficiently taken into account. Therefore, current models cannot accurately calculate the number of cache misses for each tile size, leading to sub-optimal tile sizes. On the contrary, the proposed method provides efficient tile sizes by accurately estimating the number of cache misses for each tile size.

Our experimental results show that by using our method it is possible to estimate the number of cache misses with an accuracy of about 1% using simulation and about 3% and 5.5% by using the processor's hardware counters on L1 data cache and L3 cache, respectively, leading to more efficient tile sizes for static loop kernels.

The remainder of this paper is organized as follows. In Section 2, the related work is reviewed. The proposed methodology is presented in Section 3 while experimental results are discussed in Section 4. Finally, Section 5 is dedicated to conclusions.

2 Related Work

In [20], an analytical model for loop tile selection is proposed for estimating the memory cost of a loop kernel and for identifying the optimal tile size. However, cache associativity is not taken into account. In [8], the authors combine loop tiling with array padding in order to improve the tile size selection process for specific array sizes. In [4], authors use Presburger formulas to express cache misses, but they fail to accommodate the high set associativity values of modern caches. In [16], an improved analytical model is proposed where associativity value is taken into account, but the cache hardware parameters (cache line size and associativity) and data reuse, are not efficiently taken into account.

As we showcase in this work there is ample room for improvement in existing analytical approaches, as cache line size and associativity and the arrays' memory access patterns, are not fully exploited.

Due to the problem of finding the optimum tile size is very complex and includes a vast exploration space [9], in addition to general methods, a large number of algorithm-specific analytical models also exist for Matrix-Matrix Multiplication (MMM) [12] [14], Matrix-Vector Multiplication [13], tensor contractions [15], Fast Fourier Transform [10], stencil [23] and other algorithms, but the proposed approaches cannot be generalized. In particular, regarding stencil applications, there has been a long thread of research and development tackling data locality and parallelism, where many loop tiling strategies have been proposed such as overlapped tiling [26] [5], diamond tiling [2] and others.

The second line of techniques for addressing the tile size selection problem relies on empirical approaches. A successful example is the ATLAS library [25]

which performs empirical tuning at installation time, to find the best tile sizes for different problem sizes on a target machine. The main drawback in empirical approaches is the enormous search space that must be explored.

Moreover, there are several frameworks able to generate tiled code with parameterized tiles such as PrimeTile [7] and PTile [1]. Parameterized tiling refers to the application of the tiling transformation without employing predefined tiles sizes, but inserting symbolic parameters that can be fixed at runtime [19]. In [1], a compile-time framework is proposed for tiling affine nested loops whose tile sizes are handled at runtime. In [19], authors present a formulation of the parameterized tiled loop generation problem using a polyhedral set. Pluto [3] is a popular polyhedral code generator including many additional optimizations such as vectorization and parallelization.

In [6], a thorough study on the major known tiling techniques is shown. In [21], authors use an autotuning method to find the tile sizes, when the outermost loop is parallelised. In [11], loop tiling is combined with cache partitioning to improve performance in shared caches. Finally, in [22], a hybrid model is proposed by combining an analytical with an empirical model. However, this model ignores the impact of set associativity in caches.

3 Proposed Methodology

3.1 Inefficiencies of Current Analytical Models

A. Current analytical models do not accurately calculate the tiles sizes Current methods, such as [16] [20], calculate the number of cache lines occupied by a tile, by using the following formula:

$$number.lines = \lceil \frac{tile.size.in.bytes}{line.size.in.bytes} \rceil \quad (1)$$

However, Eq. 1 is not accurate as different tiles (of the same size) occupy a varied number of cache lines. Let us give an example (Fig. 1). Consider an one-dimensional (1-d) array of 200 elements and non-overapping tiles consisting of 25 elements each. Also consider that each array element is of 4 bytes and the cache line size is 64 bytes. The array elements are stored into consecutive main memory locations and thus into consecutive cache locations. Current methods assume that each tile occupies two cache lines ($\lceil \frac{25 \times 4}{64} \rceil = 2$) (Eq. 1), therefore just two cache misses are assumed when loading the tile into the cache. However, as it can be shown in Fig. 1, half of the tiles occupy two cache lines and the other half occupy three cache lines.

