Methodology for code-optimization of memory data layouts by adaptation to high-performance-system architectures with complex memory hierarchies

Riyane Yacine Sid Lakhdar

▶ **To cite this version:**

Riyane Yacine Sid Lakhdar. Methodology for code-optimization of memory data layouts by adaptation to high-performance-system architectures with complex memory hierarchies. Performance [cs.PF]. Université Grenoble Alpes [2020-..], 2020. English. NNT : 2020GRALM058 . tel-03166041

HAL Id: tel-03166041
https://theses.hal.science/tel-03166041v1

Submitted on 11 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UGA Université Grenoble Alpes

THESIS

for the degree of

DOCTOR of GRENOBLE ALPES UNIVERSITY

Specialty: **MATHEMATICS AND COMPUTER SCIENCE**

Ministerial Order: May 25, 2016

Presented by

Riyane Yacine SID LAKHDAR

Thesis supervised by Henri-Pierre CHARLES and Maha KOOLI

Prepared at the laboratories
Laboratoire d'Intégration des Systèmes et des Technologies / CEA LIST Grenoble
in **the Doctoral School MSTII (Mathématiques, Sciences et Technologies de l'Information et de l'Informatique)**

Methodology for Code-Optimization of Memory Data-Layouts for High-Performance-System Architectures with Complex Memory Hierarchies

Defended on **11/12/2020**

Jury:

M. Denis BARTHOU Professor at INRIA-INP, Bordeaux	Reviewer
M. Matthieu Moy HDR, Associate Professor at UCB Lyon1- Ens - LIP	Reviewer
Mme. Vania MARANGOZOVA-MARTIN HDR, Associate Professor at University Grenoble Alpes - LIG	Examiner
M. Frédéric PÉTROT Professor at University Grenoble Alpes – TIMA	President
M. Lionel LACASSAGNE Professor at Sorbonne University - LIP6	Examiner
M. Henri-Pierre CHARLES HDR, Research director at CEA/LIST Grenoble	PhD Director
Mme. Maha KOOLI Research engineer at CEA/LIST Grenoble	PhD Supervisor

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. The complexity of solving efficiently this Data-Layout-Decision (DLD) problem is dramatically increased by the concurrence of complex, heterogeneous and application-specific hardware memories. Slightly modifying an optimized application or porting it to a new hardware architecture requires an important time and engineering effort. It also requires a deep knowledge of the host hardware platform.

In this thesis, we plot a first step toward automatic software-adaptation to hardware. We present an iterative data-mining-related software-optimization approach based on the detection and the exploration of the most influential parameters linked to the hardware, operating system and software. We also propose a custom data-cache-miss modeling algorithm designed to be used as fully-parameterized performance evaluation. The proposed approach is designed to be embedded within a general-purpose compiler.

In order to explore the parameters related to the data-layout implementation, we propose HARDSI, a custom patented method to solve the DLD problem. We also propose to apply our method using a custom domain-specific language and computation framework. The HARDSI method allows to choose, from a custom base of knowledge, an optimized data-layout implementation with regards to the memory-pattern followed to access the considered data-structure. The generated solutions are also specifically adapted to the properties of the host hardware-memory.

Meanwhile, we consider the singular resolution of the DLD problem on memories that are explicitly addressed by the programmer (such as embedded scratchpad memories or GPUs). The problem that we address is to find an optimized memory-placement in order to maximize the amount of frequently-accessed data to be stored within this fast yet narrow memory. In this context, we propose DDLGS, a custom patented method designed to generate a dynamic data-layout with regards to the followed memory-access pattern. The generated implementations encompass the specific load and store routines as well as the granularity attributed to each data transferred. These implementations are also able to adapt, at run time, to the input of the considered source-code.

Aiming to evaluate our implementations on different hardware environments, we have considered two different processor and memory architectures: (i) An *x86* processor implementing an *Intel Xeon* with three levels of data-caches utilizing the least recently used replacement policy and a (ii) Massively Parallel Processor Array implementing a *Kalray Coolidge-80-30* with a $16KBytes$ on-chip scratchpad memory. Experiments on linear algebra, artificial intelligence and image processing benchmarks show that our method accurately determines an optimized data-structure implementation. These implementations allow reaching an execution-time speed-up up to $48.9x$ on the *Xeon* processor and $54.2x$ on the *Coolidge* processor.

Resumé

La sélection d'une implémentation adéquate de structure de données pour un noyau de calcul donné est un problème critique pour les performances logicielles. La complexité de la résolution efficace de ce problème est exacerbée par la concurrence de mémoires matérielles complexes, hétérogènes et dédiées à une application specifique. Modifier légèrement une application optimisée ou la porter sur une nouvelle architecture matérielle nécessite un temps et un effort d'ingénierie considérable. Cela nécessite également une connaissance approfondie de la plateforme matérielle hôte.

Au cours de cette thèse, nous franchissons une première étape vers l'optimisation par l'adaptation automatique du logiciel au matériel. Nous présentons une approche itérative d'optimisation basée sur la détection et l'exploration des paramètres les plus influents liés au matériel, au système d'exploitation et au logiciel. La méthode proposée est conçue pour être intégrée dans un compilateur à usage général. Dans ce contexte, nous proposons un algorithme de génération de modèles (entièrement paramétrées) de mémoires caches. Les modèles de performance générés sont conçus pour être utilisé dans le cadre d'évaluations de performances et d'optimisation.

Afin d'explorer les paramètres liés à aux structures de données, nous proposons HARDSI, une méthode brevetée permettant la résolution du problème de l'agencement des données pour logiciel donné. Dans le but d'appliquer notre méthode, nous proposons également un langage dédié (basé sur le langage C/C++) ainsi que son environnement logiciel de compilation et d'exécution. La méthode HARDSI permet de choisir, à partir d'une base de connaissances spécialisée, une implémentation optimisée de l'agencement des données en fonction de la géométrie d'accès à la structure de données. Les solutions générées sont également spécifiquement adaptées aux caractéristiques matérielles de la mémoire hôte considérée.

De même, nous considérons la résolution du problème de l'agencement des données sur les mémoires singulières qui sont explicitement adressés par le programmeur (tel que les mémoires de type "scratchpad" ou GPU). Le problème que nous abordons est de trouver un emplacement mémoire optimisé afin de maximiser la quantité de données fréquemment accédées et à stocker dans ce type de mémoires rapides bien qu'étroites. Dans ce contexte, nous proposons DDLGS, une méthode brevetée conçue pour générer une implémentation dynamique des données sur mémoires scratchpad. Ces implémentations sont conçus par DDLGS en considérant le schéma d'accès à la mémoire spécifiquement suivi par le code a optimiser.

Dans le but d'évaluer nos implémentations sur différents environnements matériels, nous considérons deux processeurs et mémoires différents: (i) un processeur x86 implémentant un *Intel Xeon* à trois niveaux de caches de données et (ii) un processeur massivement parallel implémentant un *Kalray Coolidge-80-30* à mémoire scratchpad sur puce de 16K octets. Les expériences menées sur des noyeaux d'algèbre linéaire, d'intelligence artificielle et de traitement d'images montrent que notre méthode détermine avec précision une implémentation optimisée des structures de données. Ces implémentations permettent d'atteindre une accélération du temps d'exécution jusqu'à 48,9x sur le processeur *Xeon* et 54,2x sur le *Coolidge*.

Acknowledgments

The work defended in this thesis have been prepared for much longer than I may call. Consequently and for their tremendous work in preparing me for this thesis, I would like to address my deepest and most sincere gratitude to my parents, Amina and Rachid. The countless weekends and holidays that you have sacrificed for my education are probably the only way for a trouble-maker like me to end up studying. I would like to thank my brother Wissam for his precious help and understanding. Thanks kho for setting the bar so high that my PhD and few papers look now like a joke. I would like to say my love to my grandparents papy, makhokha (el lahe yerhamkoum wi wessa3 3likoum) and mamy. Your endless love and prayers have always been my main asset. Your kindness, unique mind and simplicity is the only way I like to see the world. No family has ever been as patient, loving and tolerant to hard times; and for that I will always be grateful.

Then, this thesis has been imagined and hardly battled by my supervisor, Henri-Pierre Charles. It has been an honor to work with you and to witness your deep understanding and culture of computers. I would like to express my sincere and tender gratitude to Maha Kooli who accepted to join the supervision of my thesis at midterm. I could hardly show you how thankful I am for your time, precious advises and meticulous reviews of my work. You and your wonderful family has always saved us a seat at your table.

In order to assess and validate my work, I had the honor to defend my work in front of an extremely competent and recognized jury. I am very honored and thankful to each member of my jury for the time and consideration that was given to my work. I am deeply convinced that this work has highly benefited from the attentive reviews and thoughtful advice from my jury.

Finally, this thesis has offered me a unique chance to meet great minds and wonderful people. It was a blessing and an honor to see Vincent Olive, Diego Puschini and Didier Lattard successively lead the department of laboratory that I belonged to. It was a also blessing to share so many discussions, meals, coffees and times with so many interesting and joyful people. You guys are a great team. Keep it up.

Last but certainly not least, I would like to offer my deepest and most sincere gratitude to my uncle Dr. Halim LEHTIHET for his invaluable help and high-standard advises. His support through the years and his constant push to strive for excellency are the main reasons why I am currently allowed to defend this thesis. He also gave me one of the most valuable thing in life: a passion.

A special thanks to the unique mind behind every formula in my thesis, Dr. Roxana DIA. Your kind support and precious smile will forever be a warm memory.

Contents

List of Abbreviations 1
List of Acronyms 2
Mathematical Notations for Cache-Miss Modeling 3
Mathematical Notations for DLD Resolution 3

1 Introduction **5**
1.1 Historical Overview 5
1.2 Context and Objectives 6
1.3 Global Approach 7
1.4 Requirements and Issues to Overcome 9
1.5 Contributions 9
1.6 Experimental Setup 10
1.7 Thesis Structure 11

I State of the Art and Scientific Methodology **13**

2 Scientific Methodology: Performance Exploration **15**
2.1 Simulation and Emulation Tools 16
2.1.1 Trace Injection Simulators 16
2.1.2 Cache, RAM and Register Simulators 17
2.1.3 Instruction-Interpretation Simulators 18
2.1.4 Trade-off Between Accuracy and Execution time 19
2.1.5 Conclusion 19
2.2 Hardware and Software Performance Modeling: The *Roofline* model 20
2.2.1 Hardware Boundaries (Roof) 21
2.2.2 Fitting the Model to the Memory/Computational Optimizations (Ceiling) 21
2.2.3 Interest of the *Roofline* Model 23
2.2.4 Conclusion: Limitation of the *Roofline* Model for our Approach 23
2.3 Performance Measurement 23
2.3.1 Hardware Performance Counters 24
2.3.2 Performance Measurement Libraries 26
2.3.3 Conclusion: Choosing a Hardware-Software Performance Tracker 29
2.4 Conclusion: Adopted method and implementation choices 31

3 State of the Art: Software Optimization **33**
3.1 Overview in Software Optimization 33
3.2 Source-Code Adaptation to Hardware 34
3.2.1 Methodologies for Code Adaptation 34

3.2.2 The *LGen* Code Generator for Basic Linear Algebra Computation . . . 35
3.3 Data-Layout-Based Software Optimization . . . 37
3.3.1 Background: Data-Cache-Miss and Performance . . . 37
3.3.2 Data Layout: Definition and terminology . . . 38
3.3.3 Dynamic-Memory Access Pattern . . . 38
3.3.4 The Data-Layout Decision Problem . . . 40
3.4 Scratchpad memories . . . 42
3.4.1 Memory Overview . . . 42
3.4.2 High-Performance-Computing-Code Optimization . . . 43
3.4.3 Embedded-Code Optimization . . . 43

II Code Optimization by Adaptation to the Hardware Memories 45

4 HARDSI: Custom Method to Solve the Data-Layout Decision problem 47
4.1 HARDSI Overview . . . 48
4.1.1 Objectives . . . 48
4.1.2 The HARDSI DSL . . . 48
4.2 Global Optimization Process . . . 50
4.2.1 Memory-Access Tracking . . . 51
4.2.2 Generating a Memory-Signature . . . 51
4.2.3 Access-Pattern Data-Base . . . 52
4.2.4 Software Optimization . . . 54
4.3 HARDSI Framework Implementation . . . 55
4.3.1 Accelerating the HARDSI Process . . . 55
4.3.2 Conventional Three Cache-Levels Architectures . . . 57
4.3.3 Pluri-architectural Software Optimization . . . 58
4.4 Contemplated Future Implementations . . . 60
4.4.1 Multicore architecture . . . 60
4.4.2 In-memory Computing . . . 60
4.4.3 Detecting Malicious-Code Injection . . . 62

5 Dynamic Data-Layout Implementation for Programmable Memories 63
5.1 Extending the HARDSI Framework to Scratchpad Memories . . . 64
5.1.1 Background and Process Overview . . . 64
5.1.2 HARDSI Transformation-Function for Scratchpad Memories . 65
5.1.3 Implementation and Multiple-Memories Issue . . . 66
5.2 DDLGS: Generating Matrix Data-Layout Dedicated to Scratchpad . 67
5.2.1 Custom Generation Process . . . 67
5.2.2 Determining the Weight of Each Matrix Cell . . . 69

5.2.3 Generalizing the Resulting Implementation to New Input Data 70
5.2.4 Potential Improvements 72

III Experimental Validation and Discussion 73

6 *HARDSI* Experimental Evaluation 75
6.1 Experimental Setup: HARDSI Code Compilation 75
6.2 Matrix Multiplication Kernel 76
6.3 Experimented Benchmark 77
6.4 Evaluation on a Three Data-cache Levels 78
6.4.1 Experimental Results Overview 78
6.4.2 Impact of HARDSI on the Different Cache-Levels 79
6.5 Enhancing the HARDSI Method to scratchpad memories 81
6.5.1 Regular Data Caches VS HARDSI Scratchpad implementation 81
6.5.2 Baseline scratchpad VS HARDSI Scratchpad implementation 82
6.6 Impact of Noise on the HARDSI Method 84

7 Conclusion and Future Work Directions 87
7.1 Summary and Conclusion 87
7.2 Perspectives and Future Works 88
7.2.1 Short Term Perspectives 88
7.2.2 Long Term Perspectives 89

Publications 91

Bibliography 95

List of Figures

1.1 Custom evaluation of performance and code-complexity for a simple matrix multiplication application (100*100 integers) using different software implementations. The potential properties of each implementation are (P) Portable to new hardware/software (H) uses hardware-specific instructions (R) requires run-time code-refactoring 6
1.2 Global view of the proposed software-optimization process. 7

2.1 Trade-off between simulation-time and hardware-model accuracy (extract from [16]). 19
2.2 Roofline Model for (a) AMD Opteron X2 and (b) Opteron X2 vs. Opteron X4 (extract from [112]). 20
2.3 Refining the *Roofline* model depending on the used (a) memory optimization and (b) compute optimization. Data corresponding to an AMD Opteron X2 and (c) Opteron X2 processor (extract from [112]). 22
2.4 Layout of the IA32-PERFEVTSEL Model-Specific Register (MSR) (extract from [23]). 24
2.5 Code snippet to access an MSR looking for a Core (or central) processing unit (CPU) cycle counts on an (2.1) Intel and (2.2) IBM processor 27
2.6 Scalasca analysis report explorer presentation of Sweep3D execution performance with 294,912 MPI processes on the IBM Blue Gene/P platform. Interface of the *Cube* tool (extract from [114]). 28
2.7 Performance benchmark of two implementations of a basic matrix multiplication algorithm [57]. The performance tracker used is (a) *Score-P* versus (b) our custom MSR-based code (inspired from Listing 2.1). The dots are the average value of ten consecutive measurements. The shaded curves denote the distance between the minimal and maximal values of the ten measurement. The experiments follow the experimental setup described in section 1.6. 30

3.1 Cache-fetch behavior while accessing a data at an address a. $a[L]$ represents the rest of the euclidean division of a by L. 37
3.2 Set of $O(N!)$ potential implementations for a two dimensional matrix. 39
3.3 Example of four memory-access patterns to a matrix data structure. 39
3.4 Hardware view of a memory hierarchy including an on-chip scratchpad memory. 42

4.1 Matrix multiplication test case. 49
4.2 Steps of the proposed optimization-process. 50

4.3 (a) Generated memory signatures of the 4x4 matrices a, b and res in the matrix-multiplication test-case; (b) Respective closest signatures in the HARDSI data base. 53
4.4 Relational data base of knowledge: Stores the required relations to retrieve the best known implementation of a given data structure knowing its memory signature. The key attributes (line identifiers) are underlined. 54
4.5 Memory-access signatures of a $(10*10)$ matrix accessed following four different patterns. The tilled patterns ((a) and (b)) have tiles of size $(2*2)$. 58
4.6 Optimized matrix implementation relative to each part of a JPEG-compression algorithm 59
4.7 Overview of a tiled version of C-SRAM memory architecture (extract from [53]). 60

5.1 Application view of a software-process (UNIX) hosted on memory-hierarchy including a scratchpad memory. 64
5.2 Notations used by the transformation function of the *HARDSI* method for scratchpad usage. 66
5.3 Illustration of the four steps of the generation of a scratchpad-dedicated data-layout (with a fixed dimension). 68
5.4 Interpolation used to extend the dimension of an optimized matrix data-layout generated by the *HARDSI* method for scratchpad usage. 70

6.1 Experimental comparison, based on the number of L3 (a), CPU cycles (b) and Translation Lookaside Buffer (TLB) (c, d) cache miss of a matrix-multiplication implemented in both *C/C++* and our custom HARDSI Domain-Specific Language (DSL). 76
6.2 Performance speed up, in terms of LLC cache-misses (load and store), between a HARDSI and baseline implementation. 79
6.3 Performance speed up, in terms of data L1 cache-misses (load), between a HARDSI and baseline implementation (logarithmic scale). . 80
6.4 Performance speed up, in terms of data L1 cache-misses (store), between a HARDSI and baseline implementation (logarithmic scale). . 80
6.5 Performance speed up, in terms of CPU cycles between a HARDSI and a *Kalray* implementation designed for scratchpad usage. 83
6.6 Percentage of variables (vertical axis) in the *gesummv* kernel that was positively identified by the HARDSI method for a given percentage of random memory accesses (horizontal axis) introduced at random moment of the kernel's execution. This figures is obtained on both the *Intel Xeon* and the *Kalray* processors. 84

7.1 Global view of the proposed software-optimization process and the different contributions. 87

List of Tables

3.1 Classification of existing code-optimization solution according to their leverage-point on source code 34

3.2 Example of $C/C{+}{+}$ routines defining a uni-dimensional implementation of a 2D matrix of size N*N 38

4.1 Example of an execution-trace (partial) of three matrix-variables a, b and res for a $(4 * 4)$ matrix multiplication test case. 51

6.1 Memory-access pattern followed by considered kernels. 78

6.2 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. 79

6.3 Performance speed up, in terms of CPU-cycles, between a scratchpad implementation and the best implementation (optimized for L1 data cache) known for each kernel. The scratchpad implementation is either the one automatically selected by the HARDSI method (first line) or the best found by testing all the matrix implementations in the HARDSI data base (second line). The value "=" means that the highest speed up reached using a HARDSI-selected implementation is equal to the one observed using the best tested implementation. . 81

Glossary

List of Abbreviations

C-SRAM Computational SRAM is a an in-memory computing architecture. It consists in a static RAM memory that embeds computing abilities. It is designed to reduce the time and energy cost of the transfers between the main memory hierarchies and the computational blocks (CPUs and accelerators)..

DMA Direct Memory Access, referring to a device that can transfer data in and out of the memory (main memory or on-chip scratch memory) without involving the processor. The processor is only interrupted at the completion of the transfer.

LLC Last Level Cache refers to the highest-level of data cache within the hardware-memory hierarchy. Unlike the first-level of cache, the LLC is usually a single hardware block that is shared by all the processing units. In the example of the *x86 Intel Xeon E3-1270* processor, the LLC corresponds to the L3 data cache..

LRU Least-Recently Used cache-replacement policy consists in evicting, whenever a new cache-line is added, the least-recently used line..

MPPA Massively Parallel Processor Array refers to a family of on-ship processors with a large (over a hundred or even a thousand) number of CPUs. These processors being organized in parallel arrays..

NUMA Non-Uniform Memory Access is a wildly-used hardware organization of the compute units and the memories designed for single-node multiprocessing. In such a hardware architecture, the memory-access time depends on the position of the data within the memory architecture across the different CPUs..

RAM Random-Access Memory is a form of non-persistent (flushed when powered-off) computer memory, typically used to store running processes data and instruction. In modern DDR4 technologies, a RAM memory is roughly ten to a thousand times faster than flash (persistent) memories..

List of Acronyms

ALU	Arithmetic Logic Unit.
API	Application Programming Interface.
BLAC	Basic Linear Algebra Computation.
BLAS	Basic Linear Algebra Subprograms.
CPU	Core (or central) processing unit.
DDLGS	Dynamic Data-Layout Generation for Scratchpad.
DLD	Data-Layout Decision problem.
DSL	Domain-Specific Language.
GPU	Graphics Processing Unit.
HARDSI	Hardware-Adapted Refactoring of Data Structure Implementation.
HPC	High-Performance Computing.
HW	Hardware.
ISA	Instruction Set Architecture.
LL	Linear Language.
LLIA	Last-Level Indirection Array.
LTO	Link Time Optimization.
MMU	Memory-Management Unit.
MPI	Message Passing Interface.
MSR	Model-Specific Register.
OS	Operating System.
PMU	Performance Monitoring Unit.
SIMD	Single Instruction Multiple Data.
SW	Software.
TLB	Translation Lookaside Buffer.
TPU	Tensor Processing Unit.
VLIW	Very Large Instruction Word.

Mathematical Notations for Cache-Miss Modeling

$@_{(x,y)}$	Virtual adress of the cell (x, y) of a matrix.
B	Size of a dynamic memory allocator basic block (as defined by the *Linux GLIBC* implementation [39]).
C	Size in Bytes of a line of an LLC cache.
C_{total}	Total number of lines within an LLC cache.
D	Memory-size (in Bytes) of a cell of a matrix.
L	Number of contiguous and lower addresses to a given $@_{(x,y)}$ that are already preloaded in the LLC cache when accessing $@_{(x,y)}$.
N	Width and length of a square matrix.
X	Width of a non square matrix.
Y	Height of a non square matrix.
x	Abscissa of a given cell of a matrix.
y	Ordinate of a given cell of a matrix.

Mathematical Notations for DLD Resolution

$H^{f,v}_{Matrix}$	Histogram (memory signature) of the values in T^f_v.
T_v	List of all the cell accesses (address or index) to a matrix varable v during an input-code execution.
T^f_v	Transformation of the list of access T_v based on the transformation function f.

CHAPTER 1

Introduction

1.1 Historical Overview 5
1.2 Context and Objectives 6
1.3 Global Approach 7
1.4 Requirements and Issues to Overcome 9
1.5 Contributions 9
1.6 Experimental Setup 10
1.7 Thesis Structure 11

1.1 Historical Overview

Hardware processor and memory performance evolution has always been driven by technology evolution. The first influential technological parameter is the size of a transistor. Reducing the size of a transistor allowed to produce faster circuits (shorter data-transfer paths) with a lower energy consumption. This tendency led, following Moore's law [74], to ship a huge amount of transistors on a single ship. Today's embedded processors use a technology-design as small as $5nm$. However, this size-reduction seems to have reached a limit [75].
A second parameter that shapes computer-performance is the hardware architecture. Since the first monolithic processor (*Intel 4004*), successive inventions have each brought an order of magnitude in terms of computational power and efficiency. All these architectures and improvements demonstrated a significant increase in the peak and the best-case computational performance. The instruction-pipeline operators allowed to improve the computing operators throughput with the drawback of augmenting the latency. Multi-level data and instruction caches allowed to improve the data-access latency with the drawback of complicating the data-access time predictability. Hyperthreading, branch prediction and multiprocessing allowed to increase the computational throughput while dramatically complicating the performance prediction and evaluation.

In parallel, the design of the compilers had to painfully integrate the support to all the hardware evolution. This pushed compiler's back-end to integrate processor models. Pipelined architectures are supported by a precise instruction scheduling. Distributed data caches and multiprocessors are supported by complex program representations such as the polyhedral model. However, the question that remains is: is the software generated by modern compilers and programming models able to largely take advantage of all the proposed hardware capacities.

