# Data-layout optimization based on Memory-Access-Pattern analysis for Source-Code Performance improvement

Riyane Sid Lakhdar, Henri-Pierre Charles, Maha Kooli

# Data-Layout Optimization based on Memory-Access-Pattern Analysis for Source-Code Performance Improvement

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

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

Maha Kooli
Univ Grenoble
Alpes, CEA, List, F-38000, France
maha.kooli@cea.fr

## Abstract

With the rising impact of the memory wall, selecting the adequate data-structure implementation for a given kernel has become a performance-critical issue. This paper presents a new methodology to solve the data-layout decision problem by adapting an input implementation to the host hardware-memory hierarchy. The proposed method automatically identifies, for a given input software, the most performing data-layout implementation for each selected variable by analyzing the memory-access pattern. The proposed method is designed to be embedded within a general-purpose compiler. Experiments on *PolybenchC* benchmark, *recursive-bilateral* filter and *jpeg-compression* kernels, show that our method accurately determines the optimized data structure implementation. These optimized implementations allow reaching an execution-time speed-up up to 48.9x and an L3-miss reduction up to 98.1x, on an *x86* processor implementing an *Intel Xeon* with three levels of data-caches using the *least recently used* cache-replacement policy.

***CCS Concepts.*** • **Theory of computation** → *Pattern matching*.

***Keywords.*** Code optimization, Source-to-source compilation, Data structure, Code adaptation to the hardware, Memory-access-pattern detection, Hardware-memory hierarchy

**ACM Reference Format:**
Riyane Sid Lakhdar, Henri-Pierre Charles, and Maha Kooli. 2020. Data-Layout Optimization based on Memory-Access-Pattern Analysis for Source-Code Performance Improvement. In *23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES '20), May 25–26, 2020, Sankt Goar, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3378678.3391874


*SCOPES '20, May 25–26, 2020, Sankt Goar, Germany*

ACM ISBN 978-1-4503-7131-5/20/05…$15.00
https://doi.org/10.1145/3378678.3391874

## 1 Introduction

In a computation kernel, the total idle-time where the processor is stalled, waiting for the end of a memory-fetch, is usually significant compared to the total execution time. Reducing the memory-fetch time by improving data and cache locality brings thus an important gain. However, finding efficient and cache-friendly data-layout placement for a given access-pattern [2] is a complex task, first due to the large size of the search space. For instance, for two-dimensional matrix of size $(N,N)$, four families of implementations can be referenced, namely the unidimensional, line, diagonal and column-major [19]. Each one of these families has $16N^2$ possible tiled versions where each submatrix (tile) may belong to each one of the previous families. Each one of these implementations has an order of $N!$ shuffled versions designed to suit accesses with various stride sizes.
In addition, the efficiency of a data-structure depends on its access pattern [19]. But the complexity for a human programmer to identify the pattern followed to access a data structure increases significantly with the loop-depth and number of conditional-branches within a software (SW) code.

For most standard data structures, multiple optimized implementations exist in the literature [19]. Proposing these optimization techniques with regards to a specific hardware (HW) and SW context, usually require an important engineering effort. In this paper, we are using these results to propose an optimized data-structure implementation within an unrelated input code.

This paper introduces *HARDSI*, a novel method that allows to automatically detect the access-pattern realized to a given data structure within an input code. Using this memory-pattern signature of the variable, the proposed method retrieves the optimized implementation for the considered data-structure from a custom data-base. This data-base maps an ideal implementation with each known pattern to access the data structure, with respect to the host HW-memory. Our method is designed as an optimization technique for general-purpose compilers. It is also taught to adapt input-codes to different present or future HW-memories. This paper also

introduces new *Domain-Specific Language* (DSL) based on *C/C++*. DSL, referred-to as the *HARDSI* language, allows the application of the proposed-method to *C/C++* source code. We develop it as a lightweight framework aiming to bypass the burden of implementing our method within complex *C/C++* compilers such as *gcc* [6] or *LLVM* [11]. For a given data structures $d$, it consists of replacing all the *C/C++* routines that implement $d$: definition, allocation, access and release, by custom-defined-primitives.