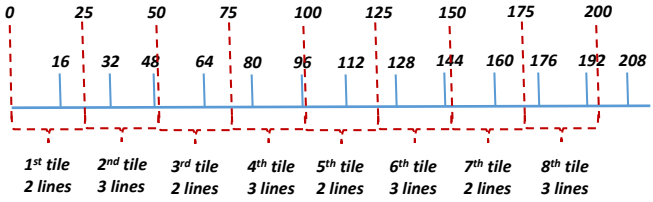

Fig. 1. An 1-d array is partitioned into tiles. 25 element tiles occupy a varied number of cache lines

The number of cache lines occupied by a tile is given by Eq. 2, where $a = 0$ or $a = 1$, depending on the tile size and cache line size values.

$$number.cache.lines = \lceil \frac{tile.size.in.bytes}{line.size.in.bytes} \rceil + a \quad (2)$$

There are cases where the tiles occupy a varied number of cache lines (e.g., in Fig. 1, $a = 0$ holds for some tile sizes and $a = 1$ holds for others) and cases where the tiles occupy a constant number of cache lines.

To ascertain that the tiles remain in the cache, in Subsection 3.2, we show that the cache size allocated must equal to the largest tile size value.

B. The tiles proposed by current analytical models cannot remain in the cache Related works such as [20] assume that if the aggregated size of the tiles is smaller than the cache size, then the reused tiles will remain in the cache; however, this holds true only in specific cases because even the elements of a single tile might conflict with each other due to the cache module effect. An improved model is proposed in [16], where the cache associativity value is taken into account, but still the tiles cannot remain in the cache in many cases, leading to a significant number of unforeseen cache misses.

Let us showcase the above problem with another example, the well-known Matrix-Matrix Multiplication (MMM) algorithm (Fig. 2). Although different tiles of A and B are multiplied by each other, the tile of C is reused $N/Tile$ times (data reuse), where $Tile$ is the tile size and N is the arrays size in each dimension. The current analytical models, such as [16], will consider data reuse in this case and therefore they will include this information to their cache misses calculation model; therefore, current models do assume that the tile of C is loaded just once in the cache, not $N/Tile$ times, which is accurate. However, **the tile of C cannot remain in the cache unless all the following three bullets hold** (in current analytical models only the first condition is satisfied):

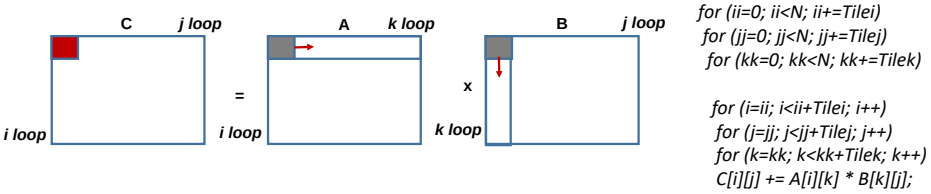

Fig. 2. An example. Loop tiling for MMM algorithm

- **Each tile must contain consecutive memory locations**
  The sub-rows of tile of C are not stored into consecutive main memory locations and therefore cache conflicts occur due to the cache module effect. A solution to this problem is array copying transformation; an extra loop kernel is added prior to the studied loop kernel where it copies the input array to a new one, in a tile-wise format; therefore, the tile elements are stored in consecutive main memory locations.
- **A cache way must not contain more than one tiles, unless they are stored into consecutive memory locations.**

Assume an L1 data cache of 32KB 8-way associative and $(Tilei, Tilej, Tilek) = (112, 32, 32)$; the size of tile of C,A and B is 14336 ($Tilei \times Tilej \times 4$ bytes), 14336 and 4096 bytes, respectively (32768 bytes in total) and they occupy $(3.5, 3.5, 1)$ cache ways, respectively (each way is 4096 bytes), assuming that each element is 4 bytes. Therefore, one cache way will be used to store part of the tiles of C and A (Way-0 in Fig. 3). In this case, Way-0 will store part of Tile of C and part of A; when the next tiles of A are loaded into the cache, they will be stored into different cache lines and therefore part of the C tile will be removed from the cache due to the cache module effect. This problem does not occur when $(Tilei, Tilej, Tilek) = (64, 64, 32)$, as the tiles occupy $(4, 2, 2)$ cache ways, respectively (Fig. 3).

For the reminder of this paper, we will be writing that a tile is written in a separate cache way if an empty cache line is always granted for each different modulo (with respect to the size of the cache) of the tile memory addresses, e.g., in Fig. 3, the tile in red is written in two 'separate' cache ways as an empty cache line is always granted for each different cache modulo value.