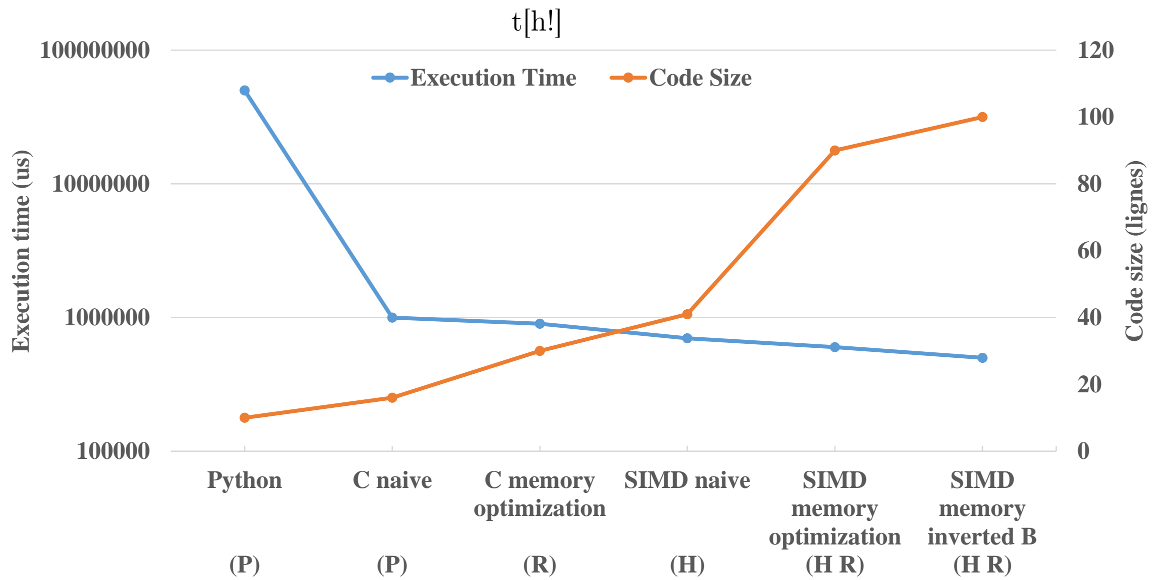


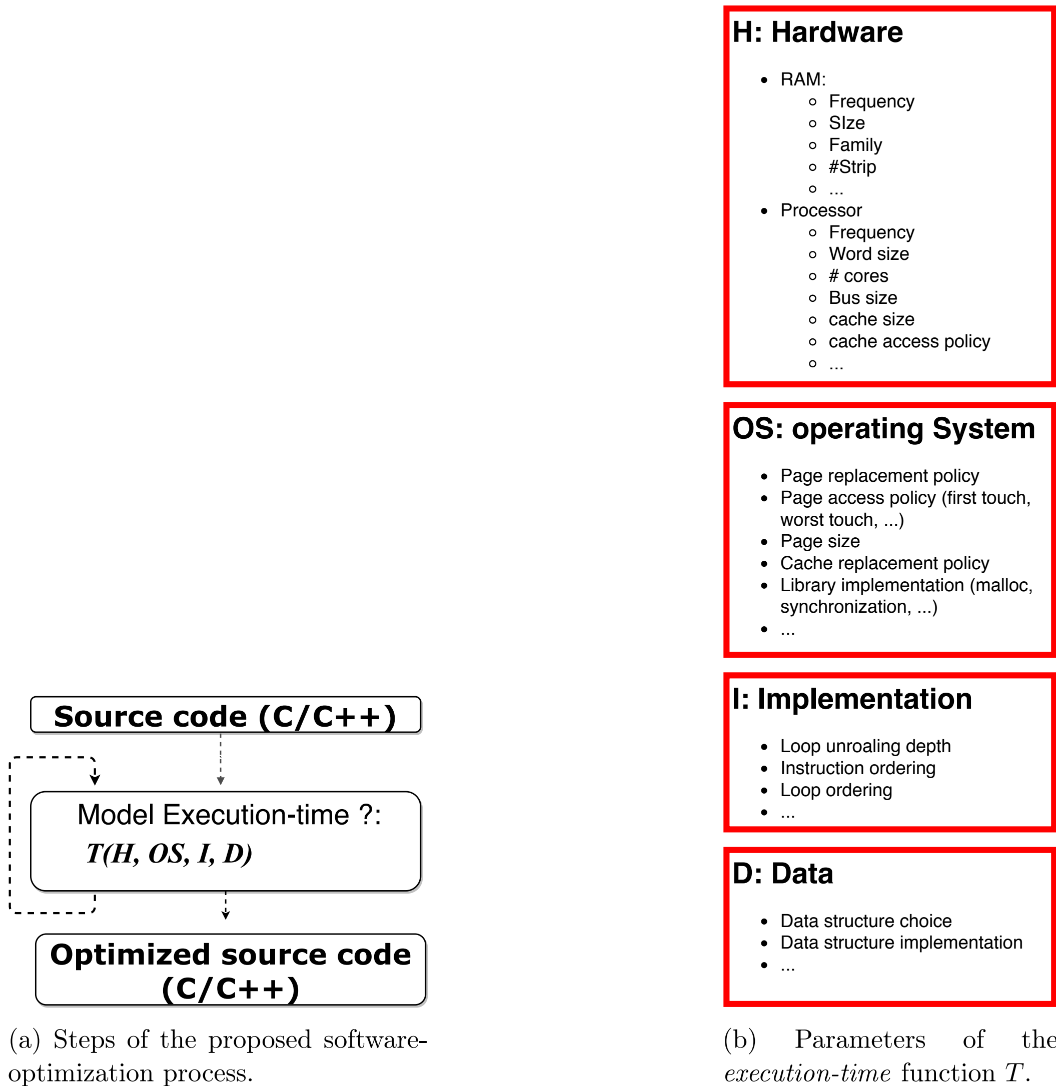
Figure 1.1: Custom evaluation of performance and code-complexity for a simple matrix multiplication application (100*100 integers) using different software implementations. The potential properties of each implementation are (P) Portable to new hardware/software (H) uses hardware-specific instructions (R) requires run-time code-refactoring

1.2 Context and Objectives

Modern High-Performance Computing (HPC) architectures encompass a large and varied set of heterogeneous accelerators. Some of them allow to accelerate computational operations (e.g. Single Instruction Multiple Data (SIMD) vector processors, Graphics Processing Units (GPUs), Tensor Processing Units (TPUs)). Others soften the data-fetch latency (e.g. caches, scratchpads and TLBs). A last category optimizes the scheduling and the interaction between memory and compute operations (e.g. *Intel's* Hyper-Threaded CPUs, C-SRAM). These hardware accelerators may bring a very interesting time and energy-efficiency improvement to a software code. Consequently, software-optimization literature has mainly focused on adapting an input source-code to a given accelerator. In Figure 1.1, we show through the example of a matrix-multiplication algorithm that such an approach has accelerated the code's execution by roughly $100X$ for an input matrix of size 100. However, this execution-time improvement has been reached at the expense of the code simplicity. The size of the code has increased by over $20X$ (regardless of the system libraries and Application Programming Interfaces (APIs) used to access each accelerator). Within an industrial ecosystem, such a size increase makes it complicated to debug, maintain and improve the code.
Moreover, a code deployed on a hardware accelerator is usually specifically developed for the considered hardware environment. The specific API of the environment is used. The data is specifically tiled and split across the considered memories, and

the data fetch is scheduled according to the hardware specificity (size of the memory and its subsystems, size of the transfer-buses, policy of access and replacements). Consequently, all this engineering time and efforts has to be spent again whenever the code is ported to a new family of hardware. Similarly, slightly modifying the code that has been optimized requires the same amount of work.

1.3 Global Approach



(a) Steps of the proposed software-optimization process.

(b) Parameters of the *execution-time* function T.

Figure 1.2: Global view of the proposed software-optimization process.

From a mathematical point of view, a source code may be mapped to its specific execution-time function T. As shown in Figure 1.2a, this multidimensional function expresses the execution time of the corresponding code with respect to the value of different external parameters. In Figure 1.2b, we summarize

the different families of adjustable-parameters that are known to influence the execution-time of a kernel: hardware, operating system, implementation, and data-placement. In this context, optimizing a software is equivalent to finding the input parameters that would minimize the corresponding execution-time function T.

The optimization process introduced in Figure 1.2a is in no mean implemented nor evaluated in this thesis. However, this thesis identifies different requirements linked to this new family of optimization approach. The different contributions of this thesis are build in order to be eventually integrated within a future tool implementing the optimization approach in Figure 1.2a.

As shown in Figure 1.2a, the considered optimization approach consists in first removing the parameters that are known not to be of any interest regarding the considered software/hardware environment. Then, the process consists different iterations aiming at converging toward an optimized value of these parameters. The parameters are split into two groups: the discrete and the non-discrete ones. For the discrete parameters, the approach is based on the simplex algorithm [26]. This algorithm is primarily used due to its robustness. It may be applied to analyze parameters with large or small set of potential values. It may also be used with non-ordered parameters (e.g. cache-allocation policy, page-replacement policy, library choice). This algorithm may only be used under the assumption that all the considered parameters are either integer or may be mapped to a finite integer set. Meanwhile, the non-discrete parameters[1] (such as the processor or the memory frequency) are determined using the gradient-descent algorithm [91]. This approach is primarily picked as it allows slight changes in the target function (T) between two consecutive iterations of the algorithm (*Haskell Curry* adaptation of the gradient-descent for convergence with non-linear optimization [111]). This property is used in order to consider several parameters in parallel: one or several simplex executions (corresponding to one or several discrete parameters) may be run along with a gradient-descent execution (corresponding to a non-discrete parameter).

Showing the optimality of the optimization found by the proposed approach is out of the scope of this study. However, this numerical approach for software optimization ensures that most important parameters for software-performance are explored. It also allows to use prior optimization-knowledge by favoring some parameters over others or by reducing the research-space. Finally, by introducing hardware and OS-related parameters, our approach sets the basis for a new way of software optimization: adaptation to the host hardware.

[1] These parameters are all continue and differentiable.

1.4 Requirements and Issues to Overcome

In order to implement our software-optimization approach, different problems need to be treated.
On the one hand, the execution-time function T may hardly be accurately evaluated for any given input source code. To the best of our knowledge, no solution has been proposed to evaluate this function T with regards to all the parameters that we identified. Even though mathematical modelings of T exist for some specific input codes [112, 44, 79, 64, 31], the corresponding models are either constant with respect to the considered parameters [112, 44, 79] or non derivable with respect to some non-discrete parameters [64, 31].
On the other hand, in order to run the optimization approaches described in section 1.3, we need to either have sufficient evaluation of the considered function T or be able to evaluate a stochastic function related to T. In both cases, we need to measure or predict the execution time of the given code[2] within the considered hardware/software environment. In section 2.1 we show that modern hardware-software simulator or executors have non negligible time constraints. Consequently, it would be impossible to implement our method using such simulators or executors at the execution-time scale of a compiler.
In this context, our approach consists in replacing the execution-time function T with a similar but easiest-to-evaluate one. For instance, if we consider that the number of data-cache misses has a direct impact on the execution-time, we may substitute the function T with a model of the number of cache misses.

1.5 Contributions

This thesis aims at proposing the ground basis to the automatic adaptation of software-code across high-performance hardware architectures. This consists in proposing different solutions and tools aiming to be eventually integrated within the proposed optimization process (Figure 1.2b). In this context, the main contributions of this thesis are:

(i). HARDSI, a patented and custom framework for automatic exploration of data-layout implementations. The corresponding source-to-source compiler is a standalone tool designed to replace each data-structure's instance with an optimized implementation. This optimization for general-purpose *C/C++* code is based on memory-pattern detection.

(ii). DDLGS, a method designed to automatically and dynamically generate data-layout-implementations for programmable (fast and explicitly-addressed) memories. The generated code allows to adapt at run-time to the dimension of the problem as well as the pattern followed to access the data.

[2] Through simulation or execution.

(iii). An evaluation of the performance-improvement brought by our source-to-source compiler method.

1.6 Experimental Setup

In this thesis, we deployed and evaluated our solutions on two different hardware platforms. The first one is an *x86* architecture implementing an *Intel Xeon E3-1270 v4* processor with an L3 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 (chapter 4). The *Perfmon2* (libpfm-4.11.0) library [32] allong with our custom patch (subsection 2.3.2) is used to access the Performance Monitoring Unit (PMU) of the *x86* 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.

The second hardware platform that we considered is a MPPA *Kalray Coolidge-80-30*. This *Coolidge* processor implements five clusters, where each one contains sixteen in-order *K1c* Very Large Instruction Word (VLIW) cores. Each core shares a 4M Bytes DDR main memory with all the cores on the same cluster and communicates with the other clusters through a NoC. Each core has also a 4-way associative L1 cache LLC containing a total of 16K Bytes made of 64 Bytes per cache-line and implementing the LRU cache-replacement policy. Even though a larger data cache (L2) is present on this platform, we do not consider it. Indeed, unlike the L2 and L3 caches of our *x86* processor, the L2 cache of the *Coolidge* processor is primarily used as a fast communication buffer between concurrent cores. It is not an extra caching level in between the main memory and the CPU. Thus, we assume that the L2 cache of the *Coolidge* processor has a fairly-negligible impact on the performance of our single-threaded target applications (as we bind each thread on a unique core). Additionally, the considered *Coolidge* processor implements a 16K Bytes scratchpad memory per core. This fast on-chip memory is explicitly accessed by the programmer through the builtin library embedded to the *Kalray-1* (4.0.0) tool-chain. In this thesis, we simulate the whole HW/SW stack using the *Kalray-1* (4.0.0) simulation platform. The *–cycle-based* option of the simulator is used to ensure a quasi cycle-accurate execution. The *k1-cos-g++* (4.0.1) based on the *g++* (7.4.1) compiler (with the -03 optimization option) is used to compile all the considered computation-kernels. The builtin library of the simulator is used to access the performance registers and the Memory-Management Unit (MMU) of the *Coolidge* processor in order to measure different cache misses and CPU cycles. As far as we know, no *automatic CPU-frequency scaling* is implemented in the used simulator.

All the presented performance results are obtained following the same procedural method (on both considered processors). Each 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 of both considered processors.

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

1.7 Thesis Structure

The rest of this thesis is organized as follows.
Part I presents the state of the art in terms of software optimizations. Chapter 2 discusses existing tools and methodologies for software-performance evaluation, measurement, modeling and estimation. Chapter 3 evaluates the existing solutions for software optimization. A particular attention is given to the optimizations by adaptation to the hardware and the software optimizations for scratchpad usage. It then focuses on what we consider as a major problem to solve for software adaptation: the Data-Layout Decision problem (DLD).

Part II presents our contributions toward an automatic software adaptation to hardware. Chapter 4 introduces HARDSI a custom patented methodology to solve the data-layout decision problem by adapting an input implementation to the host hardware-memory hierarchy. It also presents the principles and the usage of the proposed DSL and compilation framework. Chapter 5 uses the instance of programmable memories to show how to extend the proposed methodology to hardware memories with no dedicated data-layout implementations. The chapter also presents DDLGS a custom patented method designed to dynamically generate scratchpad-dedicated dynamic data-layout implementations.

Part III presents through chapter6 an experimental evaluation of our method to select an adequate data-layout implementation across different hardware platforms. It also evaluates the performance gain brought by our automatically-generated code.

Chapter 7 concludes the thesis and discusses different improvements and applications for the software-optimization approach that we propose.

Part I

State of the Art and Scientific Methodology

Chapter 2

Scientific Methodology: Performance Exploration

2.1 Simulation and Emulation Tools **16**
2.1.1 Trace Injection Simulators 16
2.1.2 Cache, RAM and Register Simulators 17
2.1.3 Instruction-Interpretation Simulators 18
2.1.4 Trade-off Between Accuracy and Execution time 19
2.1.5 Conclusion 19
2.2 Hardware and Software Performance Modeling: The *Roofline* model **20**
2.2.1 Hardware Boundaries (Roof) 21
2.2.2 Fitting the Model to the Memory/Computational Optimizations (Ceiling) 21
2.2.3 Interest of the *Roofline* Model 23
2.2.4 Conclusion: Limitation of the *Roofline* Model for our Approach . . . 23
2.3 Performance Measurement **23**
2.3.1 Hardware Performance Counters 24
2.3.2 Performance Measurement Libraries 26
2.3.3 Conclusion: Choosing a Hardware-Software Performance Tracker . . 29
2.4 Conclusion: Adopted method and implementation choices **31**

The software optimization approach that we propose is highly data-dependent. In order to efficiently run its heuristic, our method needs to have, at each iteration, an accurate evaluation of the function that it minimizes (whether it is the execution time function or any related performance function). In this chapter, we first make in section 2.1 a state of the art in terms of simulation and execution-tools designed for hardware-exploration. Second, we explore in section 2.2 the existing performance-modeling methods. The objective being to determine whether or not these models may represent an alternative to the time-consuming code-execution or simulation for performance evaluation. Third, we investigate and compare in section 2.3 the different software and hardware approaches to reliably measure different performance indicators (such as execution-time, memory latency, cache misses). Finally we conclude by choosing the adequate tools for our software optimization implementations with the expected time, reliability and accuracy constraints.

2.1 Simulation and Emulation Tools

We refer to *hardware-simulation* or *emulation* any tool designed to replicate by the means of software the functional behavior of a hardware. The difference between *simulation* and *emulation* lays in the system hosting the replication. A *simulation* is based on a software implementation of a model where the internal functions of the original systems are not taken into consideration. Meanwhile an *emulation* is a replica of the internal system functions on a different host hardware. In the rest of this thesis, we refer to both *simulation* and *emulation* tools as *simulation* regardless to the host hardware. We also refer to the hardware platform where the simulator is run as the *host hardware*. Finally, we refer to the hardware platform that is simulated as the *target hardware*

2.1.1 Trace Injection Simulators

One of the main distinctive aspect of a hardware simulation is its time-overhead compared to a simple execution. Indeed, simulating an instruction from the target hardware at software level is equivalent to recognizing, translating and interpreting it in the host-hardware instruction-set. Given that each one of these steps is equivalent to several instructions from the host hardware, a simulation-time maybe prohibitively long for many applications.
One way to reduce the simulation-time, is to reduce the level of accuracy in the formal description of the target architecture. This is often achieved using trace-based (or trace-driven) simulators [106, 90, 101, 8]. These simulators take as input a fixed sequence of trace-records relative to the execution of the corresponding software on the target hardware. Then the simulator only evaluates the instructions which results are not in the trace file. This trace may record memory references, branch outcomes or computational instructions. In order to cover a large set of potential uses, the traces fed to a simulator are usually produced by an automatic trace generator.

In an attempt to reduce the simulation time, some random-traffic generators have been proposed. Even though generating random traces is easy, it usually results in traces that diverge from effective executions. In addition to the obvious functional default of the resulting execution, this divergence forbids the deduction of any performance-evaluation from such a simulation [101, 8].
In order to understand the inaccuracy of performance-evaluation derived from trace-injection-simulators, let us consider how the memories hierarchies ar simulated. Indeed, the layered memory-systems are known to be a root cause for performance; and this high incidence is mainly related to the cache-hierarchy. Data, instruction and address caches are mainly maintained through coherency-messages. These message follow a non-deterministic and non uniform distribution. Thus, the random traffic-generators are very-likely to produce a trace that follows a different distribution [101, 8]. In order to tackle this challenge of cache-simulation accuracy, different

attempts have been made to propose a trace-generator that dynamically adapts the cache-coherency message-distribution to the observed memory-accesses [66, 67, 118]. However, as shown in [8], the cache-coherency message distribution is significantly influenced by other parameters than the memory access pattern (such as the burden on the machine [8] and the hardware message passing protocols [81]). Thus, it remains difficult to generate a traffic that accurately simulate the performance-critical memory-hierarchy behavior.

2.1.2 Cache, RAM and Register Simulators

Given the difficulty stated in subsection 2.1.1 to accurately simulate the whole memory hierarchy, different families of hardware simulators have been proposed to answer different functional requirements. In this section, we focus on the hardware simulators designed for hardware-fault evaluation on a single machine.

Reliability is a major objective for cyber-physical systems. Indeed, computing systems are inclined to create a faulty behavior due to manufacturing defects, electromagnetic interference or any other environmental perturbation. Some faulty behaviors are easy to spot as they correspond to a clearly-damaged hardware. However, some other faults such as the *Byzantine* faults may inject computational errors that are much harder to spot, leading to potentially catastrophic failures. In order to evaluate the effect of faulty behaviors on a simulated hardware, it is mandatory to first accurately simulate the fault-injection.
In [52], the authors propose a fast and flexible hardware-emulator framework designed to evaluate the reliability of a cyber-physical system. The main interest of this framework, compared to existing emulators such as *Gem5* [12] and *SimpleScalar* [15], resides in its memory-subsystem emulator. In order to accurately locate the hardware component responsible for a given fault, the author proposes to subdivide the memory simulator in different components: RAM, cache and registers. Each one of these parts is simulated based on its functional algorithm. These parts are also parameterized based on their functional properties (size, associative property, access policy). Similarly to *Gem5* [12] and SimpleScalar [15], the simulation time in [52] is dramatically reduced by simplifying the model of the components.

Evaluating the occurrence-frequency of faulty behaviors is important for hardware-performance evaluation. Indeed, such behaviors are known to be extremely time consuming [85, 52, 8, 2, 33]. Let us consider Byzantine faults[1]. In [85], the authors evaluate the average time overhead of intermittent hardware-errors [2] on general-purpose computing benchmarks. Using the *Microsoft Windows* error reporting system, this time overhead is evaluated to up to 39% of the execution

[1] Faults that generate erroneous results without crashing nor stopping the functioning of the hardware.

[2] Intermittent hardware errors represents about 40% of the total hardware failures within the considered benchmark.

time related to memory accesses. Similarly, the different countermeasures that have been deployed to track and correct hardware faults have a non-negligible effect on hardware-performance. Given the relative complexity of modern architectures, these countermeasures are largely deployed on high-performance and embedded platforms. Even optimized approaches such as [49, 33] have a minimal time overhead of roughly 6% on memory access within general purpose architectures and benchmarks.

2.1.3 Instruction-Interpretation Simulators

The simulation tools presented in subsection 2.1.1 and subsection 2.1.2 are widely used in hardware design. However, to the best of our knowledge, no attempt has succeeded in using such hardware-simulation tools for performance evaluation. The reason for that is the granularity at which the hardware is simulated.
Indeed, one of the most time-accurate families of hardware simulators are the instruction-interpretation simulators. These simulators interpret at software-level each low-level instruction of a given execution flow. Since 1990, instruction-interpretation simulators have been used for hardware exploration [77, 17, 69, 12, 15]. Given the relatively important simulation time, most studies have only focused on a specific part of the hardware at a time. In [77], the authors use the *Wisconsin Wind Tunnel* simulator to evaluate some cache-coherency protocols on a manycore architecture. Similarly, other studies have used platforms based on *SystemC* for architecture evaluation [105, 69, 17]. Different levels of description have been proposed from signal accurate in [105, 69] to message-accurate in [17]. However, all of the *SystemC*-based solutions have a well known time-efficiency drawback. This excessive overhead is mainly linked to the exponential complexity of these simulator: the modeling implemented in *SystemC* encompasses all the transient state.

Simulators that interpret low-level instruction are often known as *instruction* or *quasi cycle-accurate*. However, in order to have an accurate performance-evaluation of a hardware/software couple, an accurate estimation is required for most instructions and micro-instructions. To the best of our knowledge, one of the most promising work in terms of timing model is proposed by Rosa et al [90]. In this paper, the authors show that the accuracy of the proposed CPU model varies from 0.06% up to 10.56 depending on the used benchmark. Even though other models have shown a lower average accuracy, the model proposed in [90] is the first one to be non-constant (or composed of constant steps in chunks) with regards to CPU parameters (such as the frequency, or the number of cores used). Consequently, this model is the first to be adequate to spot the variable parameters that might bring a significant performance gain and evaluate this potential gain. However, this model only focuses on CPU timing. Thus, it does not consider a performance-critical aspect which is memory accesses. This model is also designed exclusively for real-time environment running on embedded systems (with hard timing constraints).

2.1.4 Trade-off Between Accuracy and Execution time

Processor	Memory System		
CPU Model	Classic	Ruby	
		Simple	Garnet
Atomic Simple	Speed		
Timing Simple			
InOrder			
03			Accuracy

Figure 2.1: Trade-off between simulation-time and hardware-model accuracy (extract from [16]).

In this section, we focus on *Gem5* [12], one of the most widespread hardware simulator. The *Gem5* is used in several domains, from hardware-design up to computational-language validation. Various researches [86, 3, 65, 30] have been conducted to adapt the simulator to the different hardware-simulation paradigms described in subsection 2.1.1, 2.1.2 and 2.1.3. These researches have also covered different parts and combinations of the described hardware (processor, memories, network and peripherals). Consequently, a study of the *Gem5* simulator gives an insight about the advantages, drawbacks and trade-offs to expect from a modern hardware simulator.