The main contributions of this paper are:

- A method for the detection and classification of patterns to access different data layouts.
- A framework to apply our optimization method based on pattern-detection for general-purpose *C/C++* code.
- An evaluation of the performance-improvement brought by our method.

We experiment our method on the *PolybenchC-4.2.1* benchmark [18], *recursive-bilateral filter* (RBF) [26] and *jpeg compression* (JPEG-C) [13] kernels running on an *x86 Intel Xeon* processor with three levels of data-caches [4] using the *least recently used* cache-replacement policy. Results show that the proposed method achieves an execution-time speed-up up to 48.9x and an L3 cache-miss reduction up to 98.09x. We also show that our method converges toward an optimized implementation while affected by up to 20% of noisy memory-accesses.

The rest of this paper is organized as follows. Section 2 discusses the state of the art. Section 3 details the different steps of *HARDSI* method. Section 5 discusses experiments on a computation kernel. Finally, section 6 concludes the paper.

## 2 Related Work

### 2.1 Dynamic-Memory Access Pattern

The memory-access pattern is the smallest set of consecutive accesses (read and write) to a given data structure that can be repeated in order to represent the total accesses to the data structure. Different existing works have evaluated the link between a memory-access pattern and the performance regarding modern HW-memory hierarchies (at least one level of fast memory between CPU and main memory) [2, 8, 15, 17, 19, 20]. All these approaches study manually the link of a memory-access pattern with the optimal data structure implementation. To the best of our knowledge, no automatic approach has been proposed.

The memory-access detection is intensively used by HW prefetchers [24]. However, the granularity of such method does not allow detecting pattern over a high-level data-structure. In [25], the authors present a memory access detection method that builds the *memory signature* of an application in order to detect malware-injection. Unlike the signatures that our method generates, the ones generated in [25] are not fully reproducible. Indeed, the considered framework uses non-transformed virtual addresses. It is thus subject to the variability of virtual addresses for two similar executions of the same kernel. This signature-variability prevents its use for SW optimization.

### 2.2 The Data-Layout Decision Problem

The compiler-driven SW optimization has for long consisted in determining an optimal set and order of instructions within an input source code [1, 5, 12, 16, 22]. Since the HW-memory hierarchies are getting complex, different studies rather focused on optimizing the data-placement across different levels of memory (e.g. RAM, caches or scratchpad memories) and at different scales (e.g. scalar variables, memory blocks or pages). Our approach for SW optimization is related to solving the Data-Layout Decision (DLD) problem. As for most solutions dealing with DLD at compiler scale, we assume that all possible loop-transformations and instruction-shuffling optimizations are already performed.

Two families of strategies have been proposed to tackle the DLD problem. Based on a previously observed memory-footprint, the ideal memory-placement at compile-time statically determined [14, 15, 21]. In [14], the authors introduce a general-purpose compiler approach that adapts the array allocation problem to graph coloring for register-allocation. The main issue with static approaches arises when the set of input data, used during the optimization (compilation), leads to a different behavior than the one at run-time. The optimal memory placement may then be computed based on irrelevant observations. In [17], the authors deal with this over-fitting issue by proposing to divide the considered memory (scratchpad) in clusters. The problem of populating each cluster is then reformulated as an integer-linear programming problem. This strategy scales well with programmable memories (such as scratchpads). Indeed, these types of memories require the programmer to decide which data to fetch in which memory. However, the overhead of forcing the selected data in non-programmable fast memories (such as caches) might significantly downgrade the performance. It might also create a performance-pitfall due to potential concurrences with data prefetchers.

In [2, 9], the authors propose dynamic approaches consisting to finding, at run-time, the proper placement of memory with regard to the previous memory-accesses of the current execution. In [9], dynamic loop-transformations are proposed to inject data-fetch instructions for scratchpad memories. In [2], a heuristic function is proposed to decide which data-copy to process after each fixed number of memory accesses.