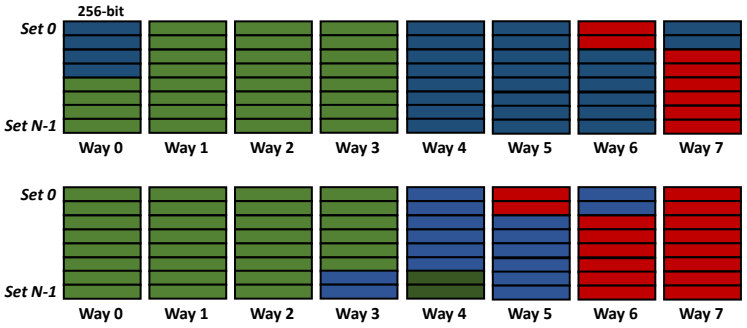

Fig. 3. An illustration of how tiles might be allocated to the cache, for the example shown in Fig. 2. On the top, $(Tilei, Tilej, Tilek) = (112, 32, 32)$ is shown, while in the bottom $(Tilei, Tilej, Tilek) = (64, 64, 32)$. Each tile is shown in a different colour.

– **Extra cache space must be granted for the non-reused tiles**
Even if the two aforementioned bullets hold, it is false to assume that the C tile will remain in the cache just because the aggregated size of the three tiles is smaller than the cache size. This is because there is no cache space allocated for the next tiles of A and B; therefore, when the next tiles of A and B are loaded into the cache they will evict cache lines from the tile of C (LRU cache replacement policy is assumed). However, if $(Tilei, Tilej, Tilek) = (64, 64, 16)$ is selected instead of $(Tilei, Tilej, Tilek) = (64, 64, 32)$, then cache space for 2 tiles of A and B is allocated and therefore the Tile of C will remain in the cache.

We evaluated the above assumptions on a PC (see Section 4) using Cachegrind tool [17] (simulation) and the following tile sets $(Tilei, Tilej, Tilek) = (112, 32, 32)$, $(64, 64, 32)$, $(64, 64, 16)$ give $(10.2, 9.8, 5.2)$ million dL1 misses and $(3.1, 3, 3, 7.4)$ Gflops, respectively (square matrices of size $N = 1344$).

3.2 The Proposed Analytical Model

Our approach is given in Algorithm 1. The proposed method generates the iterators to be tiled, their order as well as their tile sizes, for a given cache memory.

STEP.1: The iterators that loop tiling is applicable to are manually provided; not all the loops are eligible to loop tiling mainly because of dependencies.

Algorithm 1 Proposed Loop Tiling Algorithm

```
Step.1 Specify the iterators that loop tiling is eligible to (n iterators)
for (i=1,n) do
  Step.2 Generate all different iterator orderings using i out of n iterators
  for (each different ordering found in Step.2) do
    Step.3 Construct Eq. 3. Eq. 3 holds all the tiles sizes that fit and remain in
    the cache
    if (If the tiles' memory locations overlap) then
      Step.4 Merge the tiles into one and update Eq. 3
    end if
    for (each different tile size) do
      if (the tiles contain non-consecutive memory locations) then
        Step.5 Either discard this tile size or use array copying transformation
      end if
      Step.6 Estimate the number of cache misses for each tile set
    end for
  end for
end for
Step.7 Choose the tile set achieving the minimum number of cache misses
```

STEP.2: The next step is to specify the iterators that loop tiling will be applied to as well as their nesting level values. For example, in a loop kernel with three iterators (i, j, k) eligible to loop tiling, such as the original (non-tiled) version of MMM in Fig. 2, the following 15 loop tiling implementations will be generated: (i), (j), (k), (i, j), (i, k), (j, i), (j, k), (k, i), (k, j), (i, j, k), (i, k, j), (j, i, k), (j, k, i), (k, i, j), (k, j, i). All different orderings are processed so as not to exclude any efficient implementations.

STEP.3: In Steps.3-6, the main part of the proposed loop tiling algorithm takes place. First, a mathematical inequality is constructed holding the tile sizes for which the tiles fit and remain in the cache:

$$m \leq \lceil \frac{Tile_1}{L_i/assoc} \rceil + \lceil \frac{Tile_{1\_next}}{L_i/assoc} \rceil + ... + \lceil \frac{Tile_n}{L_i/assoc} \rceil + \lceil \frac{Tile_{n\_next}}{L_i/assoc} \rceil \leq assoc \quad (3)$$

where $Tile_i$ is the tile size in bytes, L_i is the cache size in bytes, n is the number of tiles, $assoc$ is the L_i associativity and m defines the lower bound of the tile sizes and it equals to the number of arrays in the loop kernel. The tile sizes not included in Eq. 3 are discarded as they cannot remain in the cache.

