Toward modeling cache-miss ratio for Dense-Data-Access-Based Optimization

Riyane Sid Lakhdar, Henri-Pierre Charles, Maha Kooli

Toward Modeling Cache-Miss Ratio for Dense-Data-Access-Based Optimization

Riyane SID LAKHDAR
riyane.sidlakhdar@cea.fr
Univ Grenoble Alpes, CEA,
List, F-38000 Grenoble, France

Henri-Pierre CHARLES
Henri-Pierre.Charles@cea.fr
Univ Grenoble Alpes, CEA,
List, F-38000 Grenoble, France

Maha KOOLI
Maha.KOOLI@cea.fr
Univ Grenoble Alpes, CEA,
Leti, F-38000 Grenoble, France

ABSTRACT

Adapting a source code to the specificity of its host hardware represents one way to implement software optimization. This allows to benefit from processors that are primarily designed to improve system performance. To reach such a software/hardware fitting without narrowing the scope of the optimization to few executions, one needs to have at his disposal relevant performance models of the considered hardware. This paper proposes a new method to optimize software kernels by considering their data-access mode. The proposed method permits to build a data-cache-miss model of a given application regarding its specific memory-access pattern. We apply our method in order to evaluate some custom implementations of matrix data layouts. To validate the functional correctness of the generated models, we propose a reference algorithm that simulates a kernel's exploration of its data. Experimental results show that the proposed data alignment permits to reduce the number of cache misses by a factor up to 50%, and to decrease the execution time by up to 30%. Finally, we show the necessity to integrate the impact of the *Translation Lookaside Buffers* (TLB) and the memory prefetcher within our performance models.

ACM Reference Format:
Riyane SID LAKHDAR, Henri-Pierre CHARLES, and Maha KOOLI. 2019. Toward Modeling Cache-Miss Ratio for Dense-Data-Access-Based Optimization. In *30th International Workshop on Rapid System Prototyping (RSP'19) (RSP'19), October 17–18, 2019, New York, NY, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3339985.3358498

1 INTRODUCTION

Over the last few decades, each new generation of hardware (HW) platform overcame the former one by introducing a brand new approach for memory storage or computation. This has led High-Performance-Computing (HPC) developers to have a very large set of potential software (SW) optimizations methods applicable to their code. However, the considered HW platforms are very heterogeneous in terms of Instruction Set Architecture (ISA), registers and memory hierarchy. Consequently, the most commonly-used SW optimizations are specific to the family of the given host HW. As an example, for a simple matrix multiplication application, [12] provides a set of SW optimizations that may be implemented and the corresponding gain. We note from this example that the code-efficiency is reached at the expense of its portability, i.e. through code specialization and using hardware-specific instructions. Thus, porting a given source code to a new family of HW and benefiting from its performance may require an important engineering effort.

The objective of our study is to walk a first step toward automatic code-adaptation of a given SW kernel to its specific host HW. In this paper, we present a custom method to estimate the data-cache-miss ratio depending on HW parameters related to the memory hierarchy (e.g. cache-line size, number of cache lines). The generated estimation depends also on some OS-related parameters such as the memory-allocation policy. The proposed method is specifically designed to be embedded within a SW platform to automatically detects the adequate code optimizations to apply to a given source code. The objective being to make it fit specifically its host HW. To illustrate our method, we consider the example of matrices accessed through memory-patterns similar to simple matrix multiplication and convolution algorithms [14]. We consider matrix lines and columns that have been dynamically allocated (within the heap of the process). We assume that the data-caches implement the *"least-recently used"* (LRU) cache-replacement policy. No assumption is made on the cache-write policy. Using the generated models, we pick the ideal data-layout implementation for the considered kernel. This allows to reduce the number of cache misses by a factor up to 50%, and to decrease the execution time up to 30%. It is noteworthy that our objective is not to propose a new matrix multiplication or convolution algorithm. Other specialized implementations and libraries ([17], [9] and [16]) exist and are proven to reach peak performance. However, we pick this example as a visual way to illustrate our *general-purpose* method.

The rest of the paper is organized as follows. Section 2 discusses the state-of-the-art. Section 3 provides a background on data-layout families to store matrices. Section 4 details the proposed method to build a data-cache-miss model parametrized. Section 5 experiments the performance-interest of the matrix data-layouts. Finally, section 6 concludes the paper.