The different variants and simulation mode of the *Gem5* simulator have been compared within [16]. This comparison is summarized in Figure 2.1. We may notice the different CPU models of the simulator as well as the simulation modes and the memory models. If we consider the example of the memory hierarchy, these different parameters allow to simulate the cache coherency protocols at different scale. The *classic* mode emulates the cache-coherency messages using a constant-cost unsafe function. The *Ruby* mode and its *Garnet* variant allows a more precise simulation of the on-chip network.
It is noteworthy that the main limit to reach an accurate simulation, the main limit is the simulation-time.

2.1.5 Conclusion

The outrageous simulation time of modern hardware platform makes it impossible to use a simulator within the optimization methodology introduced in section 1.3. Indeed, the amount of data required by our method can not be simulated within less than months for very simple peace of codes that would run in few seconds on

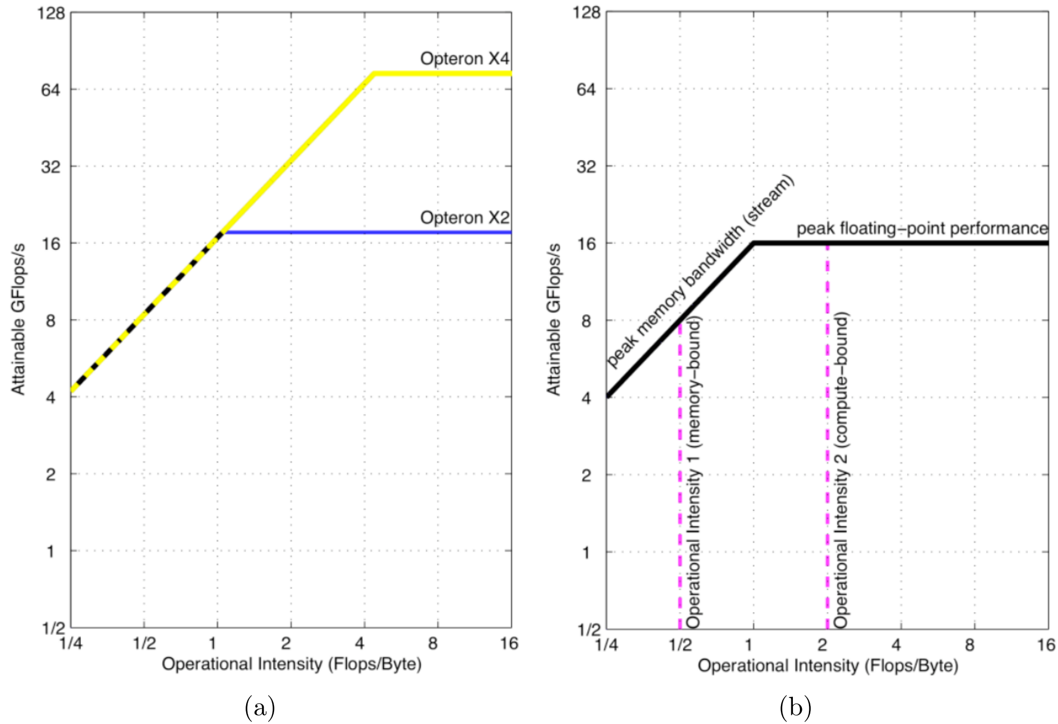


Figure 2.2: Roofline Model for (a) AMD Opteron X2 and (b) Opteron X2 vs. Opteron X4 (extract from [112]).

the physical platform.

Meanwhile, our initial experiments with *Gem5* show that lowering the simulation-model accuracy is not an acceptable workaround. Even if we only slightly go up within the speed-accuracy curve in Figure 2.1, the performance results become unexploitable. Indeed, the performance function looses its dependency from most considered hardware parameters, while the physical execution shows a clear dependency to the same parameters. Loosing such functional dependencies is crippling for our analytical approach.

Finally, following the same experimentation, we concluded that using a trace-based version of *Gem5* does not solve our simulation-time issue. Indeed, even though this simulation mode may reduce the number of instructions simulated, the corresponding timing model requires a high number of instructions in order to reach an accurate-enough estimation of the execution time.

2.2 Hardware and Software Performance Modeling: The *Roofline* model

The emergence of multi-core hardware architectures has significantly increased the difficulty to assess the performance of software and kernels. A simple and yet accurate approximation of the performance of these architectures is thus mandatory

to help software developers evaluate their programs and decide which optimization strategy to implement.

2.2.1 Hardware Boundaries (Roof)

The *Roofline* model [112] is an intuitive visual performance model used to asses a hardware/kernel couple. It allows, as a first approximation, to bound the computation performance (expressed in terms of floating-point) of a given multi-core processor according to its memory-bandwidth performance. An instance of this model for two AMD processors is presented in Figure 2.2. The main concept that the *Roofline* model lays on is the *operational intensity*: operation per byte of DRAM traffic. This allows the model to relate to a highly-constraining resource: the off-chip memory bandwidth. As shown in Figure 2.2b, the *Roofline* model indicates for a given kernel (defined by its operational intensity) whether the potential performance bottleneck is the computations or the memory accesses. This upper bound of the considered hardware architecture is built using:

- The hardware peak floating-point (delivered by the manufacturer), which gives the compute-bound section (constant) of the model.

- The hardware peak memory-bandwidth (delivered by the manufacturer or through micro-benchmarks) and the operational intensity (specific to a kernel or a program). These two measures give the memory-bound section (linearly increasing) of the model.

2.2.2 Fitting the Model to the Memory/Computational Optimizations (Ceiling)

In order to model more accurately the performance of a kernel running on the considered hardware, the *Roofline* model allows to enhance the estimation accuracy by reducing the previously defined boundary. This reduction is processed according to the optimizations implemented by the software: each optimization is mapped to the threshold curve that it potentially allows to surpass. The model allows to distinguish between two types of optimizations. The first type is the memory-linked optimizations instantiated in Figure 2.3a. Its impact on the performance is assessed by evaluating the reduction brought by the considered optimization on the memory transfers (hence the repercussion on the operational intensity). The second type of improvement is the compute optimizations instantiated in Figure 2.3b. Its impact is assessed through the evaluation of the gain in terms of computed operations in Flops. The objective of this model refinement is to rank the potential optimizations based on the corresponding gain. It also aims at determining the ones that would have no impact.

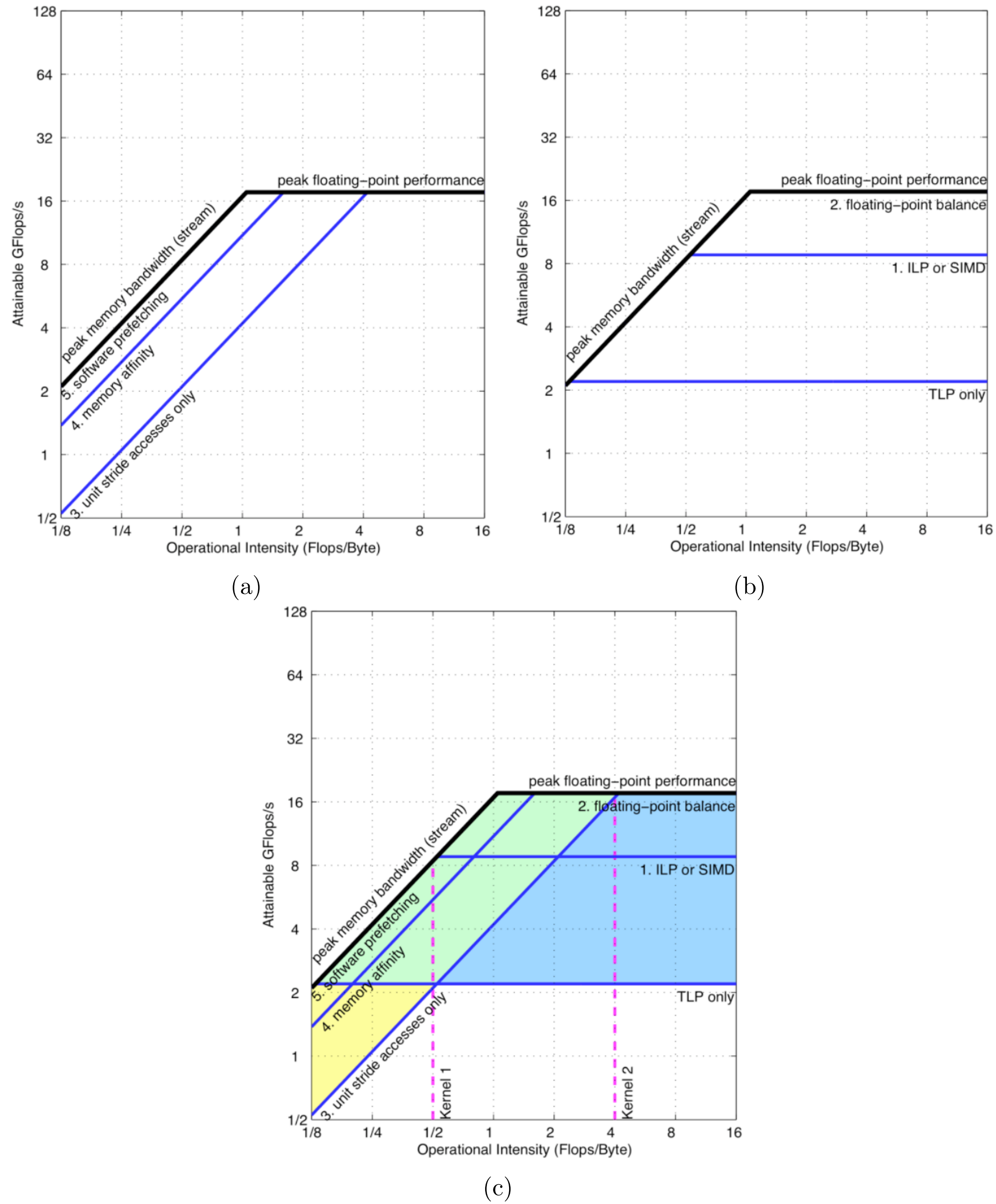


Figure 2.3: Refining the *Roofline* model depending on the used (a) memory optimization and (b) compute optimization. Data corresponding to an AMD Opteron X2 and (c) Opteron X2 processor (extract from [112]).

2.2.3 Interest of the *Roofline* Model

The main interest of the *Roofline* model is its user-friendliness. The developer may simply generate a performance approximation of his software and host hardware; this brings a precious help to identify the software optimizations that worth being implemented. The *Roofline* model also allows a relatively accurate evaluation of the impact of memory on the considered performance. The model considers the exchanges between the DRAM and the caches to measure the memory impact. Indeed, the model considers the exchanges between the DRAM and the caches as a measure of the memory impact. Such an access is most often longer than an access between the processor and the caches (excluding DMA processors). Thus, it is more likely to represent a bottleneck resource.

2.2.4 Conclusion: Limitation of the *Roofline* Model for our Approach

In the *Roofline* model, the upper-bound linked to each memory optimization (ceiling) is said to have no impact on the performance of a computation optimization. Indeed, memory optimizations might very likely be done through extra computations (trade-off between computation and memory). Consequently, these optimizations might compete with the payload computations. Hence, the reduction in the peak floating-point performance.
However, the impact of the considered memory optimizations is reflected on the increase of the computations required to reach the peak floating-point performance (steady-state). This increase might be observed through different variants in the definition of the operational intensity [112, 44, 79].
Similarly, even though the *Roofline* model is a very handy tool for guiding programmer optimization, it is harder to integrated to an automated software optimizer. The main reason is that the model shows no impact of memory optimizations on the operation intensity of the kernel.

2.3 Performance Measurement

Optimizing an application for a given hardware platform has been increasingly difficult through the last decades. The challenge of such a task comes mainly from the increasing complexity of modern microarchitectures, the diversity of workloads and the huge amount of data produced by performance tools. Meanwhile, the constantly growing distance between compute and memory-access time creates a new yet acknowledged challenge for hardware performance counters [13]: The different attributes (such as the frequency, access time and operation rate) of these two hardware parts lay at different orders of magnitude. This difference makes it difficult to evaluate within the same hardware parts (PMU, performance registers or performance monitoring software tools) the impact of memory or computation on performance.

In this section, we give an insight about the challenges and solutions proposed to accurately measure performance within a HPC ecosystem. The objective of the presented tools and methods is to identify and evaluate the impact of performance bottleneck. For the sake of clarity, we use the terminology and example used by the *Intel* processor and performance tools. However, similar principles are used by other hardware platforms with potentially a different terminology.

2.3.1 Hardware Performance Counters

Performance monitoring is a predominant issue among hardware designer. In modern hardware platforms, different devices are shipped and used for software-performance measurement. Initially, these hardware-performance counters were designed for other purposes than software monitoring (e.g. tension, frequency or derived physical quantities) [76]. Consequently, different accuracy and correctness issues have to be considered when using these hardware counters for software-performance measurement.

Nowadays, hardware performance monitoring has become intrinsically linked to the hosted software. It refers to *any hardware mechanism that enables (not necessarily by design) insight into how software performs on a microprocessor. This definition includes features as simple as timer-based interrupts, but also a broad range of things like event counters, last branch buffers, instruction-based samples, and many more* [76]. For most hardware manufacturers, the performance monitoring devices are gathered within the PMU. This hardware block is usually made of different special registers called performance registers. An example of such a register is represented in Figure 2.4. At a given time, a single register may be in charge of storing different physical quantities. The sampling, multiplexing and ordering of these registers is usually managed by user libraries (subsection 2.3.2).

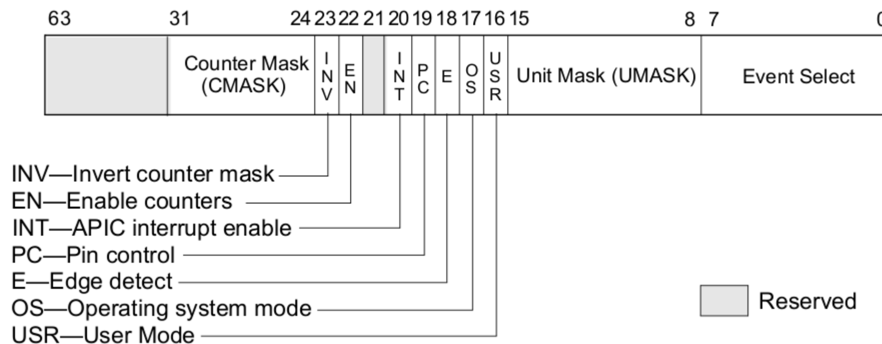


Figure 2.4: Layout of the IA32-PERFEVTSEL MSR (extract from [23]).

The main principle used for hardware performance-performance monitoring is the *event*. It defines the physical quantity that is measured. In the *Intel* hierarchy,

each event is encoded within the two first logical gaps of MSR (see bits zero to fifteen in Figure 2.4). These events are divided within five categories, namely (i) *program characterization*, (ii) *memory accesses*, (iii) *pipeline stalls*, (iv) *branch prediction*, and (v) *resource utilization*. When the event detector of a PMU receives a given event, the MSR that has initially been mapped to the considered event is updated depending on the type of event, the MSR configuration and the type of performance measurement. The first type of performance measurement is known as *counting* performance measurement. It consists of measuring the total number of events that arises in a given time slot. The second type is known as *event-based sampling*. It consists of triggering an interruption (overflow) to the processor whenever a configured number of the considered event has been triggered. Then the interruption is handled by instructing the PMU to save the status flags of the used MSR in order to be latter reported to the user.

(i). **Program characterization** events allow to monitor low-level instructions independently from the processor's implementation or the Instruction Set Architecture (ISA). This gathers all the metrics that are only relative to the software layers (from the operating system up to the end-user program) such as the number of load, store, branch of floating point instructions.

(ii). **Memory accesses** events are probably the most relevant to detect performance bottlenecks in the context of the memory wall. A use-case of such events is the enumerating of hit/misses relative to the different data, instruction and TLB caches.

(iii). **Pipeline stalls** indicate how filled is the instruction pipeline of the processor. Even though such an information would be very useful to determine the operational intensity of a kernel, it is hardly usable in the context of multicore or a manycore processors. The pipeline-filling rate observed through this pipeline-stall events gathers instructions belonging to all the processes sharing the same pipeline (hence the same CPU). Thus, the information regarding a given process is drowned by the instructions of the concurrent processors.

(iv). **Branch prediction** events allow to monitor predictive-branching instructions generated by the compiler, the prefetcher or the hardware branch predictor.

(v). **Resource utilization** encompasses all the events triggered when the processor access some specific ressource (e.g floating point accelerator, data, instruction or TLB caches).

The performance counters have proven to be very useful in hardware and software co-development. They have allowed for instance to fix *FDIV*, a major performance bug on the *Intel Pentium III Pro* [84]. However, the design of MSRs and more generally of PMUs is also known for its different and complex performance bugs.

First, overflows are a very recurrent and hard-to-spot issue for PMU registers. Indeed, the number of bits reserved to store a given physical quantity within an MSR is fixed. Given that the encoding method used is also fixed, an overflow on a performance register may not be treated[3]. Meanwhile, given that the register keeps only minimal information regarding the user-process that is tracked, an overflow is only stored on a register until the next trigger of the same event. Once the new event is triggered, the performance register does not show that the stored information is not relevant anymore.

Second, the accuracy of a performance measurement based on a PMU register may significantly and inexorably suffer from the MSR design choice. The number of MSRs is usually much lower than the number of handled performance-events (roughly ten physical registers for about two hundred events). Similarly, for the same hardware limitations, different events may be measured by the same digital logic (such as the number of misses for different caches within the same CPU core). The implemented solution for these hardware issues is time multiplexing (for both registers and digital logic). The events are measured in small and disjointed periods. The total measurement is then deduced through polynomial interpolation. Given that the sampling period is constant and not calibrated to the measured event, important variations may happen out of the sampling duration. In this case, the interpolated functions would significantly vary from the real one making the end-user result unreliable.

Finally, other accuracy issues may downgrade performance-evaluation accuracy in a parallel and multi/manycore environment. Indeed, we represent in Figure 2.5 the *C/C++* code used to access a CPU-cycle count on two state-of-the-art processors implementing an event-based PMU. We notice from these code snippets that accessing performance register requires multiple low-level instructions. Even though the time stamp of these instructions may be soften by the instruction-pipeline of the processor, accessing the register is not an atomic operation. Consequently, in the context of concurrent processes, register-time multiplexing or simply important workload, this operation requires additional synchronization. In addition to the well known time constraint [81], such a synchronization may significantly influence the the execution-time of the overall code that is assessed.

2.3.2 Performance Measurement Libraries

In subsection 2.3.1 we showed the limitations of the hardware blocks designed for performance monitoring. In order to lighten these accuracy and time-overhead issues, it is mandatory to have an adequate software to manage the PMU. In this section, we present different software and libraries designed to access at user-level the different features of the hardware PMUs. The objective is to explain how and

[3] An example of such a treatment would be to change the encoding method and potentially reduce the data accuracy.

```c
void __inline
getCycleCount(uint64_t *res)
{
  uint64_t hi, lo;
  __asm volatile ("rdtsc" :
                    "=a" (lo),
                    "=d" (hi));
  /* Requires the assembly instr:
        db $0F; db $31;
        mov [TimeStamp.Lo], eax
        mov [TimeStamp.Hi], edx
  */


  *res = (uint64_t)lo;
  *res |=((uint64_t)hi)<<32;

}
```

Listing (2.1) Code for Intel x86_64 processor

```c
void __inline
getCycleCount(uint64_t *res)
{
  unsigned int t, t0, t1;

  do
  {
    __asm volatile("mftbu %0" :
                   "=r"(t0));
    __asm volatile("mftb  %0" :
                   "=r"(t));
    __asm volatile("mftbu %0" :
                   "=r"(t1));
  } while (t0 != t1);

  *res = (unsigned long long) t0;
  *res = res << 32;
  *res |= t;
}
```

Listing (2.2) Code for IBM Power8 processor

Figure 2.5: Code snippet to access an MSR looking for a CPU cycle counts on an (2.1) Intel and (2.2) IBM processor

under which adaptations can we ship these software-tracking and monitoring tools within our software optimization approach.

The first issue of hardware PMU that we consider is the time-overhead introduced to the assessed code. In this context, a first approach is to extract as much work as possible from the performance measurement routines. This computational and memory tasks are then executed by a distant process, or by the same process out of (before or after) the performance-critical code section. This approach is largely considered by the *Scalasca* project [36]. This project encompasses different software tool-sets providing a highly scalable performance tracking and analyzing frameworks for HPC platforms. One of the main interests of the *Scalasca* tool-set is the interesting distribution of the different tasks of performance tracking among a process execution. This task sharing is managed thanks to the different *Scalasca* tools.

(i). **Score-P** [50] a compiler framework based on the *gcc* [37] compiler tool chain. This framework allows, thanks to a user-level API to compile the considered source code while injecting dedicated performance tracking instructions. These tracking instructions are optimized by minimizing the number of instructions run during the performance-critical section. Only raw data are extracted from the PMU. Building the observed performance function from these data is performed once the user-code has been entirely executed.

Thanks to this approach, *Score-P* is able to increase the sampling rate of the hardware events with a minimal time-overhead impact.

(ii). **Cube** [92] a software allowing to analyze the performance results of a given software during or after its execution. In its main functioning mode, *Cube* takes as input a static file generated after the execution of a code (tracked) compiled using *Score-P*. Then, *Cube* allows to do a performance analysis of the code that has been executed (see Figure 2.6). Different metrics are considered such as the execution time and access to hardware/software resources. This allows to find the different performance bottlenecks and their locations in code.

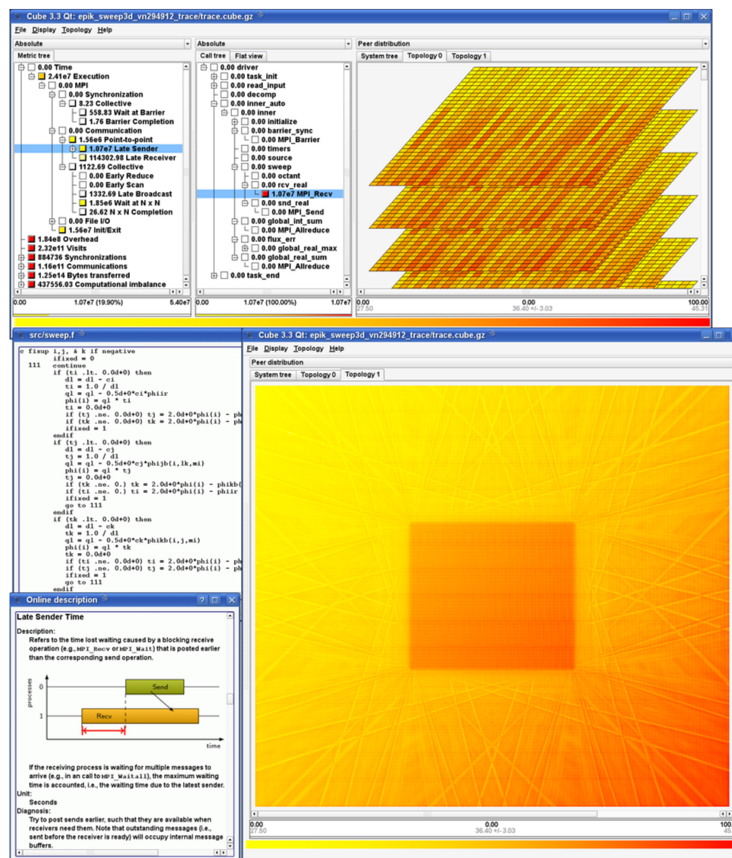


Figure 2.6: Scalasca analysis report explorer presentation of Sweep3D execution performance with 294,912 MPI processes on the IBM Blue Gene/P platform. Interface of the *Cube* tool (extract from [114]).

The *Scalasca* tools are highly optimized for massively parallel executions (e.g. multithread, multi-process, Message Passing Interface (MPI)). This optimization is mainly achieved by the concept of resource multiplexing. This allows through the virtualization of the physical MSR to sample different registers atomically (at user-level scale). However, this kind of optimization is barely efficient on a single

thread application. Indeed, the cost of MSR virtualization is only negligible when it is split among the execution-time of a high number of threads/processes.
Consequently, in the context of our optimization process, tools using the *Scalasca* approach may not be used for software optimization. The time overhead within our mono-threaded target applications is significant. Moreover, to the best of our knowledge, no evaluation nor modeling has been proposed to bound this overhead. However, given the user-friendliness of the framework, we still use it as an improvement-validation tool.

If we consider the *Linux* ecosystem, the problem is not to find a performance monitoring tool. The problem is rather to find, among all the possibilities, the adequate tool for the considered application and workload. In [32], the author splits the *Linux* profiling tools within three categories represented by their respective flagship implementation: (i) *OProfile* [21], (ii) *perfctr* [82], and (iii) *VTune* [87]. Each one of these tools is designed while targeting a specific design or metric.