The major limitation of these existing solutions is the lack of portability to new HW or SW platforms. Indeed, existing works focus explicitly either on a specific access type (regular [21] or irregular [14, 23] accesses), or on a specific HW-memory hierarchy (scratchpad memories [2] or multi-processor system on chips with some specific direct memory

access [8, 10]). To the best of our knowledge, the proposed approach in this paper is the first to port data-structure implementations, optimized for a specific HW or kernel, on a broad spectrum of applications and memory hierarchies.

## 3 Global Optimization Process

In this section, we introduce the proposed optimization approach that allows automatically converging toward the optimal data-layout implementation for each variable in a given code without altering the algorithm (*i.e.*, instruction flow). The objective of the generated *HARDSI* code is to select an optimized implementation for the used data structures that fits the followed access pattern. In this paper, we do not allow changing the memory access pattern even though the adaptation of the algorithm could bring an additional gain.

In Figure 1, we summarize the steps of our method. We also illustrate each step of the method by referring to a case-study of a matrix multiplication (Figure 3). The objective of the proposed optimization process is to find the ideal implementation of each of the three matrices given the respective memory-pattern followed to access each matrix.

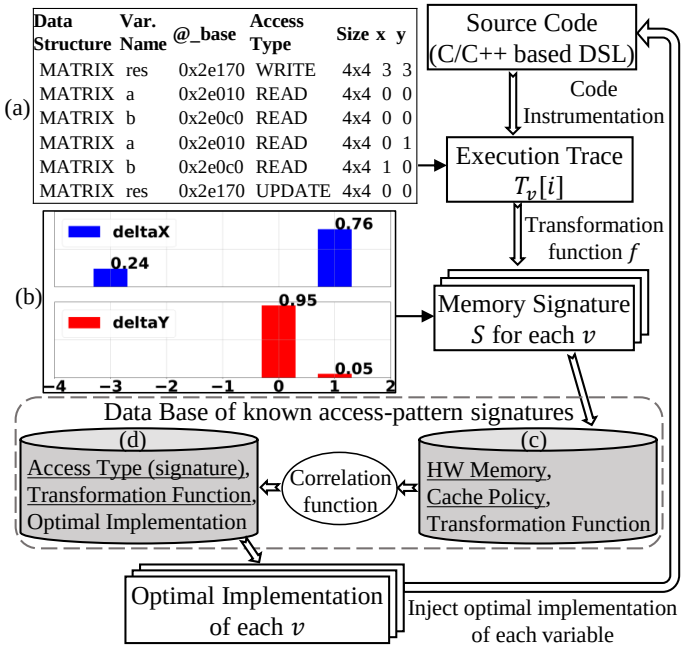


**Figure 1.** Steps of the proposed optimization-process.

### 3.1 Memory-Access Tracking

The first step of the proposed method is to observe the memory-addresses followed to access each considered variable. We propose to run the targeted execution of the input kernel. The memory-accesses of this execution are tracked by injecting a custom logging-function for each memory-access (read or write) to the considered variable. The resulting trace, presented in Figure 1.a, consists, for each variable $v$, in a set of virtual addresses $T_v[i]$, sorted according to their access rank $i$.

### 3.2 Generating a Memory-Signature

Once we traced the memory-access, the second step is to filter this trace through a *transformation function* $f$. The objective of the transformation is to remove the randomness introduced by the kernel's execution. Indeed, most general-purpose operating systems store data structures at different virtual addresses from one execution to an other. The proposed *transformation function* makes the result of this second step ($T_v^f[i]$) totally predictable (reproducible from one run to an other assuming that the same kernel's execution and input is observed).