2 STATE OF THE ART

Code optimization is a potentially endless topic in modern computer science. Various research solutions have been widely

RSP'19, October 17–18, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6847-6/19/10…$15.00
https://doi.org/10.1145/3339985.3358498

studied, ranging from *just-in-time compilation* [13] to *polyhedral compilation* [2] and *source-to-source transformation* [7]. We group some of these solutions in Table 1 and classify them according to their leverage point within an application's life cycle. We note from this table that the ecosystem of SW optimization is highly detached from HW consideration. No particular attention is given to the specificity of the host hardware.

	Algorithm	Source Code	Compilation (static)	Compilation (dynamic)	Post-mortem
FFTW [9]	X	X	X		
JIT [13]			X	X	X
Magma [17]		X	X		
deGoal [3]		X	X	X	
LGen [16]	X	X	X		
BOAST [7]	X	X	X		
Polyhedral compilation [2]		X	X		

Table 1: Classification of existing code-optimization solution according to their leverage-point on source code

Meanwhile, code specialization to a specific host hardware is still a barely-explored way of code optimization. The evidence lies in the fact that most existing hardware-performance models, which are crucial for such a approaches, are hardly exploitable in automatic SW optimization. The example of the *roofline* model [18] is representative of this trend. Indeed, this model is a couple of constant thresholds the performance of a kernel can not exceed. Thus, it cannot be used to spot the hardware parameters (such as the cache sizes, associativity or replacement policy) that may influence most the kernel's performance.

Our aim through this paper is to propose a method to build cache-miss models that exhibits a higher correlation between the source-code performance and the host-hardware characteristics. To the best of our knowledge, the closest data-cache-miss modeling methods similar to our work are proposed in [19] and [1]. These methods estimate the data-cache-miss model of a program by analyzing its corresponding *data-reuse* distance (defined as the number of distinct data elements accessed between two consecutive reference to the same element [19]). Unlike our method, these methods exhibit models that do not depend on the considered memory-access pattern of the algorithm but on the input data used during the test execution. Thus, the models generated by these methods are only valid for a specific set of input data.

3 BACKGROUND: MATRIX DATA-LAYOUTS

When dealing with memory for performance-optimization, two aspects, highly interleaved, need to be considered: the memory layout used for its storage and the pattern followed to access the addresses.

In this section, we give an overview of the *data-layout* architectures that we have considered. By *data-layout* we refer to the geometrical shape followed by data-addresses. The data-reorganizations that we deploy being at virtual-addresses level. We also assume memory access patterns similar to common matrix multiplication [5] and convolution [14].

3.1 Flattening 2D Structures Within 1D Array

A first way to access a cell (x,y) from a dynamically-declared uni-dimensional line-major matrix array is using equation: $@_{(x,y)} = @_{base} + (xN + y)D$. Thanks to its simplicity, this method allows to access a cell (at address $@_{(x,y)}$) in roughly one computation and only one memory access (assuming that the initial address $@_{base}$ of the array and the values of x and y are stored within processor-registers). Furthermore, keeping cells that belong to contiguous lines within the same block of addresses contributes to cache and page locality. Indeed, a residual from a matrix line (fetched within the caches) has a high probability to be used immediately afterward if it contains the following matrix line. It also leads to a high prediction-hit ratio for the prefetcher by creating a high regularity within the accessed addresses.

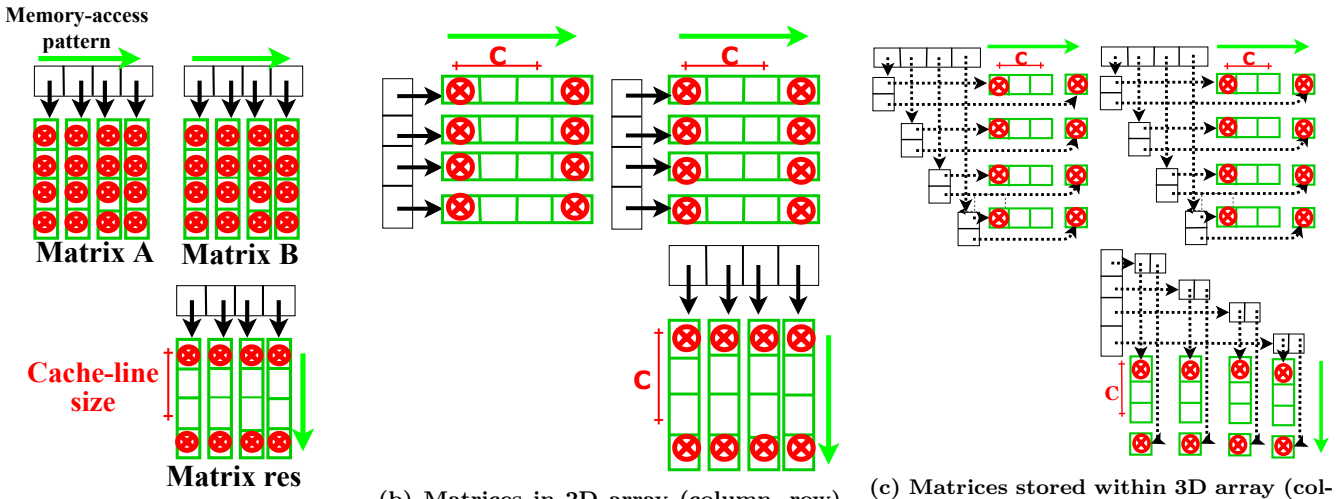
Given the growing impact of memory wall, this high locality and relatively-reduced number of memory-accesses represents an interesting performance advantage. The main limitation of this uni-dimensional data-layout family is related to the lack of scalability with respect to the number of concurrent threads. The relative proximity between addresses belonging to independent lines increases their probability to be set within the same cache-line. Accessing these addresses concurrently would thus trigger *false sharing* [10], which may increase the access time by up to 100 CPU cycles. Most methods ([15] and [4]) proposed to reduce *false sharing* are based on memory alignment. Thus, they can not be applied to the current data layout without prohibitively increasing its memory-access cost.