(i). **OProfile** is made for system-wide (from Operating System (OS) to the end-user code).

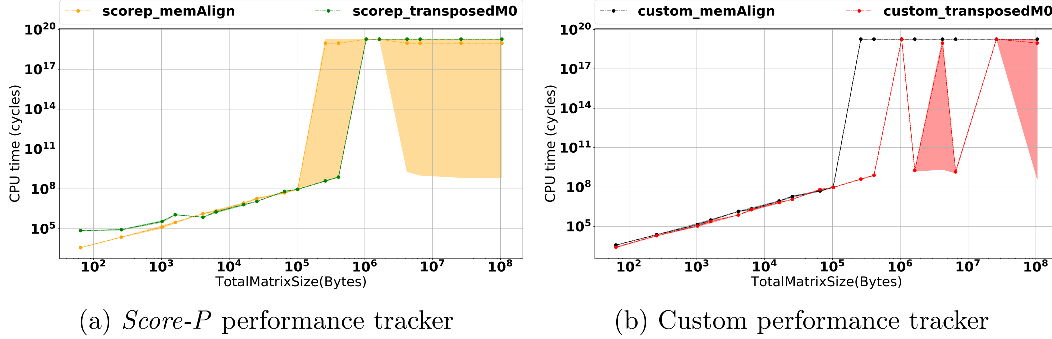
(ii). **perfctr** targets sytem monitoring at the scale of a thread.

(iii). **VTune** focuses on *Intel* architectures with a fined-grain evaluation of the driver impact on performance (using kernel privileges).

One of the main attempts in literature to conciliate all these concerns is proposed by *perfmon2* [32]. This library is conceived as a generic performance monitoring framework for different hardware platforms. This goal is mainly achieved by allowing the user to easily build interfaces with hardware specific tools (such as *VTune* for *Intel* platforms).

2.3.3 Conclusion: Choosing a Hardware-Software Performance Tracker

Our experimental experiences along with literature [57, 9, 54, 76] have thought us the potential problem of using an inadequate performance tracker. For instance, in Figure 2.7, we measure the execution time of two different implementations of a basic matrix multiplication algorithm. Both algorithms are performed on an *Intel Xeon* x86_64 architecture with three levels of data caches implementing the LRU cache-replacement policy. We use two different performance trackers: the *Score-P* performance tracker (see Figure 2.7a) and our custom performance tracker (see Figure 2.7b). For each input-matrix size, we evaluate each code with each tracker ten times. In Figure 2.7 we first notice that the experimental evaluation of each implementation is different from one tracker to another. Moreover, we may notice that the bias applied to each implementation is not constant nor regular. Using the *Score-P* tracker the *memAlign* implementation successively faster then slower than *transposedM0* implementation while using the custom tracker the



(a) *Score-P* performance tracker

(b) Custom performance tracker

Figure 2.7: Performance benchmark of two implementations of a basic matrix multiplication algorithm [57]. The performance tracker used is (a) *Score-P* versus (b) our custom MSR-based code (inspired from Listing 2.1). The dots are the average value of ten consecutive measurements. The shaded curves denote the distance between the minimal and maximal values of the ten measurement. The experiments follow the experimental setup described in section 1.6.

memAlign implementation is always similar or slower. Finally, each tracker seems more compatible with one implementation but not the other. Indeed, the *score-P* produces significant outliers (shaded curves) on the *memAlign* implementation while the custom tracker produces outliers on the *transposedM0* implementation[4].

Our approach to deal with the limitations of the state-of-the-art performance trackers is to adapt the used tracker to both host hardware and target application. We perform the adaptation to the different hardware platforms using our custom extension of the *perfmon2* library. The basic implementation of this open source library allows an easy and native adaptation to different hardware platforms (assuming the presence of a UNIX kernel OS). In our implementation, we have extended this library by allowing to choose among different tracking algorithm for a given hardware (such as the one partially presented in Figure 2.5). However, it is noteworthy that all the hardware platforms that we consider on this thesis have no software support for the perfmon2 library (such the *Kalray Coolidge* MPPA processor). In this context, our extension allows to easily replace the OS and system-related functions of the library with our calls to the API of the PMU of the considered hardware.
Meanwhile, for a given target kernel, we choose the performance-tracking algorithm to use based on the approach of Heiko Koziolek [54]. In this work, the author propose a classification of software kernels based on different parameters (e.g. memory footprint, pattern to access memory, family of PMU). In our implementation of the *perfmon2* library, we use these parameters to select the proper tracking algorithm within the classification of Heiko Koziolek. Most of the parameters used for our

[4] We assume that this points are outliers given that our experimental protocol on the considered hardware/software environment. All the data caches are flushed between successive execution and the processor frequency is kept constant. Thus, the execution time should not significantly vary for different executions of the same code with the same input.

selection (such as the memory footprint and the pattern followed to access memory) are automatically found by running a first execution of the target code at compile time.

2.4 Conclusion: Adopted method and implementation choices

In this thesis, we have adopted a scientific methodology based on our exploration of the state-of-the-art simulation tools, hardware-performance models and performance tracker utilities.

- We consider the three families of hardware platforms presented in section 1.6. These platforms allow to evaluate our contributions on an HPC, general-purpose and embedded hardware. We have also chosen to only consider UNIX operating systems.
- The main optimization phase that we perform is executed at compile time. This might necessitate to run the targeted application. Given the important time limitations of modern hardware simulators (at compile time), we have restricted our compile-time evaluations to native execution. Even though this limitation forbids us to have more exhaustive exploration (such as large frequency scaling, or page-replacement policy) we consider it as a fairly interesting trade-off.
- All the performance evaluations are withdrawn from our extension of the *Perfmon2* library. We design it as a unified API with different back-ends for each targeted hardware.
- Our optimization process results in generating a transformed source code. This code contains static transformation. It also contains dynamic user-level transformations (such as the choice of data-layout implementation) to be executed based on run-time parameters. Our optimization process also generates a pre-execution binary. This binary sets the hardware and OS parameters using the UNIX *LD_PRELOAD* environment variable and the *cpufreq-set* API of the *stdlib* standard library.

Chapter 3

State of the Art: Software Optimization

3.1 Overview in Software Optimization **33**

3.2 Source-Code Adaptation to Hardware **34**

3.2.1 Methodologies for Code Adaptation 34

3.2.2 The *LGen* Code Generator for Basic Linear Algebra Computation . 35

3.3 Data-Layout-Based Software Optimization **37**

3.3.1 Background: Data-Cache-Miss and Performance 37

3.3.2 Data Layout: Definition and terminology 38

3.3.3 Dynamic-Memory Access Pattern 38

3.3.4 The Data-Layout Decision Problem 40

3.4 Scratchpad memories **42**

3.4.1 Memory Overview 42

3.4.2 High-Performance-Computing-Code Optimization 43

3.4.3 Embedded-Code Optimization 43

In this chapter, we first sketch in section 3.1 the state of the art of software optimization approaches. We use this bird-eye-view of the domain in order to motivate the need for a new approach. Then in section 3.2 we focus on the software optimizations that adapt a given source code to the specificity and the strength of the host hardware. The objective beeing to highlight the hardware and software portability issue of this particular approach. Finally, we present in section 3.3 what we consider to be a major leverage-point for software-optimization based on hardware-adaptation: the data-layout implementation.

3.1 Overview in Software Optimization

In computer science, a particular thought has for long been given to improve the performance of the produced code. Consequently, today's literature gathers huge amount of divers, advanced researches and attempts for software optimization. The most accomplished solutions range from *just-in-time compilation* [70] to *polyhedral compilation* [11] and *source-to-source transformation* [24]. In Table 3.1, we represent some of these solutions based on their flagship implementation. We note from this table that the ecosystem of software optimization is highly detached from hardware consideration. No particular attention is given to the automatic adaptation to the specificity of the host hardware. Given that modern hardware are often specifically

designed for performance purpose, we consider this lack of hardware-awareness in software optimization as a major limitation.

	Algorithm	Source Code	Compilation (static)	Compilation (dynamic)	Post mortem
FFTW [35]	✓	✓	✓		
JIT [70]			✓	✓	✓
Magma [104]		✓	✓		
deGoal [18]		✓	✓	✓	
LGen [102]	✓	✓	✓		
BOAST [24]	✓	✓	✓		
Scalasca [36]		✓	✓		✓
Perfmon2 [32]		✓	✓	✓	
MAQAO [27]		✓	✓		
Polyhedral compilation [11]		✓	✓		

Table 3.1: Classification of existing code-optimization solution according to their leverage-point on source code

3.2 Source-Code Adaptation to Hardware

3.2.1 Methodologies for Code Adaptation

Code adaptation (or specialization) to a specific host hardware is a branch of software optimization. In consists in detecting the properties of a given hardware that suit the requirements of a given source code in terms of computation, memory or synchronization. We classify the existing approaches for code adaptation within two categories. First category is by rewriting the code using the specific API and framework of the host hardware. This probably the most explored and accomplished one [110, 6, 20, 104, 29, 34, 68] and is often implemented for most important and performance critical algorithms.

This consists in optimizing a given source code to a specific processor, memory and accelerators. It usually involve modification in the initial algorithm in order to reorganize the data (e.g. SIMD processors, GPU accelerators adapting the data to the number and size of data caches). Moreover, it is often achieved manually through libraries that maximize the use of fast and low-overhead instructions. One of the most known examples are the *MKL* [110] [1] for the *Intel* architectures. Other libraries have been proposed for different architectures: *MAGMA* [104] for hybrid CPUs-*Nvidia* GPUs, *PLASMA* [29] for multinode (OpenMP) *IBM Power8*

[1] Based on the *LAPACK* [6] and *ScaLAPACK* [20] projects.

architectures and *Julia-TensorFlow* [34] (with the XLA compiler) for *Google* TPUs. All these researches have shown very interesting performance-improvement results for different algorithms and implementations. However, the main issue is their lack of portability to new hardware and software platforms. Given that such codes have been manually optimized[2] for a specific hardware/software platform, it may hardly be ported to any different one. Similarly, any modification of the algorithm or the implementation requires an important time and engineering effort to reach sufficient optimization level.

The second family of techniques for code-adaptation to the hardware is based on a performance-evaluation of the host hardware. This approach has been mainly studied for energy efficiency purposes [51, 25, 42]. However, it has been barely-explored for execution-time improvement. This lack of consideration maybe deduced for a survey of a crucial tool for such an approach: the hardware-performance models. Most existing models, are hardly exploitable in automatic software optimization. The example of the *Roofline* model [112] (section 2.2) is representative of this case. Indeed, this model consists in a couple of constant threshold values that the performance of a kernel may not exceed. Thus, it cannot be used to spot the hardware parameters (such as the cache sizes, associativity or replacement policy) that have the most influence on the kernel's performance.

3.2.2 The *LGen* Code Generator for Basic Linear Algebra Computation

The computation (linear algebra) of matrices is a compulsory service for numerous scientific problems. Hence the need to have an efficient framework for such operations. In this context, optimizing a matrices-computation comes to reducing the number of idle/redundant computations (scalar). It also comes to adapt the corresponding code to the host hardware (ex: group the scalar computation for SIMD architectures or use matrix-related ISA instructions).

To the best of our knowledge, the *LGen* [102] code generator is the closest work to the optimization approach that we propose. It consists of a compiler that generates an optimized *C/C++* code corresponding to an input Basic Linear Algebra Computation (BLAC). For a given BLAC, the *LGen* first splits the input matrices into blocks (tiles) with respect to a fixed granularity. The resultant Linear Language (LL) expression is transformed into a $\Sigma-$LL expression (made of sums and products of tiles) using the custom-defined *gather* and *scatter* matrix-operators. Thanks to the associative, commutative and distributive properties of these operators, the resultant expression is simplified in terms of number of accesses and indexes on the matrices. A C code is then generated by mapping each element[3]

[2] This optimization also encompasses the corresponding compiler in the case of the XLA for *Google* TPUs

[3] Sum, BLAC, gather and scatter operators

of the $\Sigma-$LL expression to a predefined code pattern.
This whole process is repeated and the performances of each generated code are assessed to pick the best variant.

On one hand, the interest of the generated C code is its efficiency on dense matrices (matrices with few zero-blocks and no particular geometrical property). This efficiency-goal targets both small and large scale matrices. Moreover, the code generated by *LGen* might be adapted to vectorized hardware architectures (by adapting the granularity of the initially-generated tiles to the hardware-vector size). On the other hand, *LGen* might lead to process redundant/idle computations when the matrix has spares regions (zero-blocks) or has some symmetry/geometrical properties.

In order to tackle the previous limitations, [102] proposes to first identify a set of structured input-matrices to be split into regions according to their specific properties (e.g. zero-block, or symmetric block). The previously-defined $\Sigma-$LL expression is then generated using $\Sigma-$*CLooG*. This loop generator transforms the input BLAC (once tilled) into a set of *CLooG* statements. Each statement specifies the indexes (domain) used to scan a region of a matrix and the order (schedule) to use these indexes. $\Sigma-$*CLooG* associates each statement to a specific $\Sigma-$LL expression. Finally, thanks to *set-theory* operations, this statements are processed to generate a single statement (or a linear combination of statements). The associated $\Sigma-$LL expression represents an optimized C code of the input BLAC. It might might then be generated using *LGen*.

The generated $\Sigma-$LL expression (and the corresponding C code) has now no redundant computations[4]. Furthermore, unlike other structured-matrices library (such as *MKL* or *MAGMA*), the *LGen* may easily handle new user-defined structured matrices.

One of the main limitation of the *LGen* compiler is its lack of scalability to new implementations of a given structured[5] matrix. Indeed, *LGen* allows to define new structured matrices. Our claim through this thesis, is that the implementation of a given structured matrix must be adapted to the pattern followed to access it. However unlike for the code generated with our proposed compiler (chapter 4), the implementation of *LGen* structured matrices is unique regardless of the pattern followed to access them. It is also unchanged with regards to the host hardware memory.
Meanwhile, the method proposed in *LGen* focuses on computation restructuring. Such a code-modification is known to have a deep impact on cache behavior and eventually on performance [57]. No thought is given to the impact of the proposed

[4]Computation on zero-blocks or multiple computations on symmetrical blocks
[5]Such as symmetric, triangular or o-spared-filled [98]

reconstructions on the data layout.

3.3 Data-Layout-Based Software Optimization

In the software-optimization approach that we propose, one of the most relevant parameters for performance are the one related to the implementation of the given algorithm. In literature, the impact of code-implementation refactoring on performance has for long been proven. This has lead to numerous computational paradigms ranging from computational parallelism up to instruction re-ordering and software-level branch prediction. This has also led to the numerous optimizations options embedded within most general purpose compilers. In this section, we focus on generating optimized implementation of the data structure instantiated by the different variables of the considered code. The objective is to be able to integrate a tool that allows to automatically propose an optimized data-layout implementation for a given input kernel. The found implementation needs also to fit the underlaying hardware constraints.

3.3.1 Background: Data-Cache-Miss and Performance

When a memory-access triggers a cache miss, the time for accessing the data my be multiplied by an order of magnitude up to 10^3 times on modern L3 caches [61]. Such an access may thus be even slower than a direct access to the main memory. The number of cache-misses relative to a code's execution is fundamentally conditioned by the memory-access pattern of the application. In Figure 3.1, we show that the access to an address a within the heap of a process leads to populate a line of the cache with the data located at addresses ranging from $a - a[L]$ to $a + L - (a[L] + 1)$. This memory access is relatively costly as it triggers an access to the slow main memory. However, this cost may be hidden if an important amount of close consecutive accesses are realized in the range of addresses between $a - a[L]$ and $a + L - (a[L] + 1)$. Maximizing the usage of this prefetched memory is the corner stone of most data-layout-based software optimizations.

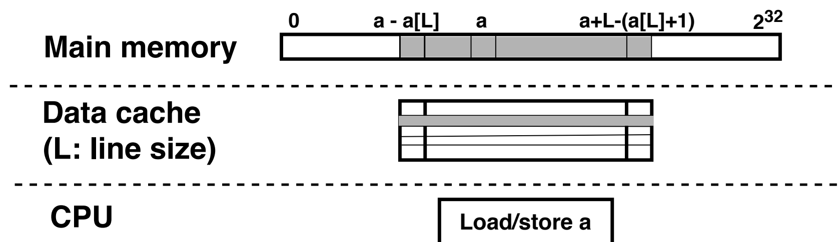


Figure 3.1: Cache-fetch behavior while accessing a data at an address a. $a[L]$ represents the rest of the euclidean division of a by L.

At the scale of a memory accesses, the impact of cache misses may be soften by

some hardware components. The memory prefetcher may for instance detect the memory-access pattern that is currently realized and fetch the data to cache before they are required by the processor. This is mainly used when the memory-access pattern includes data that are not in the range of addresses fetched to cache at each access. However, when the whole pattern does not fit the previously-described cache-behavior, the prefetcher's pipeline may be full with pending requests. The significant time to process these requests makes the prefetcher unable to reduce the impact of cache misses on the performance. Then, the number of LLC data-cache-misses becomes predominant with regards to the overall code-performance.
In this thesis, we identify such kernels where a high correlation may be observed between the data-cache misses and the execution time. In this case, we use the total number of data-cache-misses as a guidance and validation function for the optimizations that we propose. This function relatively easy to model and estimate compared to the execution time.

3.3.2 Data Layout: Definition and terminology

Define	int *matrix;
Allocate	matrix = malloc (N*N * sizeof(int));
Set	matrix[x + N*y] = value;
Get	int v = matrix[x + N*y];
Free	free(matrix);

Table 3.2: Example of *C/C++* routines defining a uni-dimensional implementation of a 2D matrix of size N*N

In this thesis, we refer to *data layout* (or *data structure*) a family of potential geometrical organizations of the user data within virtual-memory space. We refer to *data-layout implementation* (or *data-structure implementation*) the set of routines used to define, allocate, access (read, write and compute one or a set of elements) and free the data-layout. In Table 3.2, we present an example of such an implementation for a 2D matrix. It is noteworthy that an optimized data-layout implementation may bring significant time and energy improvement. Indeed, an adequate data-locality (with regards to caches, prefetchers or any memory accelerator) is a well known solution for many performance bottlenecks [57, 68, 5, 113, 71, 38, 107]. However, as shown in Figure 3.2, the research space of these implementations is relatively large. Moreover, the performance of each one of the corresponding code may dramatically vary from one implementation to an other [57]. Consequently, finding an adequate implementation for a given kernel within the constraints of the host hardware is a complex yet essential task.

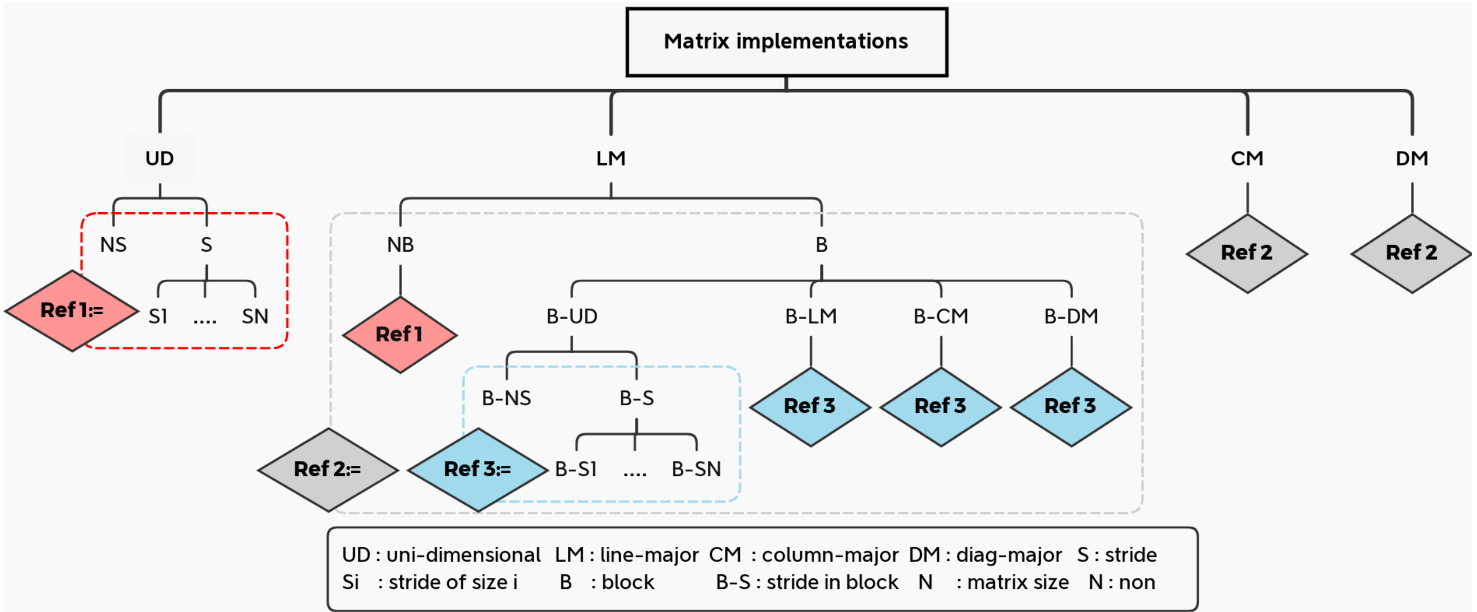
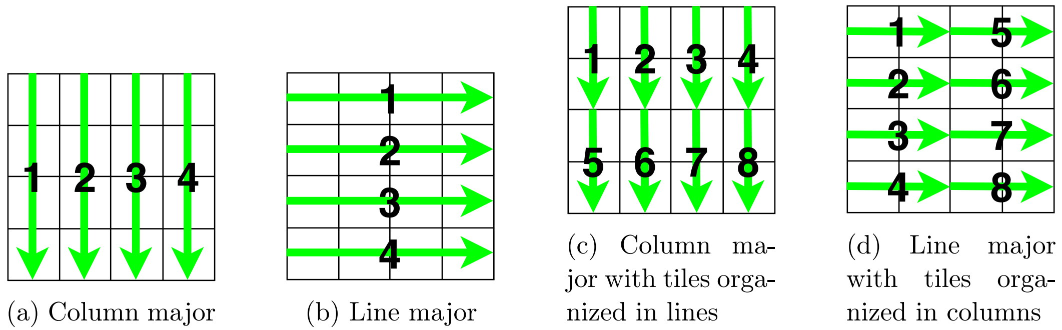


Figure 3.2: Set of $O(N!)$ potential implementations for a two dimensional matrix.



(a) Column major (b) Line major (c) Column major with tiles organized in lines (d) Line major with tiles organized in columns

Figure 3.3: Example of four memory-access patterns to a matrix data structure.

3.3.3 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. In Figure 3.3, we have represented four examples of memory-access pattern to a matrix data-layout. Many works have evaluated the link between a memory-access pattern and the performance regarding modern hardware-memory hierarchies (at least one level of fast memory between CPU and main memory) [57, 72, 19, 45, 78, 95]. To the best of our knowledge, no approach has been proposed to automatically link, for different data structures, a memory-access pattern with an optimal implementation of the data structure. This limitation represents one of the basic contributions of our approach.

The memory-access detection is intensively used by hardware prefetchers. The

algorithms used are usually kept confidential to the manufacturers. Nevertheless, one of the most intensively studied prefetching methods for general-purpose processors is the *"one block lookahead"* [99]. This method is based on the principle of the *stream buffer*, a frame of memory-addresses accessed consecutively and assumed to slide with a fixed time or access rate. Thus, this pattern-detection method works at the scale of few addresses. It may hardly be used to build an access scheme for a whole data structure.

Cache-prefetchers maybe significantly affected by dynamic-allocator behavior. Unlike other CPU prefetchers, cache prefetchers deal with virtual addresses. Most general-purpose allocators (such as *ptmalloc*, the default *Linux GLIBC* implementation [39]) introduce non-payload addresses at random locations in between blocks of the same data structure for dynamic-memory-management purpose (to reduce internal fragmentation caused by the variable *basic-block* size). This additional addresses introduce randomness in the pattern, making its detection harder.

To the best of our knowledge, the closest method for memory-pattern detection similar to our work is proposed in [115]. This method is used to build the *memory signature* (subsection 4.2.2) of an application in order to detect malware-injection. Unlike the signatures that our method generates, the ones generated in [115] 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 the method in [115] to be used for software optimization.

3.3.4 The Data-Layout Decision Problem

The compiler-driven software optimization has for long consisted in determining an optimal set and order of instructions within an input source code [60, 73, 35, 102, 11]. Since the hardware-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 software optimization is thus related to solving this DLD. As most solutions for the DLD at compiler scale, we assume that all possible loop-transformations and instructions-shuffling are already performed.