In Eq. 3, a separate tile exists for each array reference (in the loop kernel) and thus an array might have multiple tiles. Furthermore, for each tile, we grant cache space for its next tile too (to address the third bullet in section 3.1.2). Note that the overlapping tiles are merged into Step.4. All the tiles contain consecutive memory locations (1st bullet in Subsection 3.1.2). The value of $(\lceil \frac{Tile_1}{L_i/assoc} \rceil)$ is an

integer representing the number of L_i cache ways used by Tile1, or equivalently, is an integer representing the number of L_i cache lines with identical cache addresses used for Tile1. Eq. 3 satisfies that the array tiles directed to the same cache subregions do not conflict with each other as the number of cache lines with identical addresses needed for the tiles is not larger than the *assoc* value (second bullet in section 3.1.2).

$Tile_i$ which contains consecutive memory locations is given by Eq. 4:

$$Tile_i = max.number.cache.lines \times cache.line.size \times element.size \quad (4)$$

where *cache.line.size* is the size of the cache line in elements, *element.size* is the size of the array's elements in bytes and the *max.number.cache.lines* gives the maximum number of cache lines occupied by the tile (Eq. 2).

Step.4: In this step, the overlapping tiles in Eq. 3 are merged to one, normally bigger tile, which consists of their union; if the tiles match, then the new tile's size remain unchanged. Step.4 is needed so as there are no tile duplicates in the cache. For the rest of this paper we will write that two tiles overlap, if their memory locations overlap.

Consider the example where the following two array references exist in the loop body $A[i][j-2]$, $A[i][j+2]$ and j loop spans from 2 to N-2. By applying loop tiling to j loop with tile size T, the 1st tile of the 1st array reference spans within (0,T) and the 1st tile of the 2nd array reference spans within (4,T+4). These tiles are merged and a single bigger tile is created of size $(T+4)$.

Step.5 in Algorithm 1: In Step.5, all the remaining tile sizes with no consecutive memory locations are either discarded as they cannot remain in the cache or array copying transformation is applied.

It is common practice to apply array copying transformation before loop tiling in order all the tiles to contain consecutive memory locations. An extra loop kernel is added prior to the studied loop kernel where it copies the input array to a new one, in a tile-wise format. This adds an extra overhead and this is why it is performance efficient only in limited number of loop kernels.

Step.6 in Algorithm 1: In Step.6, the number of cache misses is approximated theoretically, considering the cache hardware parameters, the array memory access patterns of each loop kernel and the problem's input size. To do so, we calculate how many times the selected tiles (whose dimensions and sizes are known) are loaded/stored from/to the cache.

We are capable of approximating the number of cache misses because the number of unforeseen misses has been minimised (the reused tiles remain in the cache). This is because only the proposed tiles reside in the cache, the tiles are written in consecutive memory locations, an empty cache line is always granted for each different modulo and we use cache space for two consecutive tiles and not one (when needed). Additionally,we refer to CPUs with an instruction cache; in this case, the program code typically fits in L1 instruction cache; thus, it is assumed that the shared cache or unified cache (if any) is dominated by data.

The number of cache misses is estimated by Eq. 5.

$$Num\_Cache\_Misses = \sum_{i=1}^{i=sizeof(Tiles.List)} (repetition\_i \times cache.lines\_i) \quad (5)$$

where *repetition_i* gives how many times the array of this tile is loaded/stored from/to this cache memory (given by Eq. 7), *cache.lines_i* is the number of cache

lines accessed when this tile traverses the array (given by Eq. 6) and $Tiles.List$ contains all the tiles that contribute to Eq. 5.

The $Tiles.List$ is initialised with all the tiles specified in Eq. 3, after the merging process (Step.3b) in Algorithm 1 (the 'next' tiles are not included; the only reason they exist in Eq. 3 is to grant extra cache space). There are cases where not all the tiles contribute to Eq. 5 and this is why some tiles might be deleted from the $Tiles.List$. This happens when an array has multiple array references (in the loop body) and therefore multiple tiles. Thus, different tiles of the same array might access memory locations that have already been accessed just before and thus the tile resides in the cache; in this case, accessing the tile will lead to a cache hit, not a miss.

The *cache.lines* value in Eq. 5 is given by

$$cache.lines = \begin{cases} \frac{N}{Ty} \times \sum_{j=1}^{j=M/Tx} (\lceil \frac{j \times Tx}{line} \rceil - \lfloor \frac{(j-1) \times Tx}{line} \rfloor), \text{row-wise data array layout} \\ \sum_{j=1}^{j=tiles} (\lceil \frac{j \times (Tx \times Ty)}{line} \rceil - \lfloor \frac{(j-1) \times (Tx \times Ty)}{line} \rfloor), \text{tile-wise} \end{cases} \tag{6}$$

where (Tx, Ty) are the tile sizes of the iterators in the (x,y) dimension of the array's subscript, respectively, (N, M) are the corresponding iterators' upper bounds (for 1D arrays $Ty = 1$), $line$ is the cache line size in elements, $tiles$ is the total number of the array's tiles and $(tiles = N/Ty \times M/Tx)$ or $(tiles = M/Tx)$ whether for 2D/1D arrays, respectively.