3.2 Multidimensional Matrix Storage

A second way to store a dynamically-declared matrix is using a multidimensional array. Each cell of the array is a pointer to either a payload-data array (Figure 1a) or a pointer to another pointer's array (Figure 1). Each array (from each dimension) is dynamically allocated independently from the others. Similar principles are used by the *java* implementation of N-dimensional data layouts.

The main advantage of using a multidimensional matrix storage is its modularity. The data may be split with respect to any dimension (projection on hyper plans) in order to fit an ideal workload distribution among threads or to suite the hardware and OS specifications (cache line size, virtual page size or process-fork buffer). Figure 1 shows how adapting the data-layout according to the access pattern helps to significantly reduce the number of cache misses. We may thus improve scalability with respect to the workload, number of threads[1] and hardware dimensions.

[1] It would for instance help implement the previous solutions for *false sharing*



(a) Matrices in 2D array (column, row)

(b) Matrices in 2D array (column, row) with mixed row/line-major layout

(c) Matrices stored within 3D array (column, row, cache line)

Figure 1: Memory access pattern and cache misses of matrix multiplication (naive algorithm) on three different implementations of a matrix stored on multidimensional data layout.

In the rest of this paper and for the sake of clarity, we will always consider multidimensional matrices as line-major. We will also assume a line-major exploration.

4 PROPOSED DATA-CACHE-MISS MODEL

In this section, we propose a method to accurately model the number of data-cache misses triggered while accessing each family of data-layout presented in section 3. The objective being to highlight the parameters (such as data alignment or block subdivision) that may have a significant impact on data-cache misses (when dealing with access-pattern similar to matrix multiplication and convolution).

4.1 Data-Cache-Miss Modeling on a 1D Memory Block

The data of the considered kernels is accessed sequentially. Consequently, the proposed method to build a cache-miss model is to first consider a simple 1D array of N elements at the address a_0 and where each element is of size D bytes. The data are being fetched within a cache made of C_{total} lines where each one is of size C. Throughout all our modeling process, we assume that all this constants belong to $\mathbb{N}^*$ (set of purely-positive natural numbers).We also assume that the array has not been covered yet (hence it is not present in the considered cache).

The number n_0 of cache-misses triggered while accessing the array sequentially is given by Equation 1 (the notation $a_0[C]$ refers to the rest of the euclidean division of a_0 by C).

$$n_0 = 1 + \left\lceil \frac{ND - (C - a_0[C])}{C} \right\rceil \tag{1}$$

Accessing one byte at an address a_0 leads to fetch the corresponding data at a position $a_0[C]$ within a cache line. The rest of the cache line being populated with the data at the addresses surrounding a_0. Consequently, at the initial data access (address a_0), $C - a_0[C]$ bytes from the array are fetched into the cache. The next data access to trigger a cache miss $(a_0 + C - a_0[C])$ will thus be aligned with a cache-line size C (because $a_0 + C - a_0[C] \equiv 0(modC)$). Then, the data are fetched by chunks of size C bytes. The number of fetch processed is found by dividing the number of remaining bytes after the first access $(N * D - (C - a_0[C]))$ by the size of one chunk (C). We then take the ceiling of the result to consider the case where the last chunk of the array is smaller than C.