Two families of strategy have been proposed to tackle the DLD. Based on a previously observed memory-footprint, the authors in [97, 63, 72] propose to statically determine an optimized memory-placement at compile-time. Li et al. [63] introduce a general purpose compiler approach which 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

order to tackle such an over-fitting issue, Shoushtari et al. [78] propose to divide the considered memory (scratchpad) in clusters. The problem of populating each cluster is then reformulate as an integer-linear programming problem. Such greedy 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.
Meanwhile, the authors in [46, 19] propose a dynamic approach. It consists in finding, at run-time, the proper placement of memory with regard to the previous memory-accesses of the current execution. Kandemir et al. [46] considers dynamic loop-transformations to inject data-fetch instructions for scratchpad memories. Cho et all. [19] propose a heuristic function to decide which data-copy to process after each fixed number of memory accesses.

A major limitation of the existing solutions for solving the DLD is their lack of portability to new hardware or software platforms. Indeed, one part of these solutions focus explicitly on a specific access type (the approaches presented in [97, 22] consider only regular accesses to memory while those presented in [63, 108] consider irregular accesses). An other part focus explicitly on a specific hardware-memory hierarchy (the approach presented in [19] consider only scratchpad memories while those presented in [47, 45, 72] consider only multiprocessor system on chips with some specific direct memory access). To the best of our knowledge, our approach is the first one to port data-structure implementations, optimized for a specific hardware or kernel, on a broad spectrum of applications and memory hierarchies.

A second limitation of the solutions described above is their granularity. All of these approaches allow to optimize the placement of data that are restricted to simple scalar variables or contiguous blocks of memory. Unlike these approaches, our work considers the whole data layout (potentially multi-dimensional blocks of memory). This avoids prioritizing memory blocks that are performance-critical during the optimization process but not at run-time.

An optimized solution to the DLD is likely to downgrade the efficiency of the optimization found using instruction-reordering (e.g. -01, -02 and -03 options of the *gcc* compiler). Indeed, each one of the instruction and data-layout-based families of optimization alters the ground assertion of the other family. The instruction-reordering optimizations assume a fixed data-layout organization while an optimization based on data-layout reordering assumes a fixed instruction ordering. Given the algorithmic complexity of both families of solutions, introducing this dependency within any one of this solution would be time consuming. In this context, we believe that our analytical approach is an interesting alternative to deal with this complexity. It allows to simultaneously perform a research-space exploration while considering

both instruction and data-layout-based optimizations.

3.4 Scratchpad memories

3.4.1 Memory Overview

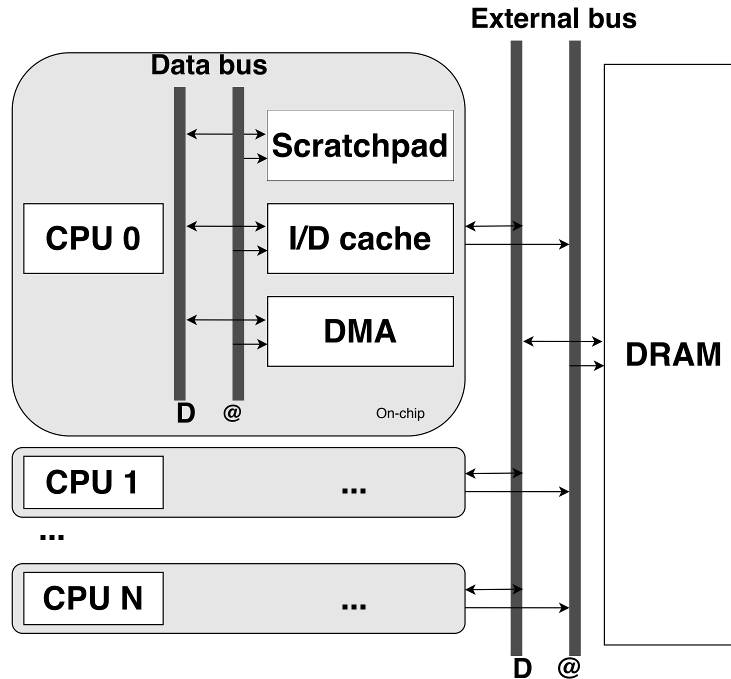


Figure 3.4: Hardware view of a memory hierarchy including an on-chip scratchpad memory.

A scratchpad memory is a fast memory designed as a programmable alternative or complement to the caches [10]. It is often considered similar to the L1 data cache in that it is usually the closest one (on-chip) to the Arithmetic Logic Unit (ALU) after the processor registers. It is also exclusively accessed through explicit instructions to move data to and from main memory. As shown in Figure 3.4, these data-transfers are often achieved using DMA systems. In this paper, we do not consider scratchpad registers which are successfully addressed by programmers using regular register-placement algorithms.

The scratchpad memories are mostly dedicated to storing temporary results that would otherwise be set in the process stack with a higher latency. Scratchpad are usually preferred to caches in real-time applications due to their higher time-predictability. However, with the increasing size of scratchpads[6], larger and more complex data layouts may be hosted within this memory. Consequently, many

[6]From few kilo bytes in the early twenties up to few mega bytes nowadays.

studies have focused on solving the DLD problem, as well as its different sub problems, on scratchpad memories. Similarly, the different access-granularities proposed by modern DMAs (e.g. rectangular or strided) has allowed considering scratchpad memories for a wider spectrum of applications with a higher memory space requirement.

3.4.2 High-Performance-Computing-Code Optimization

As they are fed by the DMA, the scratchpad memories may be used in place of a cache for mirroring the state of data or instruction sections in slower main memory. Such an application of scratchpad memories is particularly interesting for super-calculators [93, 121, 96, 80]. In [93], the authors propose a dynamic approach based on a compile-time observation of static variables in order to determine the bits that are absolutely necessary for a given computation. The selected bits are then the only one to be transfered through the costly memory hierarchy. The others being stored within the more energy-efficient scratchpad memory. In addition to its usage for energy-reduction, scratchpads are also used by super-calculators for pure time-efficiency purpose. In [121], the authors propose a custom coherency protocol to use MP-SoC scratchpads as a fast and low-latency message-passing interface between cores. This memory is designed to reduce the number of memory-request misses for *Single Instruction Multiple Data* programming. It is also shown to scale well with the number of concurrent cores. A similar approach is developed by [96] and integrated to the OpenMP language.

To the best of our knowledge, the closest scratchpad-memory-placement approach to our work is developed by Peña et al. [80]. Unlike other approaches, the data-oriented profiler developed in this work identifies the followed memory-access pattern in order to decide the corresponding optimized memory in a heterogeneous set (including but not limited to scratchpads). Such a solution allows to find an interesting data-placement within the secondary memory. However, no taught is given to the resulting extra requests on the main memory nor how this may change the previously optimized data-locality in the main memory. Indeed, despite the usage of a secondary memory (such as a scratchpad) the main memory is still widely solicited. Thus, we believe that an ideal data-layout is the one that optimizes data placement within both main and secondary memories.

3.4.3 Embedded-Code Optimization

Scratchpad memories may be useful for real-time applications, where predictable timing is hindered by cache behavior. Despite a relatively lower time and energy efficiency, regular caches continue to be preferred to scratchpad memories in software industry [94]. This is mainly due to the relatively complex programability of scratchpads. To tackle this issue, different studies have focused on automatizing the placement and management of data-layout within scratchpads. In [94],

the authors split the corresponding solutions in static, automatic and dynamic solutions. The static approaches [19, 7] consider storing static variables and other data and instructions section of a given process that are fully known at compile time. Even though static solutions may be run with a polynomial complexity and low overhead (at execution time), they may create a significant contention on the memories (both caches and scratchpads). Such contention has been tackled in [109] at the expense of complexity. Similarly, automatic approaches [100, 117, 48] extend the previous solutions to the whole process address-space. A particular attention is given in [48] to the function frames for recursive or deeply-interleaved function calls (known to create an important memory-footprint). Meanwhile, dynamic approaches [28, 80, 43, 103, 4] focus on storing and managing the heap section of a process within scratchpad.

In the context of the hierarchy defined in [94], our approach is the only hybrid one. Indeed, our approach is the only one that defines the data-layout structure at compile time and inject the code to re-scale the structure at run-time. Thus, given that the main part of the work is done at compile time, the overhead of our optimization is consequently reduced compared to other run-time approaches. However, unlike other works, we make the assumption that the considered code keeps the same execution flow regardless of the input values (conditional branches do not depend on the input of the program). This soft assumption allows us to identify at compile time the memory-access pattern followed at run-time. In addition, our work does not consider storing instructions in scratchpad. This allows us to perform a locality-based analysis regardless of the instruction dependencies. Consequently, our analysis is performed in a polynomial time while the one proposed by the state of the art are exponential (in the worst case).

Part II

Code Optimization by Adaptation to the Hardware Memories

Chapter 4

HARDSI: Custom Method to Solve the Data-Layout Decision problem

4.1 HARDSI Overview . . . 48
- 4.1.1 Objectives . . . 48
- 4.1.2 The HARDSI DSL . . . 48

4.2 Global Optimization Process . . . 50
- 4.2.1 Memory-Access Tracking . . . 51
- 4.2.2 Generating a Memory-Signature . . . 51
- 4.2.3 Access-Pattern Data-Base . . . 52
- 4.2.4 Software Optimization . . . 54

4.3 HARDSI Framework Implementation . . . 55
- 4.3.1 Accelerating the HARDSI Process . . . 55
- 4.3.2 Conventional Three Cache-Levels Architectures . . . 57
- 4.3.3 Pluri-architectural Software Optimization . . . 58

4.4 Contemplated Future Implementations . . . 60
- 4.4.1 Multicore architecture . . . 60
- 4.4.2 In-memory Computing . . . 60
- 4.4.3 Detecting Malicious-Code Injection . . . 62

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. We thus believe that reducing the memory-fetch time by improving data and cache locality may bring an important gain. However, finding an efficient data-layout placement among all the possible implementations is known to be an NP-complete problem [55]. Moreover, a given solution to this problem is only efficient on a specific set of hardware memories. Porting such solutions across several hardware platforms requires important time and engineering effort.

In this chapter, we introduce HARDSI a custom patented source-to-source transformation method for software optimization by adaptation to the specificity of the host hardware. Our compiler is designed as the part of our optimization approach (see section 1.3) in charge of exploring and choosing an optimized data-layout implementation for each variable of a given kernel running on a given host hardware. We also show how HARDSI might be used as a lightweight and standalone code optimizer and code adapter to different families of hardware memory-hierarchies.

4.1 HARDSI Overview

4.1.1 Objectives

HARDSI stands for Hardware-Adapted Refactoring of Data Structure Implementation. We design it as a custom method allowing to detect, for each supported data-structure, an optimized implementation with regards to

(i). The access to the data-structure on each variable of the code.

(ii). The family of hardware memory hosting the data-structure.

Thanks to the HARDSI framework, we are able to adapt the implementation of a given data structure to the functional requirements of each host hardware. We are also able to easily, efficiently and without any human intervention to port a source code to a new family of hardware memory. Finally, our HARDSI method is designed to take advantage of data-placement optimizations realized within a given software context and port it to a new and unrelated input code.

4.1.2 The HARDSI DSL

In order to ease the usage of the HARDSI method, we introduce a custom HARDSI DSL. An instance of this DSL is shown in Figure 4.1. The proposed language allows the programmer to write a *C/C++* code while sparing him the performance-critical choice of an adequate implementation for the data-structures implementation. If we consider the example in Listing 4.1, the code includes three variables referring to three different matrices. Using our DSL, the programmer may replace all the code routines referring to each matrix variable with our custom primitive-keywords.

- **MATRIX_DEFINE**: declares a matrix with given name m and type. For instance, for a matrix storing integers, our compiler may choose to replace this keyword with "int *m" for a unidimensional implementation, "int **m" for a two dimensional implementation, "int ***m" or "int ****m" for a tilled two dimensional implementation (with tiles of dimension one or two).
- **MATRIX_ALLOCATE**: dynamically allocates all the required memory space.
- **MATRIX_GET**: reads value at the given position of the matrix.
- **MATRIX_SET**: sets the given input value to the given position of the matrix.
- **MATRIX_ADD**: adds the given input value to the one at the given position of the matrix.
- **MATRIX_FREE**: frees all the dynamic memory of the matrix.

More generally, for each considered data structure d, we define the same set of primitives D_DEFINE, D_ALLOCATE, D_GET, D_SET and D_FREE. The developed source-to-source compiler replaces these primitives for each variable with the specific code corresponding to an optimized implementation of the data structure d. The choice of this implementation is processed at compile time through the HARDSI method.

```
1  void matrixMult() {
2    MATRIX_DEFINE(int, a);
3    MATRIX_DEFINE(int, b);
4    MATRIX_DEFINE(int, res);
5
6
7    int i,j,k, a0, b0;
8    MATRIX_ALLOCATE(int, N0,N1,a);
9
10
11   MATRIX_ALLOCATE(int, N2,N0,b);
12
13
14   MATRIX_ALLOCATE(int, N2,N1,res);
15
16
17   for (j=0; j<N1; j++)
18    for (i=0; i<N2; i++)
19     for (k=0; k<N0; k++)
20     {
21        a0=MATRIX_GET(a,k,j);
22        b0=MATRIX_GET(b,i,k);
23        MATRIX_ADD(res,i,j,a0*b0);
24     }
25   MATRIX_FREE(a,N0,N1, int);
26
27
28   MATRIX_FREE(b,N2,N0, int);
29
30
31   MATRIX_FREE(res,N2,N1, int);
32
33
34 }
```

Listing (4.1) Input code implemented using our custom HARDSI DSL.

```
1  void matrixMult(){
2    int **a;
3    int **b;
4    int **res;
5    int s =sizeof(int);
6    int sp=sizeof(int*);
7    int i,j,k, a0, b0;
8    a = (int**)malloc(N1*sp);
9    for (i=0; i<N1; i++)
10     a[i]=(int*)malloc(N0*s);
11   b = (int**)malloc(N2*sp);
12   for (i=0; i<N2; i++)
13     b[i]=(int*)malloc(N0*s);
14   res = (int**)malloc(N1*sp);
15   for (i=0; i<N1; i++)
16     res[i]=(int*)malloc(N2*s)
17   for (j=0; j<N1; j++)
18     for (i=0; i<N2; i++)
19       for (k=0; k<N0; k++)
20       {
21          a0=a[j][k];
22          b0=b[i][k];
23          res[j][i]+=a0*b0;
24       }
25   for (i=0; i<N0; i++)
26     free(a[i]);
27   free(a);
28   for (i=0; i<N2; i++)
29     free(b[i]);
30   free(b);
31   for (i=0; i<N2; i++)
32    free(res[i]);
33   free(res);
34 }
```

Listing (4.2) Corresponding $C/C{++}$ code generated by our compiler for a 3-level cache.

Figure 4.1: Matrix multiplication test case.

4.2 Global Optimization Process

In this section, we introduce the proposed optimization approach designed to automatically converge toward an optimized data-layout implementation for each variable in a given code. This implementation-selection is realized without altering the input algorithm (*i.e.*, control flow). The objective of the proposed HARDSI code is to select, for the considered data structures, an optimized implementation that fits the followed access pattern. Our claim through this thesis is that the specific memory-pattern followed to access a given data-layout is an identifier of the optimized implementation to use. In this context, we do not allow changing the memory-access pattern even though the adaptation of the algorithm could bring an additional gain (combining memory and instruction-related optimizations).

In Figure 4.2, 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 (see input and HARDSI-generated *C/C++* code in Figure 4.1). The objective of the proposed optimization process is to find the optimized implementation of each one of the three matrices given the respective memory-pattern followed to access them.

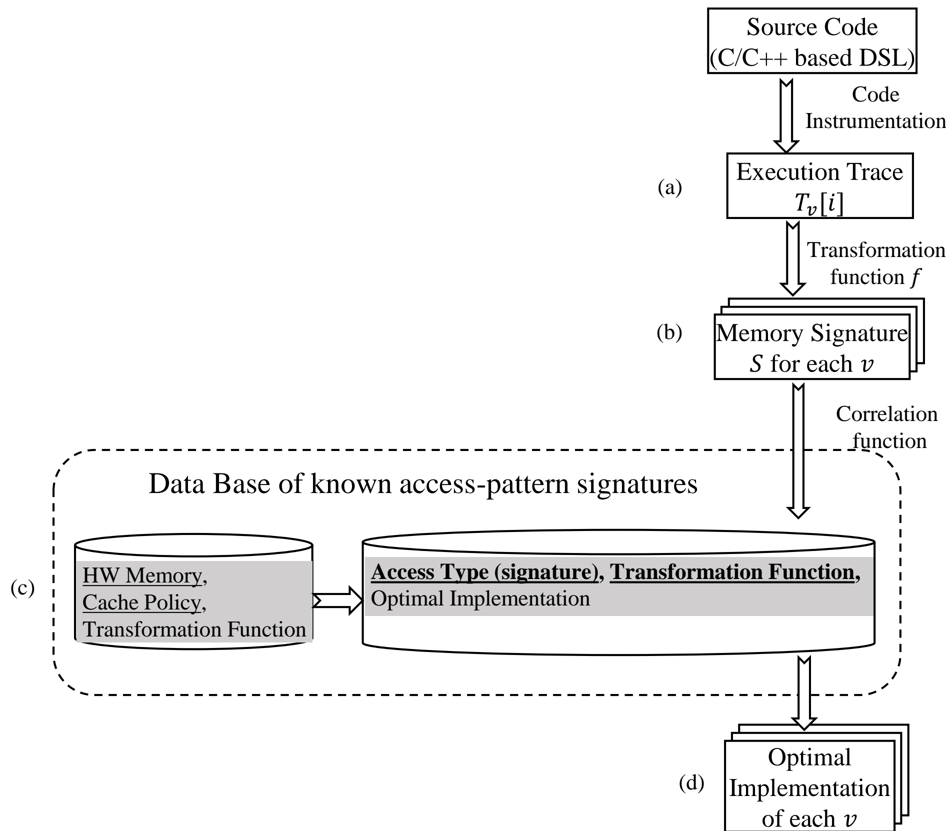


Figure 4.2: Steps of the proposed optimization-process.

4.2.1 Memory-Access Tracking

The first step of the proposed method is to observe the memory-addresses followed to access each considered variable (Figure 4.2.a). We propose to run the targeted execution of the input kernel. In Table 4.1, we present an example of an execution-trace using the considered matrix multiplication. 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, instantiated in Table 4.1, consists, for each variable v, in a set of identifiers $T_v[i]$ of the matrix cells that are accessed. These cells are sorted according to their access rank i. For the sake of clarity, in this thesis, the cells of the matrix are identified in $T_v[i]$ using their spacial indexes (x, y). However, different identifiers may be used such as the virtual or the physical addresses. For this first step, executing the considered code using the targeted hardware memory is only mandatory if we do not use matrix indexes.

Data structure	**Variable Name**	**memory @**	**Access Type**	**Size**	**x**	**y**
MATRIX	res	-	ALLOCATE	4x4	-	-
MATRIX	a	-	ALLOCATE	4x4	-	-
MATRIX	b	-	ALLOCATE	4x4	-	-
...						
MATRIX	a	0x2e010	READ	4x4	0	0
MATRIX	b	0x2e0c0	READ	4x4	0	0
MATRIX	a	0x2e014	READ	4x4	1	0
MATRIX	b	0x2e0c4	READ	4x4	0	1
MATRIX	a	0x2e018	READ	4x4	2	0
MATRIX	b	0x2e0c8	READ	4x4	0	2
MATRIX	a	0x2e01c	READ	4x4	3	0
MATRIX	b	0x2e0cc	READ	4x4	0	3
MATRIX	res	0x2e170	WRITE	4x4	0	0
...						

Table 4.1: Example of an execution-trace (partial) of three matrix-variables a, b and res for a $(4 * 4)$ matrix multiplication test case.

4.2.2 Generating a Memory-Signature

Once we traced the memory-accesses, the second step is to filter this trace through a *transformation function* f (Figure 4.2.b). The objective of the transformation is to remove the randomness introduced by the kernel's execution. Indeed, most general-purpose OSs store data-structures at different virtual addresses from one execution to an other. In this context, a *transformation function* must be designed to make the result of this second step ($T_v^f[i]$) totally predictable and reproducible from one run to an other (assuming that the same kernel's control-flow is observed).

As an example, for the *x86 Intel Xeon* processor with three levels of data-caches, the corresponding transformation used to generate all the optimized code is a δ function

$$T_v^{\delta}[i] = T_v[i] - T_v[i-1], \forall i \in [1, N_{access}]$$

where N_{access} is the number of access to the considered variable. This function is a lightweight computation that encompasses the performance-requirements of the memory hierarchy and its LRU cache-replacement policy. 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 [57].
Adapting our method to different hardware-memories corresponds to finding the adequate transformation function. In chapter 5, we introduce an example of such a function for scratchpad memories. However, automatically generating such a function for each hardware platform is one of the perspectives of this study, and is out of the scope of this thesis.

Finally, we build the memory-signature S of each variable v by generating the occurrence-histogram of each set T_v (Figure 4.3.a). We normalize each histogram by dividing by the total number of occurrences. This makes the generated signatures independent from the kernel's execution-inputs, and representing the absolute memory-behavior of the kernel.

4.2.3 Access-Pattern Data-Base

A survey of data-structure literature [57] shows the abundance of optimized implementations for most standard data structures. These solutions are often designed by programmers to address a specific issue within a given hardware and software context. But the problem is how to extend this knowledge in order to identify the adequate state-of-the-art implementation of a data structure to use within any given code.

The memory signature s, generated in subsection 4.2.2, for a given variable v of a given data structure d, identifies the specific memory access-pattern to v in the considered kernel (Figure 4.3.a). In this section, we assume the existence of a relational data base[1] for each data structure d (such as an N-dimensional matrix or a hash-map). This relation, schemed in the relation (a) of Figure 4.4, links each known pattern of access to d with the best known implementation of d. Such an implementation depends also on the underlying hardware-memory hierarchy; this dependence is identified by the used *transformation-function* and stored in the relation (b) of Figure 4.4.

[1]The usage of a data base indicates that the stored values are static and computed once. Any

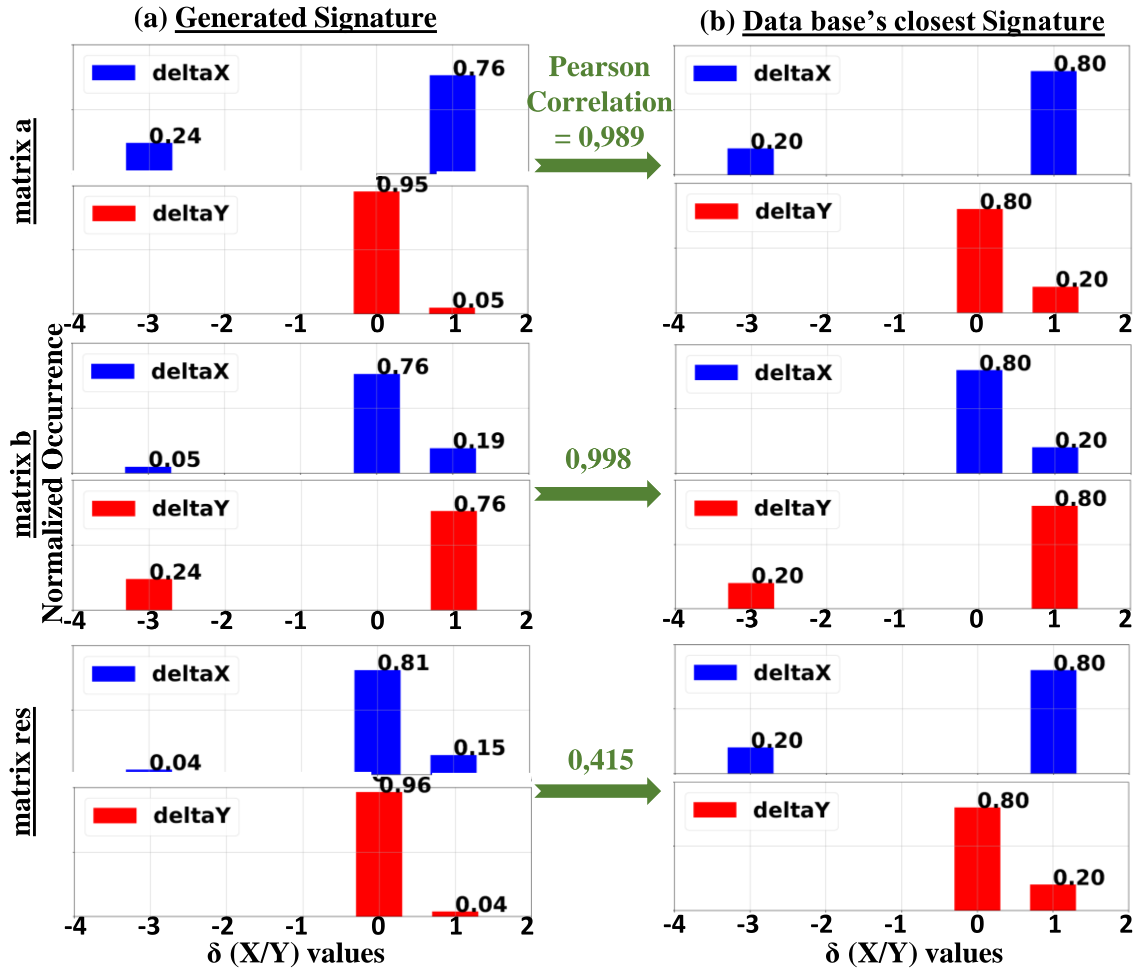


Figure 4.3: (a) Generated memory signatures of the 4x4 matrices a, b and res in the matrix-multiplication test-case; (b) Respective closest signatures in the HARDSI data base.