Let us give an example for the first branch of Eq. 6, consider a 2D floating point array and a tile of size (10×10) traversing the array in the x-axis. Also consider that $(line = 16)$ array elements. The first tile occupies $10 \times (\lceil \frac{10}{16} \rceil - \lfloor \frac{0}{16} \rfloor) = 10$ cache lines while the second tile occupies $10 \times (\lceil \frac{20}{16} \rceil - \lfloor \frac{10}{16} \rfloor) = 20$ cache lines. Although the array's tiles are of equal size, they occupy a different number of cache lines. Eq. 6, gives the number of cache lines occupied in the case where array copying has been applied and therefore the array is written tile-wise in memory; in this case, the first tile lies between $(0, 100)$, the second between $(100, 200)$ etc.

The *repetition* value in Eq. 5 is given by

$$repetition = \prod_{j=1}^{j=U} \frac{(up_j - low_j)}{T_j} \times \prod_{k=1}^{k=Q} \frac{(up_k - low_k)}{T_k} \tag{7}$$

where U is the number of new/extra iterators (generated by loop tiling) that a) do not exist in the corresponding array's subscript and b) exist above of the iterators of the corresponding array, e.g., regarding the B tile in Fig. 2, this is the ii iterator. Q is the number of new/extra iterators that a) do not exist in the array and b) exist between of the iterators of the array, e.g., regarding the A tile in Fig. 2, this is the jj iterator; the ii iterator forces the whole array of B to be loaded $N/Tile$ times, while the jj iterator forces the whole array of A to be loaded $N/Tile$ times.

4 Experimental Results

The experimental results are extracted in a host PC (Intel i7-4790 CPU at 3.60GHz, Ubuntu 18.04) and the codes are compiled using gcc 7.5.0 compiler.

The benchmarks used in this study consists of six well-known memory-bound loop kernels taken from 4.1 PolyBench/C suite [18]. These are: gemm, mvm,

gemver, Doitgen, Bicg and gesumv. The input size of the loop kernels is specified with letter 'N' (square matrices are taken of size $N \times N$).

Table 1. The error in cache misses is measured for five different tile sizes using Eq. 8 and the maximum value is shown.

		dL1 cache				L3 cache			
kernel		**Cachegrind**	**Perf (HW counters)**	**Cachegrind**	**Perf (HW counters)**	**Cachegrind**	**Perf (HW counters)**	**Cachegrind**	**Perf (HW counters)**
gemm	Input size	**N=1000**		**N=2000**		**N=3000**		**N=6000**	
	Tile sizes	(25,25,25), (40,40,25) (-,2,-),(-,4,-),(25,25,40)		(20,20,50),(25,25,25) (25,25,40),(-,2,-),(40,40,25)		(500,500,600), (-,250,-) (500,500,500) (-,200,-),(600,600,300)		(500,500,600), (-,100,-) (500,500,500) (600,600,300), (-,75,-)	
	Error	**0.8%**	**2.8%**	**0.8%**	**2.9%**	**0.8%**	**5.4%**	**0.8%**	**5.8%**
mvm	Input size	**N=6000**		**N=9000**		**N=10000**		**N=12000**	
	Tile sizes	(-,2000), (-,1000), (6,1000) (3,2000), (4,1000)		(-,2000), (-,1000), (6,1000) (3,2000), (4,1000)		(100,2000), (50,2000), (100,2500) (50,2500), (125,2500)		(20,2000), (40,2000), (80,2000) (80,3000), (60,3000)	
		0.7%	**2.8%**	**0.7%**	**2.7%**	**0.7%**	**1.9%**	**0.7%**	**2.0%**
doitgen	Input size	**N=128**		**N=256**		**N=512**		**N=600**	
	Tile sizes	(-,32,32,32), (-,16,16,64) (-,16,64,16) (-,64,16,16),(-,16,16,32)		(-,32,32,32), (-,16,16,64) (-,16,64,16) (-,64,16,16),(-,16,16,32)		(-,512,256,512), (-,512,512,256) (-,256,256,256) (-,256,512,256),(-,512,512,512)		(-,600,300,600), (-,600,600,300) (-,300,600,600) (-,300,300,600),(-,600,300,300)	
	Error	**0.9%**	**2.5%**	**0.9%**	**3.1%**	**0.9%**	**2.5%**	**0.9%**	**2.6%**
gemver	Input size	**N=6000**		**N=9000**		**N=10000**		**N=12000**	
	Tile sizes	(-,1500), (-,1000), (2,1000) (3,1000), (1,1000)		(-,1500), (-,1000), (2,1000) (3,1000), (1,1000)		(100,2000), (50,2000), (125,2000) (50,1250), (80,1250)		(20,2000), (40,2000), (80,2000) (80,1500), (60,1500)	
	Error	**0.9%**	**2.9%**	**0.8%**	**2.9%**	**0.8%**	**2.0%**	**0.8%**	**2.0%**
bicg	Input size	**N=6000**		**N=9000**		**N=10000**		**N=12000**	
	Tile sizes	(3,1000), (1,1000), (2,1000) (-,1500), (-,1000)		(-,1500), (-,1000), (2,1000) (3,1000), (1,1000)		(100,2000), (50,2000), (125,2000) (50,1250), (80,1250)		(20,2000), (40,2000), (80,2000) (80,1500), (60,1500)	
	Error	**0.9%**	**2.9%**	**0.8%**	**2.9%**	**0.8%**	**2.0%**	**0.8%**	**2.0%**
gesumv	Input size	**N=4000**		**N=8000**		**N=8000**		**N=12000**	
	Tile sizes	(2,800), (1,1000), (2,1000) (-,800), (-,1000)		(2,800), (1,1000), (2,1000) (-,800), (-,1000)		(40,2000), (40,1000), (20,2000) (20,1000), (25,2000)		(20,2000), (20,1500), (10,2000) (30,2000), (30,1500)	
	Error	**0.9%**	**2.7%**	**0.9%**	**2.7%**	**0.9%**	**2.1%**	**0.9%**	**2.1%**