In the context of simple matrix multiplication or convolution, a line j of the matrix is generally browsed after the previous one $j-1$. Given the previously described functioning of a cache, this results in an initial part L_j (bytes) of the array j being pre-loaded at the time we start accessing it. The Equation 2 shows how we have introduced this new parameter within the proposed cache-miss model in order to evaluate n_j (number of cache misses triggered while exploring the j^{th} matrix line). This is achieved by reducing the total number of bytes ND by L_j. We also increment the initial address a_j of the j^{th} line of the matrix by the same value.

$$n_j = \begin{cases} 0 & \text{if } L_j \geq ND \\ 1 + \left\lceil \frac{ND - L_j - (C - (a_j + L_j)[C])}{C} \right\rceil & \\ = \left\lceil \frac{ND + a_j}{C} \right\rceil - \left\lfloor \frac{a_j + L_j}{C} \right\rfloor & \text{if } L_j < ND \end{cases} \tag{2}$$

Finally, and for the same time-proximity reason, it is important that we know how much residual bytes[2] from a line of a matrix are loaded with each line. In Equation 3, we determine the model of that residual r_j for a given line j using the same kind of reasoning on the cache-fetch and positioning.

$$r_j = \begin{cases} 0 & \text{if } L_j \geq ND \\ C-(ND+a_j-1)[C]-1 & \\ = C\left\lfloor \frac{ND+a_j-1}{C}+1 \right\rfloor - ND - a_j & \text{if } L_j < ND \end{cases} \quad (3)$$

4.2 Extension of the Data-Cache-Miss Model to a 2D Memory Blocks

In the subsection 3.1, we considered the case where a matrix is stored within a 1D data layout. The number L_j of bytes already located in the cache when we start exploring the line j is exactly the number of residual bytes r_{j-1} of data loaded while fetching the line $j-1$.

Equation 4 gives the total number n^* of cache misses triggered during a multiplication of matrices stored within a 1D data layout. We obtain this formula by first replacing L_j by the expression of r_{j-1} in Equation 2 (using an initial value $L_0=0$). We then replace the address a_j by a_0+jND. Finally, we use *Weyl*'s criterion applied to rational numbers in order to sum the cache misses for each column.

$$n_j^{1D} = \begin{cases} \left\lceil \frac{ND+a_0}{C} \right\rceil - \left\lfloor \frac{a_0}{C} \right\rfloor \text{ if } j=0 \\ \left\lceil \frac{ND(j+1)+a_0}{C} \right\rceil + \left\lceil \frac{1-ND(j+1)-a_0}{C} - 1 \right\rceil \end{cases} \quad (4)$$

$$n^* = \sum_N^{j=0} n_j^{1D} = N * \sqrt[s]{\frac{ND+1}{C} - 1} + O(1)$$

In this section, we consider matrices stored within 2D arrays (as shown in subsection 3.2); the number L_j of pre-fetched bytes for a line j is no longer the number of residual bytes r_{j-1}[3]. In order to represent L_j, we need to model, for the given memory allocator, the distance between two consecutively allocated lines of the matrix.

In this context, we have considered *ptmalloc* (version 2.19), the heap allocator based on Dong Leas *Malloc* algorithm that has become the default *Linux GLIBC* implementation [11].

As shown on Figure 2, a memory block (also known as basic block) allocated using the *malloc* function of *ptmalloc* has a size that is a multiple of B, where B is usually equal to 8 or 16 depending on the processor architecture. The returned block contains a reserved section (tail) of size $T=8$ Bytes at its end. It is also allocated at an address that is a multiple of B (however the payload data maybe shifted within the basic block).

Allocating an array of size ND dedicates a basic block of size kB where k is the smallest strictly positive integer such that $kB \geq ND+T > (k-1)B$ (hence $k = \lceil \frac{ND+T}{B} \rceil$). Consequently, the distance D_j between two arrays of size ND allocated consecutively is $D_j = kB - ND - S$ where S is the shift of each array within its relative basic block.

[2]Residual is the number of bytes that do not belong to a given line but that are fetched along with it into the cache.

[3]Even though we still have $\forall j \in [1, N-1], L_j \leq r_{j-1}$

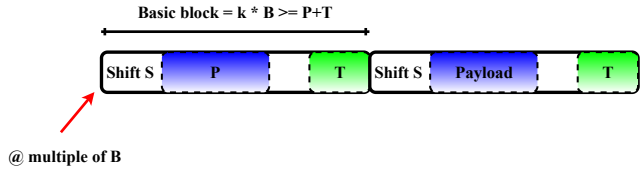


Figure 2: Example of two blocks of size P Bytes allocated using *ptmalloc*: Basic block of size multiple of B and reserved section of size T Bytes.