In this paper, we used *x86 Intel Xeon* processor with three levels of data-caches. The corresponding transformation used to generate all the optimized code is a $\delta$ function: $T_v^\delta[i] = T_v[i] - T_v[i-1]$. We assume that $T_v^\delta[0]$ is undefined. This function is a lightweight computation that encompasses the performance-requirement of the memory hierarchy used and its cache-replacement policy (*least recently used*). It makes the absolute-distance between consecutively-accessed addresses the prominent parameter for classifying data-structures and their relative access-pattern. This parameter is known to be highly correlated with the data-reuse in the considered data-caches [19]. Adapting our method to different HW-memory corresponds to finding the adequate transformation function. This task is one of the perspectives of this study, and it is out of the scope of this paper.

Finally, we build the memory-signature $S$ of $v$ by generating the occurrence-histogram of each set $T_v$ (Figure 1.b and Figure 2.a). We normalize each histogram by dividing by the total number of occurrences. This makes the generated signature independent from the kernel's execution-inputs, and representing the absolute memory-behavior of the kernel.

### 3.3 Access Pattern Data Base

The relation linking a memory-signature $S$ with the corresponding optimal implementation is a key step for the proposed method. We thus built a data base as a survey of the most efficient existing implementations of $d$ on the considered memory hierarchies (represented by the *transformation function*) [1, 9, 13, 26]. The data base relative to each $d$ is integrated within the *HARDSI* framework. We embedded tools to allow adding or updating data-structure implementations corresponding to new or preexisting memory-access patterns.

### 3.4 Software Optimization

The generated memory signature $S$ for the variable $v$ of a given data structure $d$, identifies the specific memory-pattern followed to access $v$ in the considered kernel. Thus, we use it to classify all the known access-patterns of $d$. In the data-base

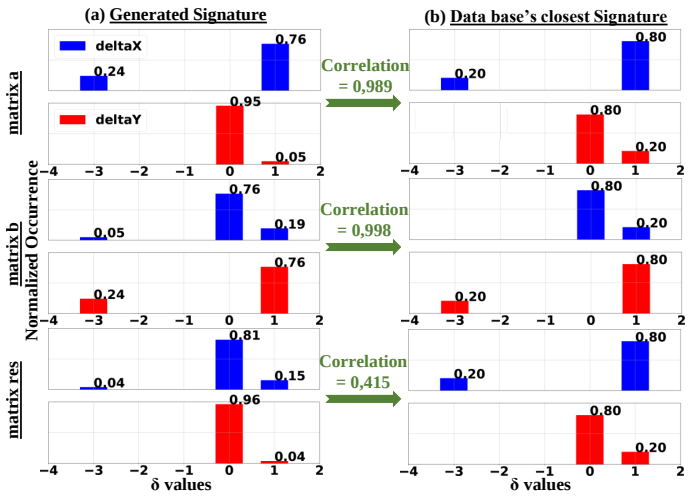


**Figure 2.** (a) Generated memory signatures of the 4x4 matrices $a$, $b$ and $res$ of matrix multiplication test case; (b) Respective closest signatures in the *HARDSI* data base. The memory accesses are observed with matrix indexes (abscissa and ordinate); transformed using the $\delta$ function; correlation is calculated using *Pearson*.

schemed in the relation presented in Figure 1.c, we link each known pattern of access to $d$ with a set of optimized implementations of $d$. The ideal implementations are also known to depend on the underlying HW-memory hierarchy. This dependence is identified by the used *transformation-function* and stored in the relation presented of Figure 1.d. An access pattern to a data structure $d$ is thus identifier of an optimal implementation of $d$ (along with the used *transformation function*). Finding the optimal implementation of $v$ is equivalent to finding the closest signature to $S$ in the data base relative to $d$, as shown in Figure 2. In order to compare the generated signature $S$ with each signature $S'$ in the data base, we use the *Pearson* coefficient [7] as a correlation function. This coefficient is defined as

$$\rho(S,S') = \frac{cov(S,S')}{\sigma_S \sigma_{S'}} = \frac{\frac{1}{N}\sum_{i=0}^{N-1}(S[i]-\mathbb{E}_S)(S'[i]-\mathbb{E}_{S'})}{\sigma_S \sigma_{S'}}$$

where $\mathbb{E}_S$, $\sigma_S$ and $N$ are the expectation, standard deviation and number of elements (including some elements with a null-occurrence) of the histogram.