4.1 Validation of the proposed methodology

In this sub-section we showcase that i) the tiles generated by the proposed methodology fit and remain in the cache and ii) the proposed equations (Step.6) can accurately estimate the number of cache misses. To validate the proposed method, we have applied the proposed methodology to L1 data cache (dL1) (32KB, 8-way) and L3 cache (8MB, 16-way). The tile sizes and the iterators to be tiled are given by Algorithm 1.

The number of cache misses is measured for five tile sizes and the maximum error value is calculated (Eq. 8) using i) Cachegrind tool [17] (simulation) and ii) Perf tool using the 'l1d.replacement', 'LLC-load-misses' and 'LLC-store-misses' hardware counters.

$$error\% = \frac{|cache.misses.measured - Eq.\ 5.misses|}{Eq.\ 5.misses} \times 100 \quad (8)$$

Cachegrind and Perf give different cache misses values, because the perf measures the number of cache misses of all the running processes, not just the process we are interested in.

In Table 1, we compare the dL1 and L3 misses as extracted from Eq. 5 against the measurements from Cachegrind and Perf. As Table 1 indicates the proposed equations provide roughly the same number of cache misses as Cachegrind. This

means that first, the proposed tiles fit and remain in the cache and second, the proposed equations give a very good approximation of the number of misses.

Regarding dL1, the error values are higher (about 3%) when using the dL1 hardware counter (Table 1), as other processes are loading/storing data from/to this memory too. Note that Table 1 shows only the tile sizes that need roughly the size of seven out of eight cache ways, or less; the tiles that use more cache space give a much higher error value, which is up to 20%. Given that this inconsistency holds only for the Perf measurements and not for Cachegrind, it is valid to assume that this is due to the fact that other processes using the dL1. In this case, each dL1 access of another process leads to an unforeseen miss.

For the same reason, on the right of Table 1, we show the tile sizes that need roughly the size of 9 out of 16 L3 cache ways, or less. mvm, doitgen, bicg, gesumv and gemver give a small L3 error value as their arrays fit and remain in L3 even without using loop tiling. This is not the case for gemm and this is why the error value in gemm is higher.

Table 2. Comparison over gcc on Intel i7-4790.