The relation linking a memory-signature S with the corresponding optimal implementation is a key step for the proposed method (Figure 4.2.c). Thus, we propose a data base built as a survey of the most efficient existing implementations of d on the considered memory hierarchies [11, 46, 62, 116]. Each memory hierarchy is identified by the corresponding *transformation function*. The data base relative to each data-layout d is integrated within the HARDSI framework. In this context, updating the most efficient state-of-the-art implementation regarding a given access pattern may be realized by simply updating an entry of the data-base.

other map-based data layout may be used as.

(a) DATA_STRUCTURE_d (Transformation, Signature, Optimal implem)

(b) TRANSFORMATION (HW memory, Cache policy, Transformation)

Figure 4.4: Relational data base of knowledge: Stores the required relations to retrieve the best known implementation of a given data structure knowing its memory signature. The key attributes (line identifiers) are underlined.

4.2.4 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. Thanks to the relational data base in Figure 4.4, an access pattern (signature s) to a data structure d is an identifier of an optimal implementation of d (along with the used *transformation function*). Thus, finding an optimal implementation of a variable is equivalent to finding the closest signature to s in the data base relative to d (as shown in Figure 4.3). Then, the final step of our HARDSI method consists in injecting the optimal implementation of each variable in the source code of the application kernel (Figure 4.2.d).

In order to compare the generated signature S with each signature S' in the data base, we use the *Pearson* coefficient [40] 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 $\mathbb{E}_S$, σ_S and N are the expectation, standard deviation and number of elements (including some elements with a null-occurrence) of the histogram. The choice of this particular correlation-function is motivated by our three functional needs:

1. Assume that the first signature s is fixed while the second s' is variable. The *Pearson* coefficient converges linearly to zero (no correlation between s and s') when expanding or translating the histogram s' (regarding an horizontal axis). This linear convergence is faster than the one of other correlation coefficients such as the *Kendall tau rank* or the *Spearman's rank* correlation coefficient. This convergence is due to the inverse-dependence of the correlation with the standard deviation $\sigma_{s'}$ of s'. From a computational point of view, this reflects the fact that translating a bin of a histogram is equivalent to changing the set of addresses represented. Hence significantly changing the represented memory-access pattern.

2. The *Pearson* coefficient is relatively tolerant to expansions of s' following the vertical axis [1]. This property is a real asset for our method. Indeed, modifying the number of occurrences of a histogram's bar without shifting it is

equivalent to keeping the same memory-access pattern while amplifying or diminishing a part of it. Thus, the resulting pattern must be relatively similar to the original one.

3. As a consequence of the two previous properties, the *Pearson* coefficient is relatively robust to external noise on s'. In our context, a noise is a set of additional memory access that are legitimate for our code but which do not belong to the pattern represented by s.

It is noteworthy that depending on the considered architecture[2], the *Pearson* coefficient may potentially be adapted. This is for instance the the case in chapter 5 where the memory signatures are 3D histograms. It may even be necessary to replace this coefficient with a more adapted one. However, in the context of the architectures considered in this thesis, such a case has never been matched. Moreover, even if a different correlation coefficient is used, the foundations (Figure 4.2) of the HARDSI method remain unchanged.

4.3 HARDSI Framework Implementation

We implemented the proposed method within a framework that permits to automatically generate the optimized source code of a given kernel. In fact, the user of our framework is required first, to implement the considered kernel in the HARDSI DSL, as shown in the example Listing 4.1. Then, our source-to-source compiler generates the corresponding optimized *C/C++* code, as shown in Listing 4.2. The proposed compiler first computes the memory signatures of each considered variable as shown in Figure 4.3.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 with this closest signature. A narrow subset of the codes associated to each histogram in our data base is presented in Listing 4.3. For the sake of space, we do not show the code relative to the tilled, stencil nor line-permuted (respectively column-permuted) versions of these matrices. A more exhaustive instance of these routines may be found in [56].

4.3.1 Accelerating the HARDSI Process

The optimization process presented in section 4.2 is designed to work with different hardware memories. Each one of these memories having its own set of data-layout implementations. However, if we consider the most popular hardware memories (caches, GPUs, TPUs), the HARDSI optimized solution may be found at the expense of a large research-space exploration (up to $O(N!)$ potential implementation). The complexity of our algorithm may then skyrocket up to $O(N!)$.

[2]Hence, depending on the used transformation function.

```
1  //Matrix per lines: each array cell is a pointer to a line
2  #define LM_DEFINE(type, name)            type **name
3  #define LM_ALLOCATE(type, X, Y, name) ALLOC2D(type, Y, X, name)
4  #define LM_GET(m, x, y)                 m[y][x]
5  #define LM_SET(m, x, y, val)            m[y][x] = val
6  #define LM_FREE(m, X, Y, type)          FREE_2D(m, X, Y, type)
7
8  //Matrix per columns: each array cell is a pointer to a column
9  #define CM_DEFINE(type, name)            type **name
10 #define CM_ALLOCATE(type, X, Y, name) ALLOC2D(type, X, Y, name)
11 #define CM_GET(m, x, y)                 m[x][y]
12 #define CM_SET(m, x, y, val)            m[x][y] = val
13 #define CM_FREE(m, X, Y, type)          FREE_2D(m, X, Y, type)
14
15 //Matrix per diagonals: each array cell is a pointer to a diagonal
16 #define DM_DEFINE(type, name)            type **name
17 #define DM_ALLOCATE(type, X, Y, name) ALLOC2D(type,1+X+Y,1+X+Y, name)
18 #define DM_GET(m, x, y)                 m[DIAG_X(x, y)][DIAG_Y(x, y)]
19 #define DM_SET(m, x, y, val)            m[DIAG_X(x, y)][DIAG_Y(x, y)]=val
20 #define DM_FREE(m, X, Y, type)          FREE_2D(m, 1+X+Y, 1+X+Y, type)
21 #define DIAG_X(x, y)                    (x==y) ? x :              \
22                                         (x>y)  ? y :              \
23                                                  x
24 #define DIAG_Y(x, y)                    (x==y) ? 0          :     \
25                                         (x>y)  ? 1+x+x-y-y :     \
26                                                  y+y-x-x
27
28 //Allocation routine
29 #define ALLOC2D(type, X, Y, name)                         \
30       name = (type **)malloc(X * sizeof(type*));          \
31       for (int i=0; i<(int)X; i++){                       \
32           name[i] = (type*)malloc(Y*sizeof(type));\
33       }
34
35 //Free routine
36 #define FREE_2D(name, D0, D1, type)                       \
37       for (int i=0; i<D0; i++){                           \
38           free(name[i]);                                  \
39       }                                                   \
40       free(name);
```

Listing 4.3: Code representing three different implementation of the 2D matrix data structure (line, column and diagonal-major) with the corresponding primitives for definition, allocation, access and free.

Using the statistical properties of the generated memory signatures, we are able to reduce the HARDSI complexity to roughly $O(Nlog(N))$. Indeed, the data-base relation that gathers all the known implementation (for a given data-layout and within a given hardware architecture) may be seen as a tree-based organization (see Figure 3.2). Optimizing a given data-layout instance is equivalent to exploring all the possible implementations that are presented in Figure 3.2. The algorithm that performs this exploration does not build the tree of potential implementation. It rather explores the potential architectures (line, column, diagonal, 1D, strided) of a matrix. Then the algorithm recursively performs the same exploration on the potential sub-blocks (tiles) of each architecture. However, at each node of this tree-like exploration, a decision may be taken about whether to explore the sub tree or not. This decision is taken with regards to an initial analyze of the memory signature of the considered variable. For instance, in Figure 4.5 we have represented four arbitrarily-picked memory-access patterns along with their signatures. If we consider the Figures 4.5a and 4.5b we may notice that a line-based [3] pattern has a signature with:

(i). A prominent deltaX bar at the value 1.

(ii). A prominent deltaY bar at the value 0

This corresponds to the fact that a matrix cell is most often accessed immediately after its left neighbor. Consequently, an input signature that would not respect such a proportion may not correspond to a line-based pattern. The same principles are applied for each one of the four families of patterns and for each one of their stride versions.

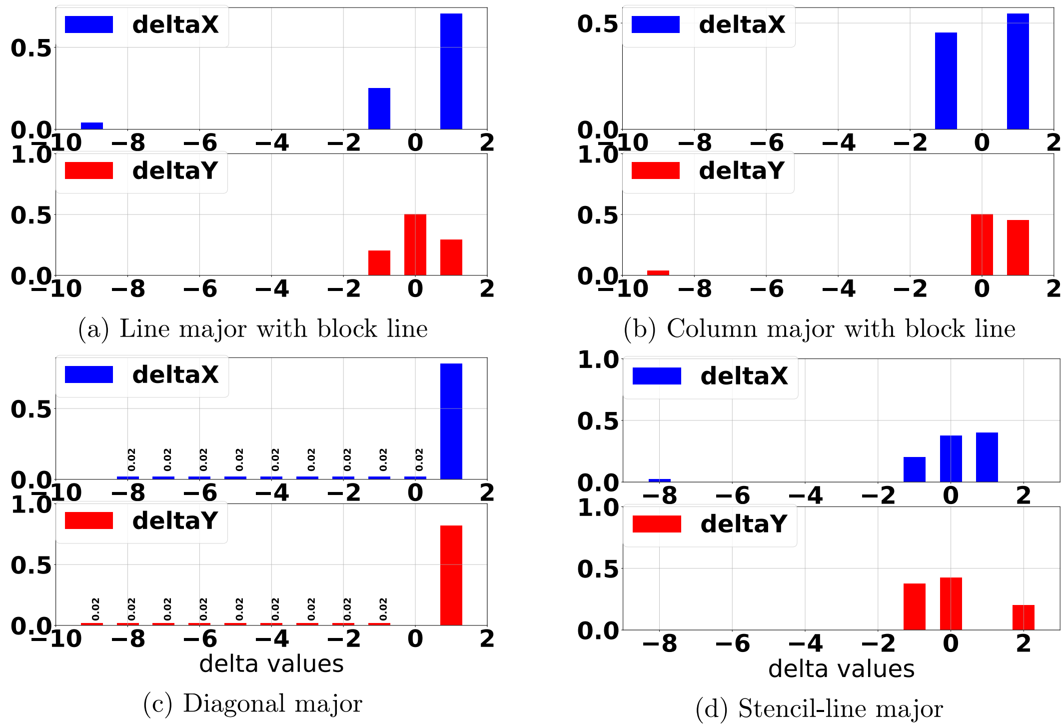
4.3.2 Conventional Three Cache-Levels Architectures

From a programmer perspective, the hardware cache-hierarchy and replacement policy can hardly be changed. Similarly, the access pattern to a given data structure is imposed by the algorithm. The objective of our approach is to find the data-structure's implementation that fits best with both the access pattern and the hardware memory. Consequently, in the context of conventional three-cache-levels architectures, for each address a which is accessed and fetched to the cache, the surrounding addresses[4] should be the one that are the most likely to be referenced again before their eviction.

In order to reach the data-locality required by the conventional memory architectures, we believe that our HARDSI method is a powerful tool. By detecting the memory-access pattern to a given data structure d, our method may find an implementation of d that gathers user-data accordingly. Indeed, let us consider

[3] This encompasses the line-major patterns as well as the stencil and the tilled patterns where each block is accessed in a line major way.

[4] Which are fetched in cache along with a.



(a) Line major with block line

(b) Column major with block line

(c) Diagonal major

(d) Stencil-line major

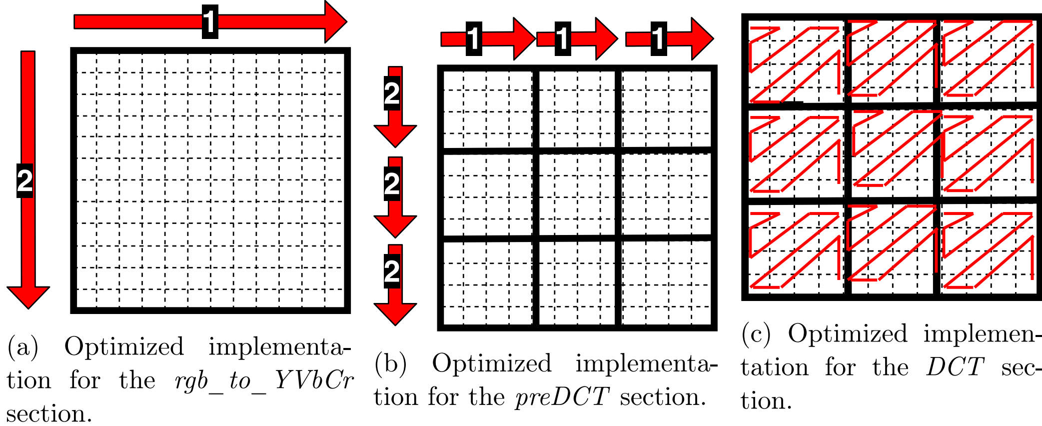
Figure 4.5: Memory-access signatures of a $(10 * 10)$ matrix accessed following four different patterns. The tilled patterns ((a) and (b)) have tiles of size $(2 * 2)$.

the case of the three matrices presented in Listing 4.1. In this example, the first operand matrix a and the result matrix res are both explored following a line-major way while the second operand matrix b is explored following a column major way. These two different types of explorations have been spotted by our method. Therefore, our compiler injects a line-major-based implementation for a and res and a column-major-based implementation for b.

4.3.3 Pluri-architectural Software Optimization

Software-guided memory-access detection is particularly useful for large and complex code sets (high number of nested loops with conditional branches decided at run-time). In this case detecting the followed access pattern to a given data-layout is highly complicated for a human programmer. Hence, the need to our software-guided tool. Moreover our statistical approach suits well with conditional branches that may change the control-flow. These branches may change the memory-access pattern depending on the user inputs. By running our HARDSI method with learning input that cover these different control-flows, our compiler may propose two families of solutions. The first one is to find a data-layout implementation that performs well with all the considered cases (even though a better solution could be found exclusively for a particular case). The second one is to spot the conditional branch(es) responsible of this issue. The programmer may then decide

```
void jpegCompression(rgbColor **imageMatrix)
{
        rgb_to_YVbCr (imageMatrix);
        preDCT       (imageMatrix);
        DCT          (imageMatrix);
        quantization (imageMatrix);
        encoding     (imageMatrix);
}
```



(a) Optimized implementation for the *rgb_to_YVbCr* section.

(b) Optimized implementation for the *preDCT* section.

(c) Optimized implementation for the *DCT* section.

Figure 4.6: Optimized matrix implementation relative to each part of a JPEG-compression algorithm

to split his code accordingly. By using the HARDSI compiler on this new code, an optimized solution is proposed for each created section. Each one of these optimized data-layout implementations may then be easily loaded at run-time by the generated code.

Our method makes it simple to handle a well-known performance critical issue: the case where different patterns are followed to access the same data layout (at different locations of the same code). An example of such a use-case is schemed in Figure 4.6 for the JPEG-compression algorithm [62]. In this code, a same matrix (storing the picture to compress) is successively accessed following different patterns represented in Figure 4.6a, Figure 4.6b and Figure 4.6c. Therefore, each one of these code section is associated with a unique optimized implementation. Moreover, the optimized implementation of one section downgrades significantly the performance of each other section. Thanks to our approach, we are able to first detect the antagonism between these different sections. Then, using our custom performance models [57] we are able to compute the performance gain brought by each optimized implementation of each section. We are also able to evaluate the cost of refactoring the matrix from one implementation to an other. By balancing these two costs, we may decide at the beginning of each section whether to refactor the matrix or not. This decision is taken at run-time as it depends on the size of the matrix (which is not known at compile time).

4.4 Contemplated Future Implementations

4.4.1 Multicore architecture

In the HPC ecosystem, multicore architectures represent an undeniable asset for performance. However, such architectures leave the programmer with an important problem to deal with: the memory placement between concurrent cores. The complexity of this problem increases significantly with the increase of hardware-memory levels (L1, L2, L3 data caches, buffers and scratchpads [10]) due to their different access times and paradigms. Despite the relatively high cost of accessing remote-memory data, these transfers are mandatory for an efficient use of massively multicore platforms.

The HARDSI method can be used to optimize the data placement on a single node's memory. The cost of transferring data may thus be (partially) hidden by the gain brought by an optimized placement of data during its access.
Meanwhile, the HARDSI method includes the different times used to access each considered memory as a parameter of its optimization process (by adapting the *transformation function*). Thus, it can be used to generate an optimized memory-placement among the different CPU-local memories, which reduces the total amount of transferred data.

4.4.2 In-memory Computing

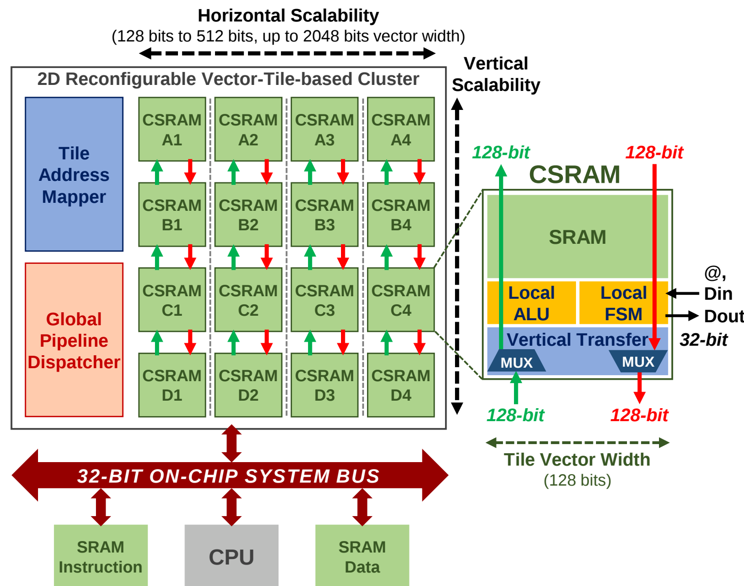


Figure 4.7: Overview of a tiled version of C-SRAM memory architecture (extract from [53]).

The computational SRAM (C-SRAM) [53] (or in-memory computing SRAM) is

an example of heterogeneous memories considered in embedded (low power) HPC. It permits to perform computations directly in or next to the SRAM memory array. This allows to soften the impact of the memory wall by reducing the number of costly data-transfers between the CPU and the main memory.
An inherent problem to the C-SRAM memory is the inner-placement of data. Indeed, as shown in Figure 4.7, the C-SRAM is usually organized in tiles (or blocks) of memory where each tile is split in rows (or vectors). An ALU is shipped within each tile in order to perform computations between two input-rows of the same tile. The computations being processed between aligned-data (in columns) that belong to the same memory tile. In order to take advantage of the C-SRAM, it is mandatory to properly place the data to be computed in order to minimize the number of costly transfers (i) between the main and the C-SRAM memory (ii) within the C-SRAM to align the input data of a computation.

We believe that the HARDSI method can be used to efficiently balance data between the C-SRAM and the main memory. Indeed, this method is able to detect the memory pattern followed by the considered source code. Using the statistical approach presented in section 4.2, the HARDSI method is able to link the considered data-layout with a known implementation that would (i) fit the inputs and order of the computations (ii) maximize the usage of the C-SRAM memory-space. Such a usage of the HARDSI method would be realized in the case where different efficient implementations would exist for C-SRAM usage. Otherwise, we show in chapter 5 how to dynamically generate such data-layout implementations for memories similar to C-SRAM.

In order to identify a C-SRAM-adapted code, the HARDSI method needs to integrate a new constraint during its optimization process. The data fetched to C-SRAM must be aligned according to the computations that will be realized. Unlike the example introduced in subsection 4.2.4, the objective must not be the reduction of data-cache misses.The new objective must be the reduction of the number of memory alignments required before running a computation. In this context the *transformation function* that we propose is

$$T_v^{cs}[i] = \operatorname{abs} f^c(i) - f^c(i-1)$$

where:

- $f^c(i) = @_i^0[L] - @_i^1[L]$
- $(@_i^0, @_i^1)$ are the addresses of the operands in the ith computation.
- L is the size of a line of the C-SRAM and $@_i^0[L]$ is the rest of the euclidean division of $@_i^0$ over L.

This function allows to model (in the generated signature) the number of shifts in the C-SRAM memory (realized to align the data in order to perform each computation).

Such a parameter is, to the best of our knowledge, the most influent on the execution-time of the kernel.

4.4.3 Detecting Malicious-Code Injection

The HARDSI *transformation function* may be adapted to reach other matters than hardware fitting. If we consider three-cache-level architecture, the used *delta function* ensures that the resulting memory-signature (histogram) is fully-reproducible across successive code-executions. Indeed, this function considers the distance between the accessed addresses. Unlike raw virtual addresses, the distance between virtual addresses remains constant at each execution (assuming the same control-flow is followed by the code). Consequently, detecting a variation of a variable's signature between two executions is equivalent to detecting a non legitimate set of memory accesses[5]. Thus, it is possible to use the HARDSI method for malicious and faulty code detection. Our method may for instance replace the learning function introduced by *Zhixing Xu et al.* for machine-learning-based malware detection [115]. Unlike the function proposed by Zhixing Xu et al., the generated memory-signatures do not depend on the dimensions of the considered data structure.

[5] Memory access that has not been intended by the programmer.

Chapter 5

Dynamic Data-Layout Implementation for Programmable Memories

5.1 Extending the HARDSI Framework to Scratchpad Memories . . . 64

5.1.1 Background and Process Overview 64

5.1.2 HARDSI Transformation-Function for Scratchpad Memories 65

5.1.3 Implementation and Multiple-Memories Issue 66

5.2 DDLGS: Generating Matrix Data-Layout Dedicated to Scratchpad 67

5.2.1 Custom Generation Process 67

5.2.2 Determining the Weight of Each Matrix Cell 69

5.2.3 Generalizing the Resulting Implementation to New Input Data . . . 70

5.2.4 Potential Improvements 72

In this thesis, we refer to *programmable memory* any memory that requires to be explicitly addressed by the programmer (or any user-level software). This encompasses scratchpads, GPUs, TPUs and C-SRAM memories. An instance of non-programmable memories are the data-caches as they are implicitly populated by the prefetcher, the DMA or other hardware and system-level mechanisms.

In this chapter, we present our method to extend the HARDSI framework in order to support programmable memories. This extension is entirely presented through the instance of scratchpad memories. We present in section 5.1 the different steps and requirements of this extension of the HARDSI method. Then, in section 5.2, we introduce DDLGS, a custom and patented method designed to generate dynamic and pattern-adapted data-layout implementations for scratchpad usage. The implementations generated by this method embed the load and store functions to be used at each access to an address of the considered structure. These implementations also allow to compute an optimized granularity for each transfer from (respectively to) the scratchpad memory.

5.1 Extending the HARDSI Framework to Scratchpad Memories

5.1.1 Background and Process Overview

From a programmer perspective, a scratchpad memory is a high-speed and low-latency extension of the main memory. As shown in Figure 5.1, accessing a scratchpad may be done through the DMA. A simple way to process such an access at high-level code (e.g. *C/C++*) is by considering the scratchpad as a pre-allocated dynamic-memory chunk (virtual address-space). This chunk may then be used to store data for rapid retrieval using regular pointers or static variables. Even though accessing a scratchpad memory through the DMA bypasses data-caches, such an access has a non-uniform memory-latency. In order to efficiently run a code on scratchpad memory, the objective is to optimize the usage of the relatively narrow scratchpad space. The objective is also to find the ideal granularity for each transfer.