The final step is to inject the optimal implementation of each variable in the source code of the application kernel.

## 4 Implementation Framework of *HARDSI*

We implemented the proposed method within a framework that permits to generate automatically the optimized source code of a given kernel. We implement first the considered kernel in the *HARDSI* DSL, as shown in the example in Listing 1. Then, we use our source-to-source compiler to generate the corresponding optimized *C/C++* code, as shown in the example in Listing 2. The proposed compiler has first computed the memory signatures of the variables as shown in Figure 2.a. These signatures are then compared to the data base signatures in order to select the most correlated one. The chosen implementation for a given variable is then the code associated within this closest signature.

```
void matrixMult() {
  MATRIX_DEFINE(int, a);
  MATRIX_DEFINE(int, b);
  MATRIX_DEFINE(int, res);


  int i,j,k, a0, b0;
  MATRIX_ALLOCATE(int, N0,
                       N1,
                       a);
  MATRIX_ALLOCATE(int, N2,
                       N0,
                       b);
  MATRIX_ALLOCATE(int, N2,
                       N1,
                       res);
  for (j=0; j<N1; j++)
   for (i=0; i<N2; i++)
    for (k=0; k<N0; k++) {
     a0=MATRIX_GET(a,k,j);
     b0=MATRIX_GET(b,i,k);
     MATRIX_ADD(res,i,j,a0*b0);
    }
  MATRIX_FREE(a,N0,N1, int);


  MATRIX_FREE(b,N2,N0, int);


  MATRIX_FREE(res,N2,N1, int);


}
```

**Listing (1)** Input code implemented using our custom *HARDSI* DSL

```
void matrixMult(){
  int **a;
  int **b;
  int **res;
  int s =sizeof(int);
  int sp=sizeof(int*);
  int i,j,k, a0, b0;
  a = (int**)malloc(N1*sp);
  for (i=0; i<N1; i++)
    a[i]=(int*)malloc(N0*s);
  b = (int**)malloc(N2*sp);
  for (i=0; i<N2; i++)
    b[i]=(int*)malloc(N0*s);
  res = (int**)malloc(N1*sp);
  for (i=0; i<N1; i++)
    res[i]=(int*)malloc(N2*s)
  for (j=0; j<N1; j++)
    for (i=0; i<N2; i++)
      for (k=0; k<N0; k++){
        a0=a[j][k];
        b0=b[i][k];
        res[j][i]+=a0*b0;
      }
  for (i=0; i<N0; i++)
    free(a[i]);
  free(a);
  for (i=0; i<N2; i++)
    free(b[i]);
  free(b);
  for (i=0; i<N2; i++)
   free(res[i]);
  free(res);
}
```

**Listing (2)** Corresponding *C/C++* code generated by our source-to-source compiler

**Figure 3.** Matrix Multiplication Test Case.

## 5 Results and Discussion

### 5.1 Experimental Setup

In this paper, we used an x86 hardware architecture implementing an *Intel Xeon E3-1270 v4* processor with an 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 system is used based on the *Linux* (3.16.0-4) kernel. The *g++* (4.9.2) compiler (with the -03 optimization option) is used to compile the considered computation-kernels. This includes the native *C/C++* code and the one generated by our *HARDSI* source-code generator. The *Perfmon2* library [3] is used to access the performance-management unit of the processor in order to measure different cache misses and CPU cycles. The *cpupower* toolkit is used to disable the *automatic CPU-frequency scaling* of our processor.