	Tiling for dL1 only							**Tiling for dL1 and L3**						
kernel	**dL1 misses gain**	**perf. gain**	**tile size**	**dL1 misses gain**	**perf. gain**	**tile size**	**Pluto perf. gain**	**L3 misses gain**	**perf. gain**	**tile size**	**L3 misses gain**	**perf. gain**	**tile size**	**Pluto perf. gain**
gemm	**N=600**			**N=900**				**N=1800**			**N=3400**			
	x40.9	x3.7	(60,60,10)	x61.0	x4.1	(60,60,10)	1.01	x57.2	x4.09	(900*,60,900*) (60,60,10)	x59.3	x4.2	(850*,50,850*) (50,50,20)	1.01
mvm	**N=9000**			**N=12000**				**N=9000**			**N=12000**			
	x1.5	x0.98	(-,3000)	x1.5	x0.98	(-,3000)	x0.91	x1.00	x1.01	(120,3000)	x1.00	x1.01	(100,3000)	x0.91
doitgen	**N=256**			**N=512**										
	x34.2	x1.79	(64,64,16)	x41.4	x1.93	(64,64,16)	x0.99	-	-	-	-	-	-	-
gemver	**N=8000**			**N=12000**				**N=8000**			**N=12000**			
	x2.0	x1.08	(-,2000)	x2.0	x1.09	(-,2000)	x0.89	x1.00	x1.09	(80,2000)	x1.00	x1.18	(60,2000)	x0.89
bicg	**N=8000**			**N=12000**				**N=8000**			**N=12000**			
	x2.0	x0.92	(-,2000)	x2.0	x0.92	(-,2000)	x0.42	x1.00	x1.03	(80,2000)	x1.00	x1.04	(60,2000)	x0.42
gesumv	**N=8000**			**N=12000**				**N=8000**			**N=12000**			
	x1.25	x0.91	(-,2000)	x1.25	x0.91	(-,2000)	x0.87	x1.00	x1.01	(40,2000)	x1.00	x1.01	(30,2000)	x0.87

4.2 Evaluation over gcc compiler and Pluto

In all cases, the six studied loop kernels are compiled using 'gcc -O2 -floop-block -floop-strip-mine' command and the generated binaries are those that the proposed methodology is compared to. The '-floop-block -floop-strip-mine' option enables gcc to apply loop tiling transformation. The C codes of the proposed method are compiled using 'gcc -O2' command.

On the left of Table 2, the proposed methodology has been applied to dL1 only. The proposed method provides significant dL1 miss gains at all cases but performance gains just for gemm, doitgen and gemver. Reducing the number of dL1 misses does not always align with performance; in this case, the selected tile sizes for mvm, bicg and gesumv (which minimize dL1 misses) slighly increase the number of L3 misses and this is why performance is degraded. Note that

the dL1 miss gain is higher in gemm and doitgen comparing to the other loop kernels, as all their tiles achieve data reuse; the tiles remain in L1 and also being loaded many times from L1, highly reducing the number of L1 misses.

It is important to note that the baseline binary code that we compare our method to for mvm,bicg and gesumv in Table 2, does not include loop tiling (although the loop tiling option has been enabled, gcc disables its application in gesumv, bicg and mvm, by considering it not performance efficient).

On the right of Table 2, the proposed methodology has been applied first to dL1 and then to L3. Applying loop tiling for mvm, gemver, bicg and gesumv just for L3 cache is pointless as their arrays fit and remain in the cache even for very large input sizes and as a consequence the number of L3 misses cannot be reduced. However, applying loop tiling for L3 to the implementations shown on the left of Table2 is beneficial, as these implementations give a higher number of L3 misses than the no tiled implementations. Regarding doitgen, applying loop tiling to L3 cannot give any gain as the arrays fit in the cache. The '*' in Table 2 indicates that these iterators are interchanged.

The proposed methodology has been also evaluated using Pluto [3] (version 0.11.4). For a fair comparison, only the loop tiling phase of Pluto is activated. Pluto applies square tile sizes of size 32 at all cases and this is why gcc perfomrs better. Pluto is a powerful tool which is not limited to loop tiling and if we enable all its phases, then it provides higher speedup values than gcc.

5 Conclusions and Future Work

In this article, we first demostrate two important inefficiencies of current analytical loop tiling models and provide insight on how current models can overcome these inefficiencies. Second, we propose a new model where the number of cache misses is accurately estimated for each generated tile size. This is achieved by leveraging the target memory hardware architecture and data access patterns.

As far as our future work is concerned, the first step includes the validation and evaluation of the proposed method to other CPUs. Second, we plan to work towards correlating the number of cache misses with execution time.

Acknowledgements

This work has received funding from the European Unions Horizon 2020 research and innovation programme under Grant Agreement No 957210 - XANDAR: X-by-Construction Design framework for Engineering Autonomous & Distributed Real-time Embedded Software Systems.