In Equation 5 we have estimated the number pre-fetched bytes L_J for an array j of a matrix. It is obtained by retaining the distance D_j from the number of residual bytes r_{j-1} of the previous array $j-1$.

$$L_j = \begin{cases} 0 \text{ if } j=0 \text{ or } D_j > r_{j-1} \\ \\ r_{j-1} - (kB - ND - a_{j-1}[kB]) \text{ else} \end{cases} \quad (5)$$

By injecting the expression of L_j into the general expression of n_j (see Equation 2), we obtain the number of cache misses n_j^{2D} triggered while exploring a line of a matrix stored within a 2D data layout:

$$n_j^{2D} = \left\lceil \frac{ND+a_j}{C} \right\rceil - \left\lfloor \frac{a_j - kB + a_{j-1}[kB]}{C} \right\rfloor + \left\lfloor \frac{ND-1}{C} + 1 \right\rfloor \quad (6)$$

5 RESULTS AND DISCUSSION

In this section, we experiment the correctness and the accuracy of the generated performance-models. In this paper, the accuracy refers to how precisely a model allows to spot the parameters (WH, SW and OS related) that may significantly influence the performance of the corresponding kernel. In fact, our objective is to find such parameters for a given source-code in order to tune them. Any further understanding for "accuracy" is not relevant for our approach of SW optimization. Consequently, we compare non of the cache-miss models that our method generates with experimental evaluations.

5.1 Experimental setup

All the presented performance results are obtained following the same experimental protocol. Each considered point is assessed (experimental run) 10 times[4]. The corresponding values that we present in Figure 3 and Figure 5 are the average of the results of these 10 runs. It is noteworthy that all the performance gain that we show has been obtained without changing the original algorithm nor re-ordering the instructions.

All the performance experimentations have been processed on an x86 HW architecture implementing an *Intel Xeon E3-1270 v4* processor with a l3 cache (LLC) containing a total of 8M Bytes made of 128 Bytes per cache line and implementing the LRU cache-replacement policy. A *Debian* (4.9.2) operating

[4]Between two consecutive experimentations, we make sure to flush all the considered data-caches using the *CFLUSH* instruction from the ISA of our *Intel* processor

system has been used based on the *Linux* (3.16.0-4) kernel. The *g++* (4.9.2) compiler (with the -03 optimization option) has been used to compile the considered computation-kernels (matrix multiplication) and matrix data structures. The *Perfmon2* library [8] has been used to access the performance management unit of the processor in order to measure different cache misses and CPU cycles.

5.2 Experimental Impact of Data Layout on Performance

In this section, we present an experimental evaluation of three custom implementations of the data layouts introduced in section 3. In Figure 3, we show how a proper alignment of the matrix lines (respectively columns) along with a proper orientation the matrix majors (according to the access pattern) allowed us to reduce the number of cache misses by a factor up to 50%. We also show how the reached cache-miss reduction can eventually lead to an execution-time improvement (up to 30%). Indeed, the Figure 3a confirms that the 1D data layout fits poorly to caches once a matrix line exceeds the size of a cache line (128 Bytes): a line of the matrix will most likely cross different cache lines leading to reject part of a matrix line while the algorithm still needs to access it.
Meanwhile, Figure 3c and Figure 3d show the clear advantage of using optimizations based on cache-alignment when dealing with multi-dimensional data structures (orange curve compared to the blue one). Furthermore, these figures show how transposing arrays (from row-major to column-major) allowed us to significantly reduce cache-misses when a matrix-line exceeds a cache line size (128 Bytes). In this case, returning from the end to the beginning of a matrix-line leads to consider a new cache line; which is likely to have been fetched-out.

5.3 Using the Models to Choose Software Optimizations

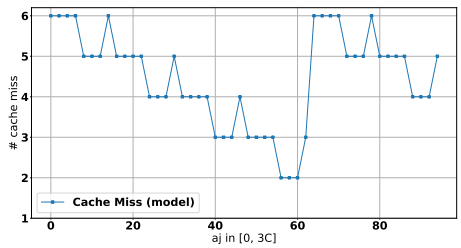


Figure 4: Number of cache misses triggered while sequentially accessing a single column (at the address a_j) of a matrix stored within a 2D data-layout using a cache line of size $C=32$ (Bytes). The array contains $N=10$ cells of size $D=2$ (Bytes) each. The considered allocator (ptmalloc) allocates blocks with a size multiple of $B=16$, containing a tail of size $T=8$ bytes.