All the presented performance results are obtained following the same procedural method. Each considered point is assessed (experimental run) 10 times, and the presented results

are the average of these runs. Given its relatively small value (smaller than 1% for all the experiments) no variation is presented. The performance gain that we show in this section are obtained without re-ordering the instructions of the original algorithm. We make sure to flush all the data-caches between two consecutive experiments using a *CFLUSH* assembly instruction.

The *HARDSI* code is transformed into native *C/C++* (using our source-to-source compiler) once and assessed using different inputs. No recompilation is made between two executions. We use input matrices with the following sizes as a learning input: {(4,4)(4,4)}, {(4,7)(13,4)}, {(128,128)(128,128)}. The statistical properties of the *HARDSI* method spare the need to use larger inputs during this optimization phase because the same behavior is observed with larger inputs. The list of matrix implementations embedded within our *HARDSI* data-base is shown in Figure 4.

All the results presented in this section are obtained using float matrices. Similar results might be observed using other basic types of data such as integers or doubles.

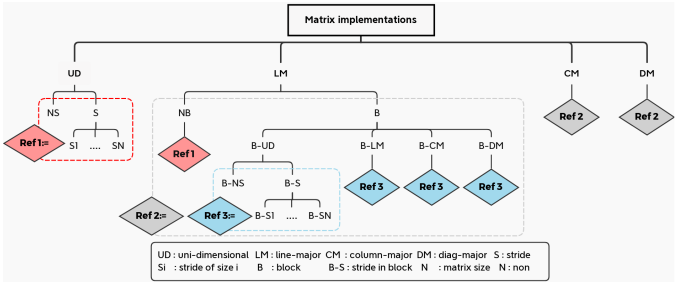


**Figure 4.** Matrix implementations in the *HARDSI* data-base

### 5.2 Experimented Benchmark

In order to evaluate the ability of our method to select an optimized data-structure-implementation for different access types, we use different benchmarks *PolybenchC-4.2.1* [18] benchmark suite. It is designed for polyhedral-compilation evaluation, and encompasses kernels with a large specter of matrix-access-types ranging from block up to stencil-walks. For each one of these access-types, we select one kernel. Additionally, in order to evaluate the interest of our method on a combination of basic access-patterns, we use kernels implementing a *jpeg-compression* (JPEG-C) [13] and *recursive-bilateral-filter* (RBF) [26]. The list of memory-access patterns evaluated through all these kernels are presented in Table 1. We consider five different input-matrix sizes: 50, 100, 600, 1000, 2048. This allows representing different cases where the data fit one of the DL1, L2 or L3 caches (i.e., fitting one cache line or the whole cache).

### 5.3 Results and Discussion

The metric used in the experimental evaluation of this section is the speed up between a baseline (native *C/C++*) and the corresponding *HARDSI* implementation. This speed-up is defined by the performance-ratio of the baseline to the *HARDSI* implementation.

| | Line-maj | Column-maj | Stencil | Line-maj block-line | Line-maj block-diag |
|---|---|---|---|---|---|
| covariance | ✓ | | | | |
| correlation | ✓ | | | | |
| adi | ✓ | | ✓ | | |
| gesummv | ✓ | ✓ | | | |
| floyd-warshall | | ✓ | | | |
| lu | ✓ | ✓ | ✓ | | |
| JPEG-C | | | | ✓ | ✓ |
| RBF | ✓ | ✓ | | ✓ | |

**Table 1.** Memory-access pattern followed by considered kernels.

Table 2 shows that our *HARDSI* method is able to select the most efficient (in terms of CPU cycles) data-structure implementation for each kernel and with respect to each input size. Indeed, the first line of the table (speed up of the implementation automatically generated by our *HARDSI* method) is always equal to the second line (speed up of the kernel using the best known implementation of the data structure in the considered context). The best speed up of a kernel is the highest speed up reached while evaluating the kernel using each known implementation of the matrix data-structure in the data-base.