References

1. Baskaran, M.M., Hartono, A., Tavarageri, S., Henretty, T., Ramanujam, J., Sadayappan, P.: Parameterized tiling revisited. CGO '10 (2010)
2. Bondhugula, U., Bandishti, V., Pananilath, I.: Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. IEEE TPDS pp. 1285–1298 (2017)
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN **43**(6) (Jun 2008)
4. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. SIGPLAN Not. **36**(5), 286297 (May 2001)
5. Cohen, A., Zhao, J.: Flextended Tiles: a Flexible Extension of Overlapped Tiles for Polyhedral Compilation. ACM TACO (2020)

6. Hammami, E., Slama, Y.: An overview on loop tiling techniques for code generation. 2017 IEEE/ACS AICCSA pp. 280–287 (2017)
7. Hartono, A., Baskaran, M.M., Bastoul, C., Cohen, A., Krishnamoorthy, S., Norris, B., Ramanujam, J., Sadayappan, P.: Parametric multi-level tiling of imperfectly nested loops. In: ICS '09. p. 147157. ICS '09, New York, NY, USA (2009)
8. Hsu, C.h., Kremer, U.: A quantitative analysis of tile size selection algorithms. J. Supercomput. **27**(3), 279294 (Mar 2004)
9. Kelefouras, V., Djemame, K.: A methodology correlating code optimizations with data memory accesses, execution time and energy consumption. Journal of Supercomputing **75**(10), 6710–6745 (October 2019)
10. Kelefouras, V.I., Athanasiou, G.S., Alachiotis, N., Michail, H.E., Kritikakou, A.S., Goutis, C.E.: A methodology for speeding up fast fourier transform focusing on memory architecture utilization. IEEE Trans. on Signal Processing (2011)
11. Kelefouras, V., Georgios, K., Nikolaos, V.: Combining software cache partitioning and loop tiling for effective shared cache management. ACM Trans. Embed. Comput. Syst. **17**(3), 72:1–72:25 (May 2018)
12. Kelefouras, V., Kritikakou, A., Mporas, I., Kolonias, V.: A high-performance matrix—matrix multiplication methodology for cpu and gpu architectures. J. Supercomput. **72**(3), 804844 (Mar 2016)
13. Kelefouras, V., Kritikakou, A., Papadima, E., Goutis, C.: A methodology for speeding up matrix vector multiplication for single/multi-core architectures. J. Supercomput. **71**(7), 26442667 (Jul 2015)
14. Kelefouras, V.I., Kritikakou, A., Goutis, C.: A Matrix–Matrix Multiplication methodology for single/multi-core architectures using SIMD. Supercomputing, Springer (January 2014). https://doi.org/10.1007/s11227-014-1098-9
15. Li, R., Sukumaran-Rajam, A., Veras, R., Low, T.M., Rastello, F., Rountev, A., Sadayappan, P.: Analytical cache modeling and tilesize optimization for tensor contractions. In: SC '19 (2019)
16. Mehta, S., Beeraka, G., Yew, P.C.: Tile size selection revisited. ACM Transactions on Architecture and Code Optimization **10**(4) (Dec 2013)
17. Nethercote, N., Walsh, R., Fitzhardinge, J.: Building workload characterization tools with valgrind. In: IISWC. p. 2. IEEE Computer Society (2006)
18. POUCHET, L.: Polybench/c, http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/, accessed: 2020-10-10
19. Renganarayanan, L., Kim, D., Strout, M.M., Rajopadhye, S.: Parameterized loop tiling. ACM Trans. Program. Lang. Syst. **34**(1) (May 2012)
20. Sarkar, V., Megiddo, N.: An analytical model for loop tiling and its solution. In: IEEE ISPASS. pp. 146–153 (2000)
21. Sato, Y., Yuki, T., Endo, T.: An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation. ACM TACO (2019)
22. Shirako, J., Sharma, K., Fauzia, N., Pouchet, L.N., Ramanujam, J., Sadayappan, P., Sarkar, V.: Analytical bounds for optimal tile size selection. CC'12 (2012)
23. Stoltzfus, L., Hagedorn, B., Steuwer, M., Gorlatch, S., Dubach, C.: Tiling optimizations for stencil computations using rewrite rules in lift. ACM Transactions on Architecture and Code Optimization **16**(4) (Dec 2019)
24. Tavarageri, S., Pouchet, L.N., Ramanujam, J., Rountev, A., Sadayappan, P.: Dynamic selection of tile sizes. HIPC '11 (2011)
25. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing **27**(1–2), 3–35 (2001)
26. Zhou, X., Giacalone, J.P., Garzarán, M.J., Kuhn, R.H., Ni, Y., Padua, D.: Hierarchical overlapped tiling. CGO '12 (2012)