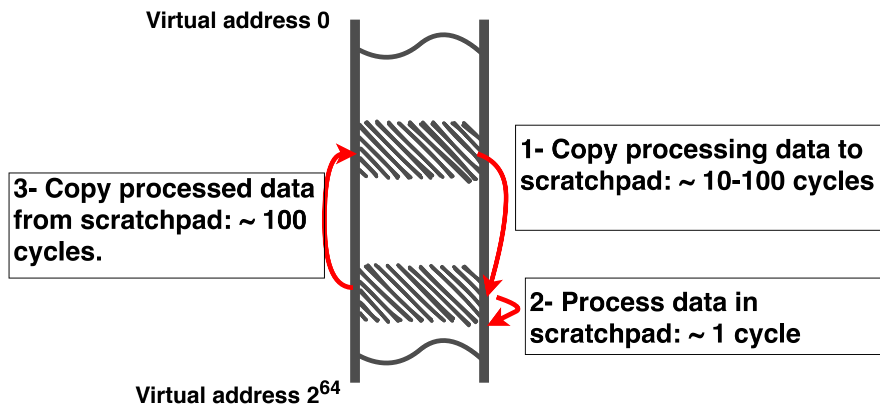


Figure 5.1: Application view of a software-process (UNIX) hosted on memory-hierarchy including a scratchpad memory.

Extending the HARDSI framework to support scratchpad (or any programmable) memory requires to define an adequate *transformation function*. As explained in subsection 4.2.2, this mathematical operator allows to construct the unique memory-signature that is followed to accessed the considered data-layout within an input code. This function needs to be adapted to the requirements of the used memory. If we consider the instance of scratchpads, an optimized implementation is the one that maximizes the number of useful addresses[1] stored within scratchpad while minimizing the number of accesses and transfers from (respectively to) the main memory. This needs to be done while prioritizing the addresses that are the most likely to be reused within a close time and then benefit from a scratchpad usage.

[1]Set of addresses that are relatively frequently accessed within a relatively short period of time.

5.1.2 HARDSI Transformation-Function for Scratchpad Memories

The *transformation function* that we propose for a variable v on a platform with a scratchpad memory is defined in Equation 5.1. This function associates the value $T_v^{SPM}[i]$ to the ith memory-access to the variable v ($T_v[i]$). We assume that when the number of accesses to a given address $T_v[i]$ is equal to one, then $T_v^{SPM}[i]$ is equal to zero[2].

$$\begin{aligned} T_v^{SPM}[i] &= \frac{1}{Occ(i)} * Av(i) \\ &= \frac{1}{\sum_{j=0}^{N-1} s_i(j)} * \frac{\sum_{j=0}^{Occ(i)-1} \sum_{k=0}^{N-1} Dirac(\sum_{l=0}^{N-1} s_i(l) - j)}{\sum_{j=0}^{N-1} s_i(j) - 1} \end{aligned} \tag{5.1}$$

where:

- $Occ(i)$ is the number of accesses to the address $T_v[i]$ in T_v
- $Av(i)$ is the average distance between two consecutive accesses to the address $T_v[i]$ in T_v
- N is the total number of accesses to the variable v (cardinal of the array T_v)
- s_i is the similitude function described in Figure 5.2a. $s_i[j] = 1$ if the ith address accessed is the same as the jth access to the considered data layout.

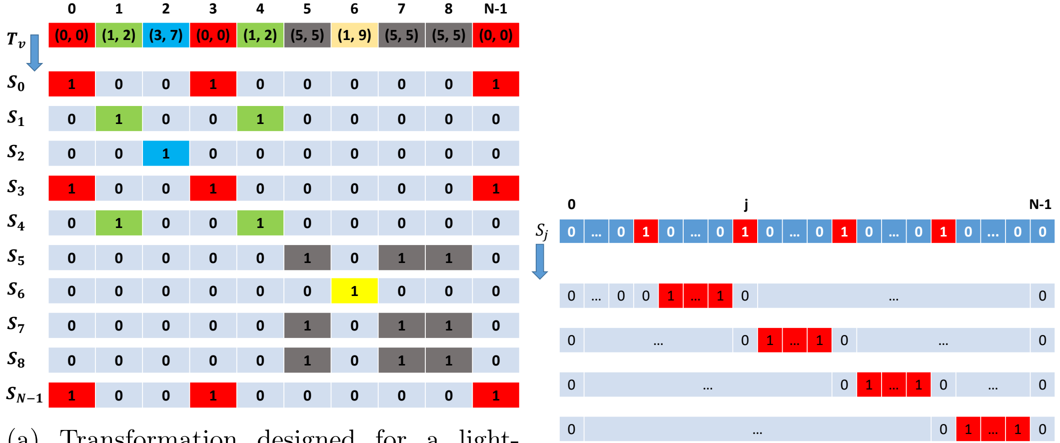
$$\forall i \in [0, N-1], s_i : \begin{cases} [\![0, N-1]\!] & \rightarrow \{0,\ 1\} \\ j & \rightarrowtail \begin{cases} 1 & \text{if } T_v[i] = T_v[j] \\ 0 & otherwise \end{cases} \end{cases}$$

- $Dirac$ is the discrete Dirac function:

$$\begin{cases} \mathbb{Z} & \rightarrow \{0,\ 1\} \\ j & \rightarrowtail \begin{cases} 1 & \text{if } j = 0 \\ 0 & otherwise \end{cases} \end{cases}$$

The *transformation function* proposed in Equation 5.1 is designed to classify the access-patterns to a data layout d following two properties. First, in order to decide whether or not to store a given data (i^{th} address $\alpha[i]$ accessed in d) in scratchpad memory, we need to classify the addresses of d according to their accesses-frequency (hence the first term $\frac{1}{Occ(i)}$). Second, we assume that the size of a scratchpad is smaller than the total size of manipulated data. Thus, in order to maximize the amount of data that might benefit from the scratchpad, it is primordial to evaluate the number of accesses to d between two consecutive accesses to $\alpha[i]$. A light-weight computation of this value is the average distance between two consecutive accesses to the address $\alpha[i]$ (hence the second term $Av(i)$). Finally,

[2]This reflects the fact that it is not worthy to store in scratchpad memory an address that is not reused. It also avoids division by zero in Equation 5.1



(a) Transformation designed for a light-weight computation of the number of occurrences of each address of the considered data layout.

(b) Transformation designed for a light-weight computation of the average distance between two consecutive accesses to the same address.

Figure 5.2: Notations used by the transformation function of the *HARDSI* method for scratchpad usage.

we build our *transformation function* based on a greedy approach: The further are two consecutive accesses to $\alpha[i]$, the more this address needs to be accessed in order to benefit from storing it in the scratchpad (hence the multiplication of the two previous terms).

The Occ and Av terms are built using the similitude arrays s_i instantiated in Figure 5.2a. For the ith address of d that is accessed ($T_v[i]$), $s_i[j]$ indicates whether or not the address accessed at $T_v[j]$ is the same as $T_v[i]$. Thus, the number $Occ(i)$ of accesses to the address $T_v[i]$ is $Occ(i) = \sum_{j=0}^{N-1} s_i(j)$. Meanwhile, to compute the average distance between two consecutive accesses to an address of d, we use the transformation of each similitude array s_i in Figure 5.2b.

5.1.3 Implementation and Multiple-Memories Issue

In order to adapt our *HARDSI* method to scratchpad memories, the process described in chapter 4 is implemented using the *transformation-function* in Equation 5.1. The considered kernel needs to be run in order to observe the memory-access pattern to each variable regardless of the target host-memory. However, when we are dealing with secondary programmable memories [3], two main problems need to be considered. First, the data transfers from and to the secondary memory generate a ratio from one to roughly ten accesses per legitimate access in the main memory. Even-though our method is shown to be resistant to noisy memory-accesses (section 6.6), these data transfers may change the most relevant access-pattern that was previously detected by our method (regardless to the used memory). Consequently,

[3]Such as scratchpad memories, GPUs, or C-SRAMs

the optimized data layout implementation that was previously selected for the main memory may not be ideal anymore for the part of d which remains in the main memory.
Second, in order to optimize an input code on scratchpad using the *HARDSI* method, we need to have at our disposal a set of data-layout that are specifically optimized for each considered access-pattern. Moreover, these data layouts need to be designed for scratchpad usage. However, as shown in section 3.4, the state of the art in terms of matrix data-layout for scratchpad usage does not propose enough implementations for all the patterns that we have identified. Moreover, we consider that existing solutions do not use the full memory-space of modern (i.e. relatively large) scratchpads. Consequently, we propose in the following section a custom approach to overcome the issue of in-adapted or under-performing data-layouts implementations.

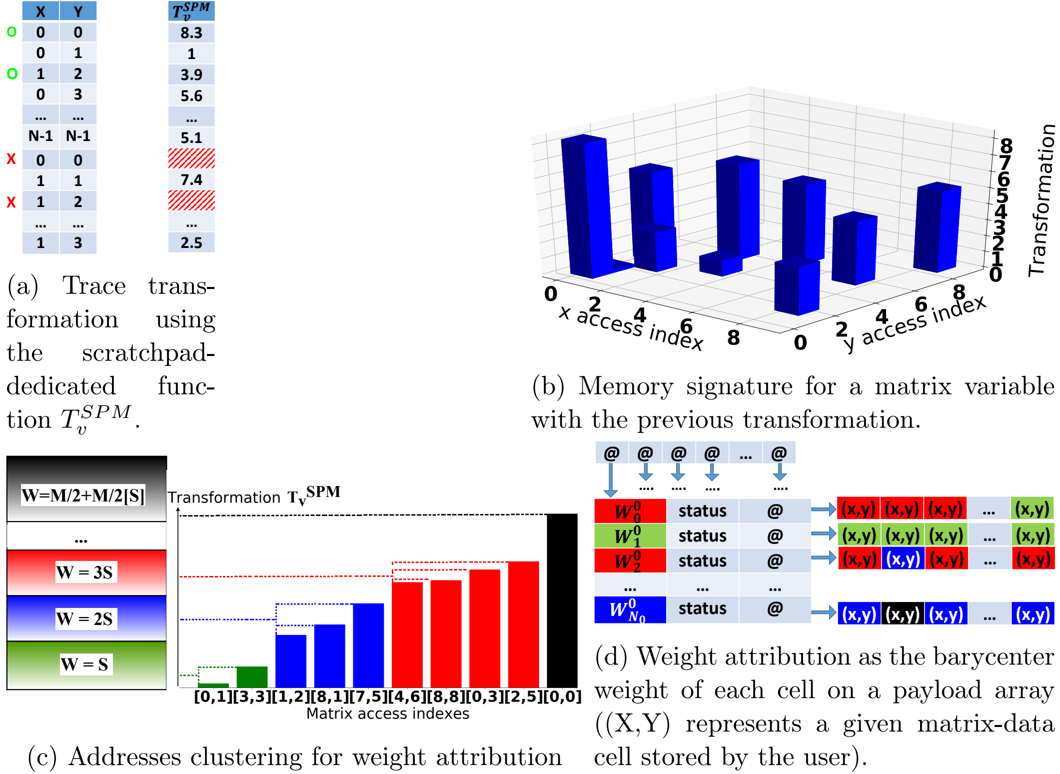
5.2 DDLGS: Generating Matrix Data-Layout Dedicated to Scratchpad

In this section, we describe DDLGS, a custom method proposed to dynamically generate data-layout implementations for scratchpad usage. The different steps of our method are illustrated in Figure 5.3. DDLGS stands for Dynamic Data-Layout Generation for Scratchpad. It consists in generating, for a given matrix, an optimized implementation, with regards to its specific access-pattern on both scratchpad and main memory. In order to improve the data-layout efficiency, the generated implementations are intended to dynamically (at run-time) adapt to the size of the considered matrix. Unlike the state-of-the-art approaches for data-layout implementation, we consider that modern scratchpads are large enough to host parts of a data-layout (such as a matrix). There is no need anymore to limit scratchpad usage to intermediate scalar variable or results.

5.2.1 Custom Generation Process

Let $H^{p,v}_{Matrix}$ be the histogram corresponding to a given pattern p followed to access a matrix data-layout within the considered source code. As stated in subsection 5.1.3, porting a given source code to a platform with a secondary memory changes the pattern observed in the main memory. However, we believe that defining an optimized implementation $I0^{p}_{Matrix}$ of a matrix while considering no scratchpad, is a mandatory step when optimizing a code for scratchpad usage. Indeed, this first step allows to generate a matrix implementation with a high level of data-locality while only considering the followed access-pattern p. The implementation $I0^{p}_{Matrix}$ is generated as described in section 4.2[4]. As shown in Figure 5.3d, the optimized matrix implementation I^{p}_{Matrix} that we generate keeps the same structure as $I0^{p}_{Matrix}$. Our

[4]Using the HARDSI method and based on the transformation function used for regular cache memories.



(a) Trace transformation using the scratchpad-dedicated function T_v^{SPM}.

(b) Memory signature for a matrix variable with the previous transformation.

(c) Addresses clustering for weight attribution

(d) Weight attribution as the barycenter weight of each cell on a payload array ((X,Y) represents a given matrix-data cell stored by the user).

Figure 5.3: Illustration of the four steps of the generation of a scratchpad-dedicated data-layout (with a fixed dimension).

approach is similar to the principle of a data-cache with a dynamic line-size scaling [41]. It consists in determining, for each newly-accessed address $\alpha[i]$ of $I0^p_{Matrix}$, the number $w_{\alpha[i]}$ of contiguous addresses that must be loaded along with $\alpha[i]$ whenever it is accessed while not present in scratchpad. This number $w_{\alpha[i]}$, referred-to as the weight of a matrix cell, is computed at compile time. As shown in Figure 5.3d, this weight is shared by all the matrix cells on the same payload-array[5]. It is computed as the barycenter of the weights of all the cells on the same payload-array. For performance purpose, the number $w_{\alpha[i]}$ of bytes that are transferred from and to the scratchpad is a multiple of a constant size S. In order to speedup the transfers from and to the scratchpad, we also ensure that the payload arrays are allocated aligned with the value of S. In the context of this thesis, the value of S is arbitrarily fixed. Finally, as shown in Figure 5.3d, each weight $w_{\alpha[i]}$ is stored along with a status byte and a set of load and store routines. The load and store routines are used to specifically read and write the corresponding payload-array from and to the main memory. An example algorithm that we design as a write routine is presented

[5] All the implementations that we generate for a 2D matrix are multidimensional arrays with a dimension potentially higher than two. The payload array is the one that contains the data stored by the matrix (as opposed to the indirection array which contains pointers to a subset of the considered data layout).

in algorithm 1. Meanwhile, the status byte indicates the current state of the corresponding payload array (e.g. dirty bit, address in main memory and parameters of the load and store routines).

Algorithm 1: Write the value input at the cell (x, y) of an input matrix potentially hosted on both scratchpad and main memory

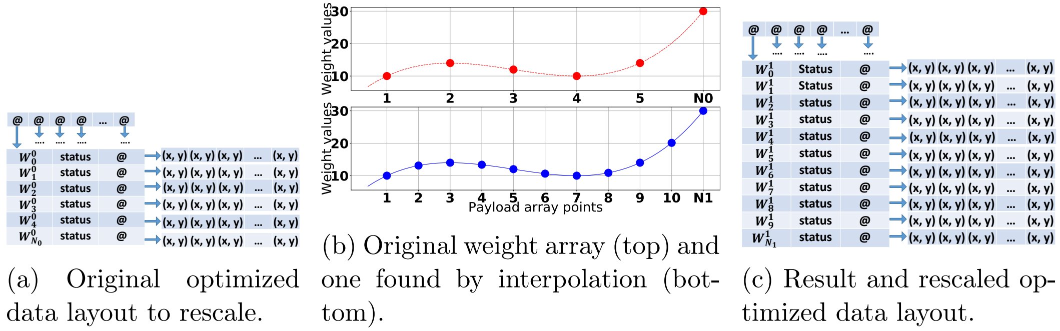
```
(w, status) = readMetaData(matrix, x, y);

if (isDataBelongScratchpad(w, x, y)) then
    @InScratchpad = getInScratchpad(status, x, y);
    if (@InScratchpad ≠ None) then
        updateInScratchpad(@InScratchpad, value);
        updateStatus(status, "inScratchpad", "dirty", x, y, 0);
    else
        (@MainMem, nbCell) = getDataToLoad(matrix, x, y, w);
        writeInMainMem(@MainMem, x, y, value);
        (statusEvicted, dataEvicted) = evictDataInScratchpad(w);
        if (statusEvicted ≠ None) then
            updateStatus(statusEvicted, "notInScratchpad", dataEvicted);
        transfertToScratchpad(@MainMem, nbCell);
        updateStatus(status, "inScratchpad", "dirty", x, y, nbCell);
    end
else
    writeDataInMainMemory(matrix, x, y);
end
```

5.2.2 Determining the Weight of Each Matrix Cell

The weight attributed to each payload array of the generated data-layout is probably the most critical parameter for performance [41]. In order to compute it for each variable v accessed following a pattern p, we use the corresponding histogram (memory-access signature). This histogram $H^{p,v}_{Matrix}$ is computed using the *transformation function* in Equation 5.1 and following the *HARDSI* method as shown in Figure 5.3: We track the accessed matrix-cells (Figure 5.3a) and build the signature as a histogram of the transformed row data (Figure 5.3b). Then, as shown in Figure 5.3c, the bins (cells of the matrix) of $H^{p,v}_{Matrix}$ are sorted from the less likely cell to benefit from scratchpad (lowest value for T^{SPM}_v) up to the most likely one (highest value for T^{SPM}_v). The bins with a zero value are not considered as it corresponds to cells that are used at most once. Thus, it would be counterproductive to load them to scratchpad memory.

As shown in Figure 5.3c, the weight of each matrix-cell is computed by clustering



(a) Original optimized data layout to rescale.

(b) Original weight array (top) and one found by interpolation (bottom).

(c) Result and rescaled optimized data layout.

Figure 5.4: Interpolation used to extend the dimension of an optimized matrix data-layout generated by the *HARDSI* method for scratchpad usage.

the values of the bins in the sorted histogram. The median value of each cluster is mapped to a predefined weight-value scale. Thanks to this mapping, we attribute to each cell within a given cluster the same weight as the median. The proposed scale is defined as a linear progression. It starts at an arbitrarily-fixed threshold base-value and increases following a stair with an arbitrarily fixed step-size. The maximum value of the scale is empirically fixed to $\frac{M}{X} + \frac{M}{X}[S]$ where M is the size in bytes of the scratchpad memory and X is a strictly positive integer. This corresponds to the fact that a transfer from or to the scratchpad may fetch at most one over X of the size of the memory (for the cells of the matrix that are the most likely to be reused in the considered code execution).
The used clustering method is the *Agglomerative Mean-Shift Clustering* (*AMSC*) algorithm [119]. On the one hand, this algorithm is used as it does not require to know the number of clusters beforehand (even though the maximum number of potential clusters might be set). This allows not to bias the result by forcing it to fit within an arbitrarily-fixed number of clusters. On the other hand, the *AMSC* algorithm might be parameterized in order to limit the maximum cardinal of a given cluster. These two properties allow to limit the usage of the scratchpad to the matrix-cells that are the most likely to benefit from the fast memory. Consequently, this greedy approach will limit the contention on the scratchpad in order to increase the overall performance.

5.2.3 Generalizing the Resulting Implementation to New Input Data

Through the presented method, we instantiate a data-layout designed to be used on a host hardware with a scratchpad memory. However, this implementation is only valid for the dimension of the data-layout provided during the optimization process. In this section, we show how to port the generated instance of optimized-implementation to any new input data dimension. It is noteworthy that this part of the optimization process is the only one that is computed at run-time (by the routine used for the allocation).

The optimized matrx-implementation that we propose for a given code has the same structure regardless of the input parameters. Indeed, we know from [88] that changing the input-values of a code[6] does not change the structure of the optimized matrix selected by the *HARDSI* method (as long as the memory-pattern followed to access the considered data-layout is the same). The only difference is the size of each Last-Level Indirection Array (LLIA)[7].

Let us consider an optimized matrix-implementation generated following subsection 5.2.2 for a given code and input value (Figure 5.4). Let us also consider a LLIA of this matrix that we represent with its size N_0 and its set of weights $w_{N_0}^{\{i\}}$. Our objective is to compute the set of weight $w_{N_1}^{\{i\}}$ of the corresponding LLIA (where N_1 is the size of the new LLIA) for an optimized matrix used in the same code with different input values. In order to attribute the weight-values $w_{N_1}^{\{i\}}$ for the new matrix, we use a custom transformation of the weight values $w_{N_0}^{\{i\}}$ of the previously generated matrix.

First, the set of weights $w_{N_0}^{\{i\}}$ is modeled using a custom approach of the *natural cubic spline with continuity* C^2. We propose this custom interpolation in order to model the weight function using a piece-wise polynomial. A third-degree polynomial (or spline) is built within each interval $[w_{N_0}^{i}, w_{N_0}^{i+1}])$. This method allows to interpolate the points in $w_{N_0}^{\{i\}}$ without shifting nor significantly emphasizing the local-extremums in this set and while avoiding *Runge's* (or edge-oscillation) phenomenon [14]. Hence the transfer of the different properties previously decided for the initial matrix-implementation to the new one. This property is ensured through the usage of our custom conditions:

- C^0: Each one of the $N_0 - 1$ polynomials crosses two contiguous nodes from $w_{N_0}^{\{i\}}$.
- C^1: Each two polynomials on each neighbor intervals have an equal derivative (slope) on their shared junction point.
- C^2: Each two polynomials on each neighbor intervals have an equal second-degree derivative (shape) on their shared junction point.
- C^{custom}: Each polynomial crossing a local extremum $w_{N_0}^{e}$ from the set $w_{N_0}^{\{i\}}$ has a null first derivative.

The $N_0 - 1$ splines ($4N_0 - 4$ real unknown coefficients) are then computed using the equations deduced from the previous conditions. The condition C^{custom} allows to generate twice as many equations as the number of local extremums in $w_{N_0}^{\{i\}}$.

[6] Hence changing the dimension of the considered matrix

[7] As shown in Figure 5.4, this array is the closest one to the payload array in the list of indirection arrays. It is the one that stores the weights of the payload arrays.

Indeed, for each extremum point $w_{N_0}^e$, the derivative of the upper and lower polynomials must be null in $w_{N_0}^e$. Given that the number of local extremums is between 0 and N_0, then the number of equations deduced from C^{custom} is in $[0, 2N_0]$. Following the same reasoning, the conditions C^0, C^1 and C^2 allow to generate respectively $2N_0 - 2$, $N_0 - 2$ and $N_0 - 2$ equations. However, the equations generated by condition C^{custom} are not always compatible with the one generated by C^1 as C^{custom} is a stronger condition on the derivatives than C^1. Thus, for each equation in C^{custom}, we reject the corresponding equation in C^1. Consequently, the resulting system is made of $4N_0 - 6$ equations with $4N_0 - 4$ variables. By adding a second-derivative condition on the edges (natural cubic splines), we have as many equations as variables. Given that these linear equations are linearly-independent (by construction), the corresponding matrix is reversible and the system admits a solution. The set of splines modeling the weight function is determined using a *Gaussian elimination* on this system.

Once we have interpolated the weights $w_{N_0}^{\{i\}}$ of the previous matrix, we determine the wights $w_{N_1}^{\{i\}}$ of the new matrix through the equation:

$$\forall i \in [0, N_1 - 1]$$
$$w_{N_1}^i = s_{\lfloor i \frac{N_0 - 2}{N_1 - 1} \rfloor}(i \frac{N_0 - 2}{N_1 - 1})$$

where s_x is the interpolation polynomial between $w_{N_0}^x$ and $w_{N_0}^{x+1}$ (for $x \in [0, N_0 - 2]$).

5.2.4 Potential Improvements

In this section, many parameters of our method have been fixed based on experimentation:

- Size S of a basic-block to be transferred from and to the scratchpad.
- Scale to be applied to the weight clusters.
- Minimum, maximum and step of the linear progression for the cluster mapping.
- Maximum fraction X of the scratchpad to be atomically fetched.

We acknowledge that finding these values and tuning them to the specificity of a given kernel is not straightforward. Changing such parameters may have a deep impact on the performance of the implementations that we generate. However, optimizing the value of these parameters is out of the scope of this thesis.