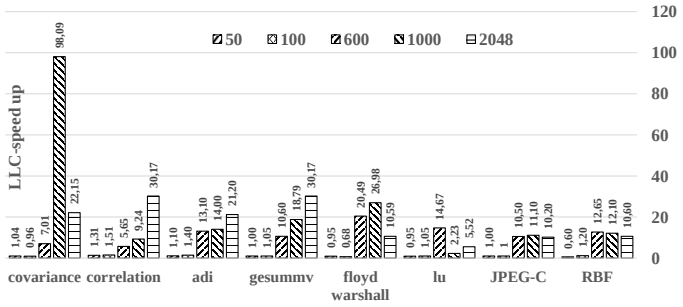


**Figure 5.** Performance speed up, in terms of LLC cache-misses, between a *HARDSI* and baseline implementation.

In Table 2, we show that our *HARDSI* method allows bring an execution-time speed-up ranging from 1 to 48.9$x$. Our method is able to keep, in worst case, the default implementation of a matrix (column-major with no stride nor sub-block) when no implementation fits better with the realized access pattern. This case is primarily observed on the stencil kernel *adi*. Thanks to our method, we reache up a 2.3$X$ speed up using a custom matrix implementation. This implementation duplicates column-data within each line in a configuration that allows using single-instruction-multiple-data operations to update each cell. To the best of our knowledge, no optimized matrix-implementation has been proposed (for the considered memory-hierarchy and stride-size). The complexity of this case comes from the simultaneous realization

| Matrix size | 50 | 100 | 600 | 1000 | 2048 | 50 | 100 | 600 | 1000 | 2048 | 50 | 100 | 600 | 1000 | 2048 | 50 | 100 | 600 | 1000 | 2048 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | covariance | | | | | correlation | | | | | adi | | | | | gesummv | | | | |
| Speed up *HARDSI* | 1.0 | 1.1 | 3.2 | 8.3 | 12.2 | 1.0 | 1.0 | 3.1 | 7.7 | 12.1 | 1.4 | 1.6 | 1.6 | 1.4 | 2.3 | 1.4 | 1.3 | 7.1 | 12.2 | 12.5 |
| Speed up Best SoA | = | = | = | = | = | = | = | = | = | = | 1.0 | 1.0 | 1.0 | 1.0 | 1.3 | = | = | = | = | = |
| Reference Best SoA | [1] | | | | | [9] | | | | | [9] | | | | | [1] | | | | |
| | floyd-warshall | | | | | lu | | | | | JPEG-C | | | | | RBF | | | | |
| Speed up *HARDSI* | 3.9 | 5.4 | 28.9 | 48.9 | 29.4 | 1.0 | 1.0 | 1.5 | 4.0 | 6.3 | 1.0 | 3.0 | 7.9 | 10.4 | 12 | 2.1 | 10.9 | 34.5 | 42.5 | 45.9 |
| Speed up Best SoA | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = | = |
| Reference Best SoA | [9] | | | | | [1] | | | | | [13] | | | | | [26] | | | | |

**Table 2.** Performance speed up, in terms of CPU-cycles, between a HARDSI and baseline implementation. The value "=" (respectively "<") means that the highest speed up reached using a state of the art (SoA) implementation is equal (respectively smaller) to the one observed using the *HARDSI* implementation.

of two antagonistic access-patterns (line and block-column major). Thus even though one pattern fits properly the cache-behavior, the second one does not.

For matrix-sizes smaller than the total L3 size (1000 and 2048), Table 2 shows that the observed execution-time improvement for all the kernels other than *adi* ranges from $4x$ up to $48.9x$. We show in Figure 5 that this time improvement is mainly explained by the reduction of L3-cache-misses (from 2.23 up to $98.09x$). Thus, our method allows to properly identify the memory-access pattern to each data structure. This proper identification allows linking each detected pattern with an efficient implementation of the corresponding data structure.