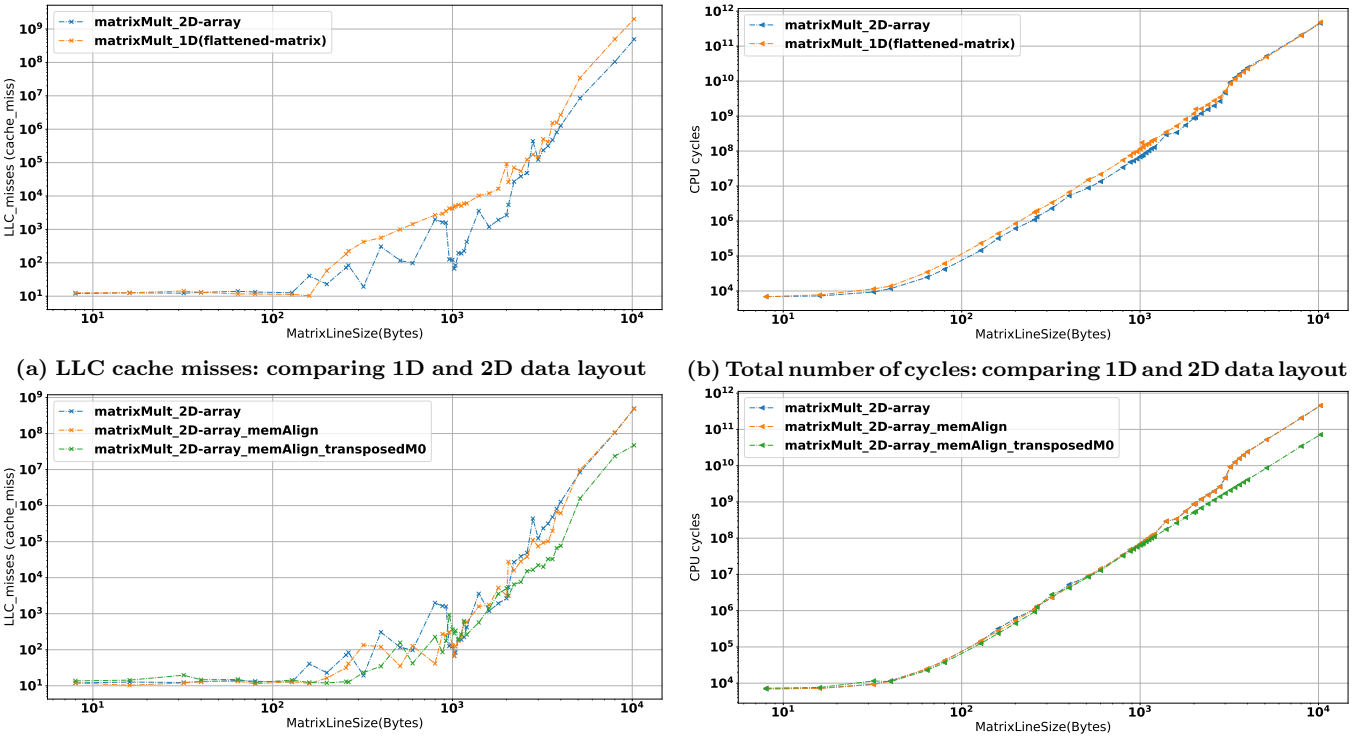
The model presented in Equation 2 shows that for a matrix stored within the presented 1D data structure, $\forall a_0 \in [0,C-1], n^*$ belongs to $[\frac{ND}{C}, \frac{ND-1}{C}+1]$. Thus, any optimization based on memory alignment (changing the value of a_0) regarding such a data layout would at most bring a gain of 1 cache miss; which is insignificant compared to the total cache miss number which is in order of magnitude of $N\sqrt[8]{N}$ (see Equation 4).

On the contrary, for a matrix stored within the presented 2D data-layout, memory alignment may bring a cache-miss improvement in an order of magnitude of N. Indeed, Figure 4 shows that by selecting an ideal address a_j for a line j of a matrix, we may gain i cache-misses while exploring it (with $i=5$ cache misses in the conditions of Figure 4). More generally, we show using *Abel*'s theorem that this improvement i is in $O(\frac{C}{N*D^2})$. Given that the size of the basic blocks used by the considered memory allocator is unique, the distance between two arrays allocated consecutively is unique. Consequently, this cache-miss improvement is identical for the N arrays of the matrix. Hence the $N*i$ cache-miss improvement for exploring the N arrays of the matrix.
It is noteworthy that in the case of operands for matrix multiplication, this number is in order of magnitude of N^2 given that each line (respectively column) is explored N times.