For matrices smaller than the size of the used L3 cache (50, 100 and 600), Figure 5 shows that a cache-miss speed-up as small as $0.20x$ is observed. The *HARDSI* implementation has thus led to an increased number of L3 cache misses by up to $5x$ while an execution-time improvement is still observed (CPU cycles speed up $\geq 1$). Our interpretation is that the results observed at the L3 level are not relevant. For these matrix sizes, the L3 cache is not much solicited due to the sufficient space in DL1 and L2 to store all the considered data. Meanwhile, the DL1 and L2 are intensively solicited. Thus, the speed up observed on DL1 and L2 (not represented in this paper) is the one that explains the execution-time speed up. This also explains the relatively modest speed-up observed for these small matrices (50, 100 and 600) compared to the one observed for bigger matrices (1000 and 2048). Given that an L3-miss latency is about 3 to 10 times bigger than a DL1 or L2 latency, then reducing the number of DL1 or L2 misses brings less time improvement than reducing the number of L3 misses.

## 6 Conclusion

In this paper, we present a novel process designed to build the memory signature of a variable within a computation kernel. We introduce a custom framework in order to automatically generate the optimal implementation of a data-structure's variable based on its memory-signature and with respect to the specificity of the host hardware-memory.

## References


[1] Mohamed Benabderrahmane et al. 2010. The polyhedral model is more widely applicable than you think. In *CC*.

[2] Doosan Cho et al. 2008. Compiler driven data layout optimization for regular/irregular array access patterns. ACM.

[3] Stephane Eranian. 2006. Perfmon2: a flexible performance monitoring interface for Linux. In *OLS*.

[4] David Friedman et al. 2004. Three-dimensional memory cache system. US Patent 6,711,043.

[5] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *ICASSP*.

[6] Arthur Griffith. 2002. *GCC: the complete reference*. McGraw-Hill.

[7] Larry L Havlicek et al. 1976. Robustness of the Pearson correlation against violations of assumptions. *Perceptual and Motor Skills* (1976).

[8] Ilya Issenin et al. 2006. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC*.

[9] Mahmut Kandemir et al. 2001. Dynamic management of scratch-pad memory space. In *DAC*. IEEE.

[10] Mahmut Kandemir et al. 2005. Memory systems and compiler support for mpsoc architectures. In *MpSoC*. Elsevier.

[11] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conf*.

[12] Rahman Lavaee et al. 2019. Codestitcher: inter-procedural basic block layout. In *CC*. ACM.

[13] Alain M Leger et al. 1991. JPEG still picture compression algorithm. *Optical Engineering* (1991).

[14] Lian Li et al. 2005. Memory coloring: A compiler approach for scratchpad memory. In *PACT*.

[15] Samy Meftali et al. 2001. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *ISSS*. ACM.

[16] Charith Mendis et al. 2019. Revec: program rejuvenation through revectorization. In *CC*.

[17] Abdolmajid Namaki Shoushtari. 2018. *Software Assists to On-chip Memory Hierarchy of Manycore Embedded Systems*. Ph.D. Dissertation. UC Irvine.

[18] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* (2012).

[19] Sid Lakhdar Riyane et al. 2019. Toward Modeling of Cache-Miss Ratio for Dense-Data-Access-Based Optimization. In *RSP*. ACM.

[20] Manuel Serrano et al. 2019. Property caches revisited. In *CC*.

[21] Aviral Shrivastava et al. 2016. Automatic management of software programmable memories in many-core architectures. *IET CDT* (2016).

[22] Daniele Spampinato et al. 2016. A basic linear algebra compiler for structured matrices. In *CGO*.

[23] Manish Verma et al. 2003. Data partitioning for maximal scratchpad usage. In *ASPDAC*. ACM.

[24] Christopher B Wilkerson et al. 2019. Instruction and logic for software hints to improve hardware prefetcher effectiveness. US Patent 10,229,060.

[25] Zhixing Xu et al. 2017. Malware detection using machine learning based analysis of virtual memory access patterns. In *DATE*.

[26] Qingxiong Yang. 2012. Recursive bilateral filtering. In *ECCV*.