5.4 Extension of the Modeling to TLB-Misses and Date Prefetcher

For the sake of simplicity, we consider, in this paper, the number of data-cache misses as a direct cause of performance downgrade. We show on most of our experimental results in section 5 the validity of such an assumption. However, we acknowledge the fact that this relation is not always straight-forward. For instance, Figure 5 presents an experimental assessment of a multitreaded implementation of matrix multiplication, comparing two different 2D data layouts. We conclude from this figure that the cache-misses and cycle-counts are not always correlated. Our guess is that the prefetcher is responsible for many useless cache misses while miss-predicting addresses. This would principally explain the fact that our model is most often underestimating the experimental number of cache misses. As a future work, we are thus considering two approaches to deal with this experimental gap.
On one hand, we are trying to propose an experimental methodology to model the behavior of the prefetcher regarding our data-access pattern.
On the other hand, we are trying to propose a new data layout that would try to ease the pattern-detection work of the prefetcher. It is mainly based on multi-dimensional data-layout (to enhance modularity regarding parallelization) that would aggregate as much payload blocks as possible. The objective being to increase the linearity of addresses accessed sequentially.

[5]The algorithm subdivides the result-matrix in T^2 (T: number of threads) sub-matrices (see Figure 5d). A thread i is in charge of the i^{th} column of sub-matrices. This thread accesses its corresponding sub-matrices sequentially starting from the one at height i.



(a) LLC cache misses: comparing 1D and 2D data layout

(b) Total number of cycles: comparing 1D and 2D data layout

(c) LLC cache misses: comparing 2D data layout optimizations

(d) Total number of cycles: comparing 2D data layout optimizations

Figure 3: Experimental comparison, according to LLC cache miss and CPU cycles, of a matrix-multiplication implementation using 4 different data layout: 1D array, simple 2D array, 2D array with blocks aligned with cache lines and 2D array with blocks aligned with cache-lines and transposed first operand and result matrix.

Meanwhile, data and instruction caches are not the only one to suffer from contention, nor the only one that may lead to an important time overhead. Indeed, given the rising complexity of OS-pagination systems[6] [6], more and more contention is applied to TLB. In this context, Figure 5 shows the impact of TLB-misses (Figure 5b) on the performance (Figure 5a) of a multithreaded implementation of the algorithm Figure 5d. Indeed, we can notice that the execution time is much more correlated with the experimental number of TLB misses than the number of L3 misses. The impact of TLB-based optimization may thus be more relevant in such a work-case than focusing on data caches.

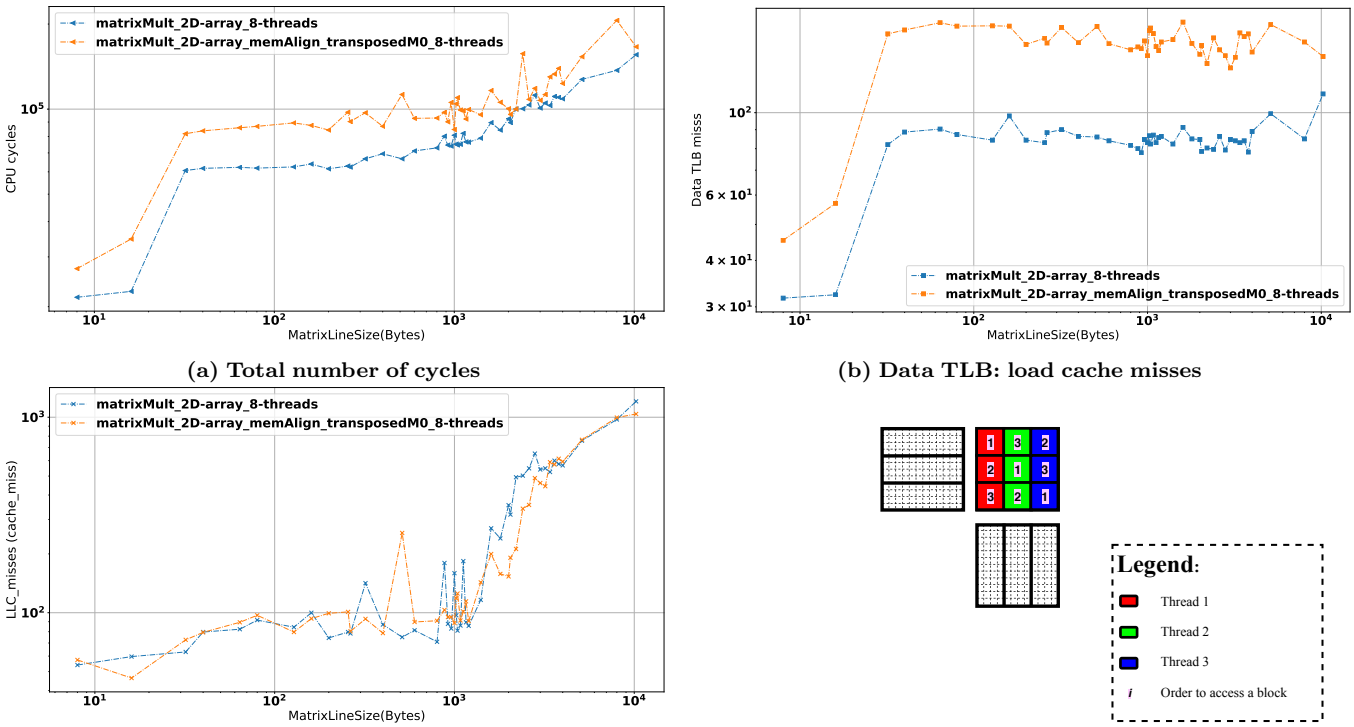
6 CONCLUSION

In this paper, we introduced a methodology to predict the number of data-cache misses triggered while accessing a matrix following the naive multiplication or convolution algorithms. We used the generated models to validate or reject some families of software optimization, regarding the software-parameters that they consider: memory alignment, memory clustering and data-layout projection. In the context of software-optimization decision, we showed how our generated models can be more accurate then the existing worst-case or roof-line models: they allow to predict *performance-trends* regarding the considered hardware parameters and the access pattern.

[6]We mainly refer to the rising number of address-accesses needed to transform virtual to physical addresses

REFERENCES

[1] Nasser Alsaedi et al. 2018. Applying Supervised Learning to the Static Prediction of Locality-Pattern Complexity in Scientific Code. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE.
[2] Mohamed-Walid Benabderrahmane et al. 2010. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*. Springer.
[3] Henri-Pierre Charles et al. 2014. deGoal a tool to embed dynamic code generators into applications. In *International Conference on Compiler Construction*. Springer.
[4] Wesley W Chu and Holger Opderbeck. 1972. The page fault frequency replacement algorithm. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*. ACM.
[5] Thomas H Cormen et al. 2009. *Introduction to algorithms*. MIT press.
[6] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient address translation for architectures with multiple page sizes. *ACM SIGOPS Operating Systems Review* (2017).
[7] Johan Cronsioe et al. 2013. BOAST: Bringing optimization through automatic source-to-source transformations. In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. IEEE.
[8] Stephane Eranian. 2006. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*.



(a) Total number of cycles

(b) Data TLB: load cache misses

(c) Last level of cache (L3)

(d) Data TLB: store cache misses

Figure 5: Experimental comparison of two implementations of a multithreaded version of matrix-multiplication[5]running 7 threads pinned on independent CPU cores and using 2 different data layouts: simple 2D array and 2D array with blocks aligned with cache-lines and transposed first operand and result matrix.

[9] Matteo Frigo and Steven G Johnson. [n.d.]. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*. IEEE.

[10] Millad Ghane et al. 2015. False sharing detection in OpenMP applications using OMPT API. In *International Workshop on OpenMP*. Springer.

[11] Daniel Häggander and Lars Lundberg. 1998. Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In *icpp*. IEEE.

[12] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* (2019).

[13] John McCarthy. 1978. History of LISP. *ACM Sigplan Notices* (1978).

[14] Richard ONeil et al. 1963. Convolution operators and $L(p, q)$ spaces. *Duke Mathematical Journal* (1963).

[15] Riyane SID-LAKHDAR and Pavel SAVIANKOU. 2017. On the Impact of Asynchronous I/O on the performance of the Cube re-mapper at High Performance Computing Scale. (2017).

[16] Daniele G Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM.

[17] S Tomov et al. 2009. Magma library. *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA* (2009).

[18] Samuel Williams et al. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[19] Yutao Zhong et al. 2007. Miss rate prediction across program inputs and cache configurations. *IEEE Trans. Comput.* 56, 3 (2007).