Part III

Experimental Validation and Discussion

Chapter 6

HARDSI Experimental Evaluation

6.1 Experimental Setup: HARDSI Code Compilation 75
6.2 Matrix Multiplication Kernel 76
6.3 Experimented Benchmark 77
6.4 Evaluation on a Three Data-cache Levels 78
6.4.1 Experimental Results Overview 78
6.4.2 Impact of HARDSI on the Different Cache-Levels 79
6.5 Enhancing the HARDSI Method to scratchpad memories 81
6.5.1 Regular Data Caches VS HARDSI Scratchpad implementation . . . 81
6.5.2 Baseline scratchpad VS HARDSI Scratchpad implementation 82
6.6 Impact of Noise on the HARDSI Method 84

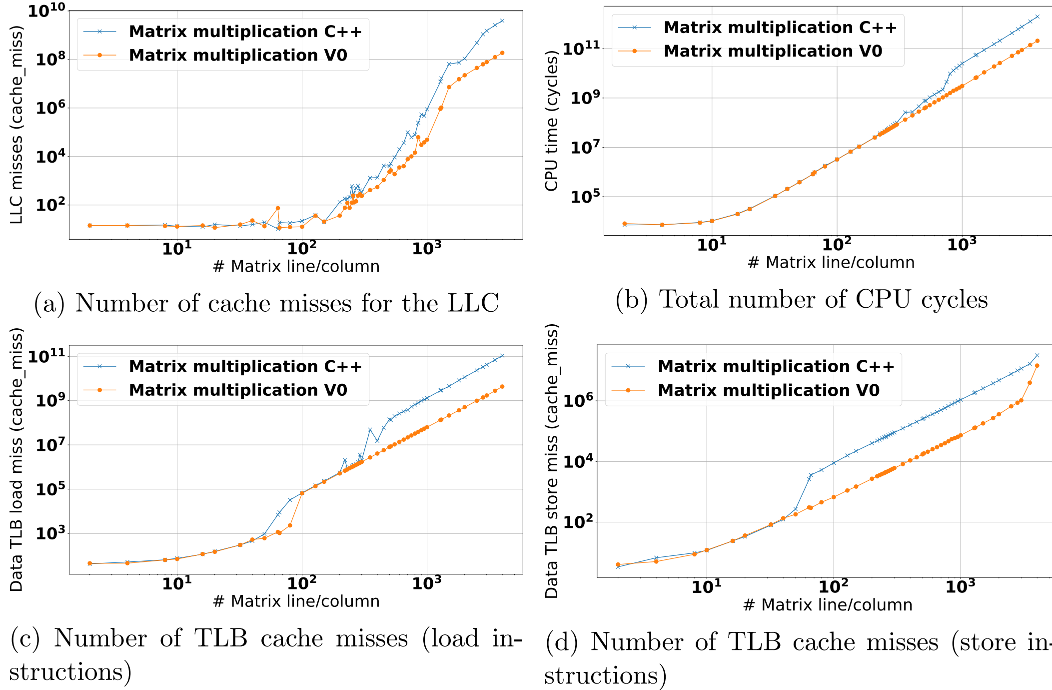
In this chapter, we assess experimentally the impact of the HARDSI approach on a data-intensive computation kernel. We present in section 6.1 the developed experimental setup to compare the HARDSI-generated code with the preexisting implementations. The objective being to present process used to build learn and compile the HARDSI with no biases related to the algorithms to implement. Then, we present in section 6.2 a first and full experimental evaluation using a matrix-multiply algorithm. In section 6.3 we introduce an experimental benchmark designed to evaluate the HARDSI code based a large set of memory-access patterns and data-layout architectures. Using this custom benchmark, we propose in section 6.4 and section 6.5 an experimental benchmarking of the HARDSI framework on both a regular three-data-caches and en embedded scratchpad-memory architectures. Finally, we present in section 6.6 a custom experimentation designed to show the strength of the HARDSI method toward external and internal memory noise.

6.1 Experimental Setup: HARDSI Code Compilation

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 (the considered kernels exhibit the same memory-access pattern regardless of the input values). The list of matrix implementations embedded within our HARDSI

data-base for both considered hardware platforms is shown in Figure 3.2. The matrix implementations in this base regarding the *x86* processor are extracted from literature. These implementations are used to populate the data-base corresponding to the *Coolidge* processor. Generating these matrix-implementations for scratchpad memories is done using the method in section 5.2.

6.2 Matrix Multiplication Kernel



(a) Number of cache misses for the LLC

(b) Total number of CPU cycles

(c) Number of TLB cache misses (load instructions)

(d) Number of TLB cache misses (store instructions)

Figure 6.1: Experimental comparison, based on the number of L3 (a), CPU cycles (b) and TLB (c, d) cache miss of a matrix-multiplication implemented in both *C/C++* and our custom HARDSI DSL.

In this section, we evaluate the impact of the data-structure-implementation generated by our method on the performance of a matrix-multiplication kernel. This experimental evaluation is processed on the considered *x86* processor. We compare two different implementations:

(i). **A native *C/C++* code** where the three matrices are defined and allocated using column-major 2D arrays (the cell at column x and line y of such matrices m being accessed using $m[x][y]$).

(ii). **The HARDSI code** shown in Listing 4.1 where the matrices are defined and accessed through the primitives of the HARDSI language relative to the matrix-data-structure.

Figure 6.1b shows that our HARDSI implementation allows to reduce the execution time of this kernel by up to $10x$. In order to explain this performance gain, we first assess the behavior of data-caches during the experimentation. We show in Figure 6.1a, that the same gain factor is observed in the number of data-cache misses. However, we notice when comparing the Figure 6.1a with Figure 6.1b that the two set of graphs are barely correlated. This allows to state some factors, other than data-cache enhancement, may have caused the observed performance gain.

Figure 6.1c and Figure 6.1d indicate that TLB-miss enhancement is another major improvement-factor brought by the HARDSI method. Indeed, these figures show that our custom implementation allows to reduce the number of TLB misses up to $9x$ and $11x$ during the execution of *load* and *store* instructions respectively.

Our experimental results show a significant asset of our optimization approach. When a data structure increases the locality of data consecutively-accessed, it does not simply reduce the number of data-cache misses but it also reduces the contention on all the hardware memory hierarchy. Indeed, Figure 6.1d shows that for matrices with a size around 50, the number of TLB misses for the native $C/C{+}{+}$ code increases brutally (10 times) compared to our custom HARDSI version. Such matrices have a memory size of $10KBytes$ (50 lines * 50 columns * 4 Bytes per cell). Unlike our implementations of these matrices, the $C/C{+}{+}$ version of the code leads to continuously have a majority of this whole memory in cache, which represents most of our L1 space ($32KBytes$). Each cache miss resulting from this cache-flood forces the operating system to realize many memory accesses for operations such as the virtual-page replacement. Each one of these operations being done at virtual-address level, an important contention is applied on the TLB (which stores translations between virtual and physical addresses). Hence the phenomenon observed in Figure 6.1d.

6.3 Experimented Benchmark

The results presented in section 6.2 assess the interest of our method when generating matrices accessed following a single family of memory pattern (line/column major with no stride nor sub-block). In order to enhance the set of evaluated patterns, we use the *PolybenchC-4.2.1* [83] 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 the access-types in the benchmark, 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) [62], *recursive-bilateral-filter*

(RBF) [116] and a *matrix-fast-exponentiation*[1] (PMFEA) [120]. The list of memory-access patterns evaluated through all these kernels is presented in Table 6.1.
On the *x86* processor, we consider five different input-matrix sizes: 50, 100, 600, 1000, 2048. This allows to represent different cases where the data fit one of the DL1, L2 or L3 caches (i.e., fitting one cache line or the whole cache).
Similarly, we consider five matrix sizes on the *Coolidge* processor: 30, 60, 100, 300, 900. This allows to assess cases where the data fits completely, partially or barely within the considered scratchpad (without exceeding the main-memory size and causing virtual-page swapping).

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

Table 6.1: Memory-access pattern followed by considered kernels.

6.4 Evaluation on a Three Data-cache Levels

6.4.1 Experimental Results Overview

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.

Table 6.2 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.

In Table 6.2, we show that our *HARDSI* method allows bring an execution-time speed up ranging from 1 to 48.9x. Our method is able to keep, in the 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.

[1]This algorithm, designed for parallel MPI usage, has been adapted to a single-node single-thread implementation.

Matrix size	50	100	600	1000	2048	50	100	600	1000	2048	50	100	600	1000	2048
	covariance					correlation					adi				
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
Speed up Best SoA	=	=	=	=	=	=	=	=	=	=	1.0	1.0	1.0	1.0	1.3
Reference Best SoA	[11]					[46]					[46]				
	gesummv					floyd-warshall					lu				
Speed up ***HARDSI***	1.4	1.3	7.1	12.2	12.5	3.9	5.4	28.9	48.9	29.4	1.0	1.0	1.5	4.0	6.3
Speed up Best SoA	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Reference Best SoA	[11]					[46]					[11]				
	JPEG-C					RBF					PMFEA				
Speed up ***HARDSI***	1.0	3.0	7.9	10.4	12	2.1	10.9	34.5	42.5	45.9	5.4	17.0	30.1	47.1	40.7
Speed up Best SoA	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Reference Best SoA	[62]					[116]					[120]				

Table 6.2: 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.

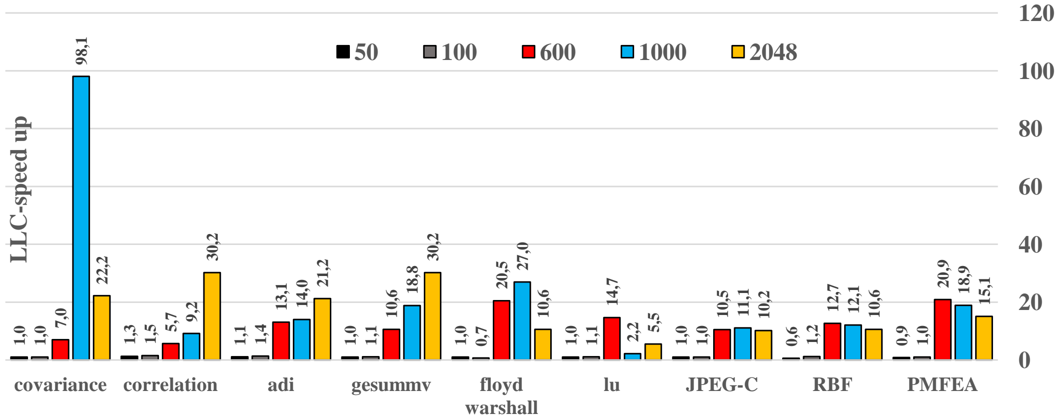


Figure 6.2: Performance speed up, in terms of LLC cache-misses (load and store), between a HARDSI and baseline implementation.

This case is primarily observed on the stencil kernel *adi*. Thanks to our method, we reache up a $2.3X$ speed up using a custom matrix implementation. This implementation [56] 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 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.

6.4.2 Impact of HARDSI on the Different Cache-Levels

For matrix-sizes higher than the total L3 size (1000 and 2048), Table 6.2 shows that the observed execution-time improvement for all the kernels other than *adi* ranges

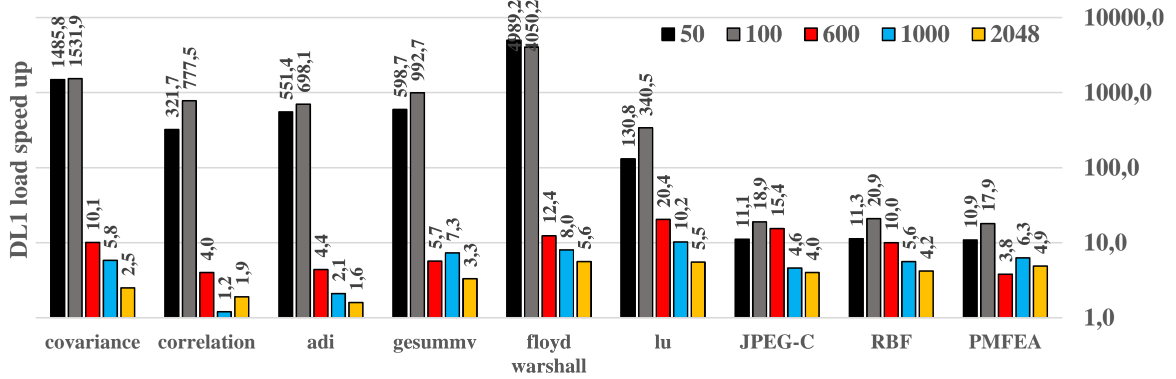


Figure 6.3: Performance speed up, in terms of data L1 cache-misses (load), between a HARDSI and baseline implementation (logarithmic scale).

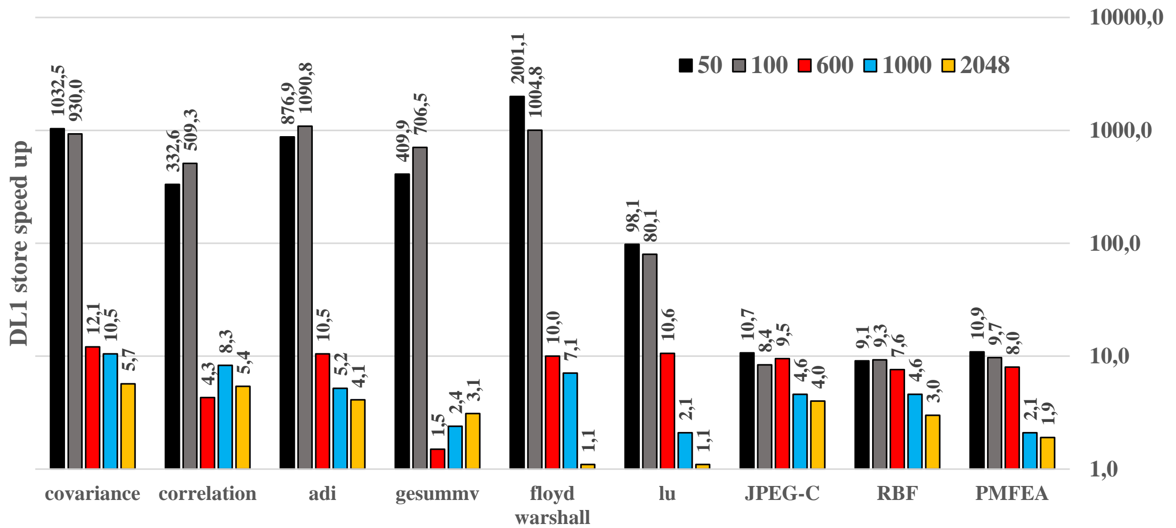


Figure 6.4: Performance speed up, in terms of data L1 cache-misses (store), between a HARDSI and baseline implementation (logarithmic scale).

from $4x$ up to $48.9x$. We show in Figure 6.2 that this time improvement is mainly explained by the reduction of L3-cache-misses (from 2.23 up to $98.09x$).

For matrices smaller than the size of the used L3 cache (50, 100 and 600), Figure 6.2 shows a cache-miss speed up as small as $0.20x$. 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 ≥ 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 the DL1 and L2 caches in order to store all the considered data. Meanwhile, the DL1 and L2 are intensively solicited. Thus, the speed up observed on DL1 and L2 (Figure 6.3 and Figure 6.4) 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.

6.5 Enhancing the HARDSI Method to scratchpad memories

In this section, we evaluate experimentally the impact of the *HARDSI* method on applications hosted on a platform with an on-chip scratchpad memory (*Coolidge* platform). All the scratchpad implementations considered by our HARDSI method are dynamically generated using the custom DDLGS algorithm introduced in chapter 5.

6.5.1 Regular Data Caches VS HARDSI Scratchpad implementation

The metric used in this section is the speed up (performance-ratio) between a scratchpad and a non-scratchpad implementation of each kernel. The non scratchpad implementation is the best known implementation of each kernel on the *Coolidge* processor while only considering its data-cache (L1-LLC) and no scratchpad memory. It is obtained thanks to the *HARDSI* method using the data-base relative to cache memories. Meanwhile, we consider two versions of the scratchpad implementation. In the first line of Table 6.3, this implementation is the one automatically selected by the *HARDSI* method as described in chapter 5. In the second line, it is the best known scratchpad implementation. This implementation if found by testing for each kernel all the known implementations in the data base.

Matrix size	30	60	100	300	900	30	60	100	300	900	30	60	100	300	900
	covariance					**correlation**					**adi**				
Speed up *HARDSI*	4.8	6.2	7.9	2.1	2.0	3.3	4.7	5.6	1.1	1.1	7.8	9.3	12.1	2.0	1.1
Speed up Best DB	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
	gesummv					**floyd-warshall**					**lu**				
Speed up *HARDSI*	8.5	10.2	9.1	4.2	2.5	3.0	4.9	4.2	1.3	1.9	5.2	5.7	7.7	1.8	1.0
Speed up Best DB	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
	JPEG-C					**RBF**					**PMFEA**				
Speed up *HARDSI*	10.0	15.0	20.8	7.1	5.3	12.9	20.5	30.1	6.4	5.9	10.1	20.6	54.2	7.8	1.9
Speed up Best DB	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=

Table 6.3: Performance speed up, in terms of CPU-cycles, between a scratchpad implementation and the best implementation (optimized for L1 data cache) known for each kernel. The scratchpad implementation is either the one automatically selected by the HARDSI method (first line) or the best found by testing all the matrix implementations in the HARDSI data base (second line). The value "=" means that the highest speed up reached using a HARDSI-selected implementation is equal to the one observed using the best tested implementation.

Similarly to the observations made for data-caches, Table 6.3 shows that our *HARDSI* method is able to select the best presented matrix-implementation for scratchpad usage. Indeed, the first line of Table 6.3 (speed up of the *HARDSI* implementation) is always equal to the second line (speed up of the best known implementation). Meanwhile, our result show that all the memory-access patterns do not take evenly-advantage of scratchpad memory. We notice that the tilled and stencil algorithms (*JPEG-C*, *RBF* and *PMFEA*) are the one that reach the higher speed up (up to 54.2x). This adequate behavior is due to the optimized and atomic usage of each block (tile) loaded into scratchpad before considering a next one. Meanwhile, kernels following a linear and non-tiled access-pattern (*covariance*, *correlation* and *gesummv*) benefit less from using a scratchpad memory (up to 7.9x). For these memory-access, the only optimization brought by our custom implementations is the ability to properly identify the data to prefetch in scratchpad before their usage.

The speed up of the scratchpad implementation compared to the non-scratchpad one may be explained in two contexts. On the one hand, when the the matrix fits in scratchpad (matrix sizes lower than 100), the highest speed ups are observed. This shows that our implementation benefits from the existing space in scratchpad and manages to lighten the cost of back-and-forth transfers between main and scratchpad memory. Our implementation identifies properly the matrix cells that would benefit from scratchpad usage. On the other hand, when the the matrix size is larger than the scratchpad size (matrix sizes higher than 100), the contention on the scratchpad increases dramatically the number of costly transfers from and to the main memory. Hence the relatively lower performance. In this work, we were not able to reach a matrix size where the scratchpad usage downgrade the performance. This is primarily due to the relatively limited main-memory size of our *Coolidge* platform (4M Bytes).

6.5.2 Baseline scratchpad VS HARDSI Scratchpad implementation

In this section, we compare our HARDSI scratchpad-adapted code with the corresponding state-of-the-art code proposed by *Kalray* the manufacturer of the considered processor. The results are presented in terms of CPU-cycles speed up: ratio of the *Kalray* to the HARDSI code execution time. It is noteworthy that *Kalray* did not disclose the implementation of all the algorithms that we consider in our benchmarks. In this cases (*floyd-warshall*, *JPEG-C*, *RBF* and *PMFEA*), we have proposed a custom *C/C++* implementation of these kernels. These implementations is based on same approach as the one followed by *Kalray* for the other kernels: storing the temporal data as well as the intermediate results that reused more than twice. The results that we present have been implemented using the best known implementation of the matrix data-structure for each kernel (without loading any part of it in the scratchpad memory).

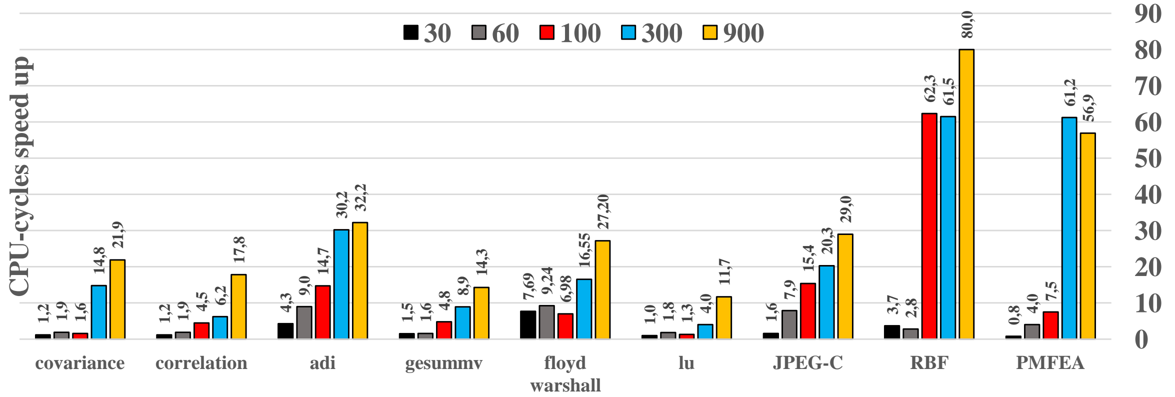


Figure 6.5: Performance speed up, in terms of CPU cycles between a HARDSI and a *Kalray* implementation designed for scratchpad usage.

In Figure 6.5, we show that the HARDSI code brings the higher improvement on the larger matrices (300 and 900). This gain, even though mostly higher than $1X$, is relatively limited on smaller matrices (30, 60, 100). This size-related improvement is mainly due to the method followed by the two set of implementations. The implementations proposed by *Kalray* are prety comonly used in litterature. It consists in only loading in scratchpad intermadiary results and highly-used temporary variables. Meanwhile, the implementation generated by our HARDSI compiler loads different intensively-used parts of the matrix in scratchpad. Consequently, the cost of our approach is harder to soften when the total number of operations is too low (small matrix-size or simple kernels). This leads in the worst case to a speed up of 0.8 (*PMFEA* kernel with an input-matrix-size of 30). However, this speed up lower than one is only observed once. In all the other unfavourable case, the speed from $1X$ up to roughly $4,5X$. This time-improvement is mainly due to the dynamical adaptation of our approach to the input of the considered code. Indeed, the code generated by the HARDSI compiler refactors the data-layout implementation at run time. At that time, the number of matrix-cells that are loaded at each accessed cell may be ideally set given that all the attributes of the problem are known: (i) ration between the matrix and memory size (ii) temporal distance between two accesses to the same cell and (iii) memory pattern followed to access each data-layout.

Such speed ups are particularly frequent for complex kernels (*JPEG-C*, *RBF*, *PMFEA*) with different successive patterns and a large number of interleaved/successive loops. In this context, the code iterates on single memory-access patterns and takes advantage of the optimization proposed at each iteration.

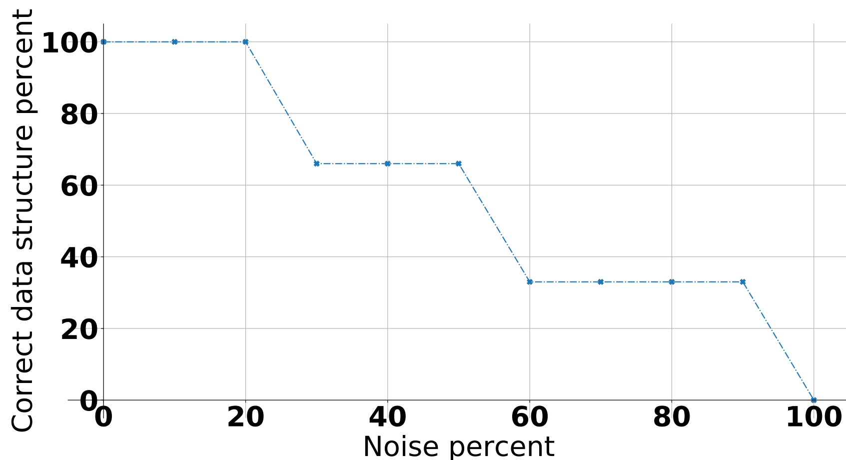


Figure 6.6: Percentage of variables (vertical axis) in the *gesummv* kernel that was positively identified by the HARDSI method for a given percentage of random memory accesses (horizontal axis) introduced at random moment of the kernel's execution. This figures is obtained on both the *Intel Xeon* and the *Kalray* processors.

6.6 Impact of Noise on the HARDSI Method

In a computation kernel accessing a given data-structure, we consider as noise a set of arbitrary accesses interleaved at arbitrary positions within a known memory-access pattern and that does not belong to the pattern. These are legitimate memory-accesses (not injected by malicious code) designed to store temporary information or to run functions of the algorithm. If we consider the kernels described in section 6.3, the number of noisy memory accesses is ranging from 2 up to 16% of the total number of memory-accesses made on the considered data structures. Noisy-memory accesses make it harder to detect the main memory-access pattern, which is the most interesting one to detect (as it is the most likely to trigger the most cache-misses and prefetcher-misses).

In Figure 6.6, we evaluate the resistance to noise of our HARDSI method. For this purpose, we inject different amount of random memory-accesses (from 0 up to 100% of the total number of memory-accesses) at random times during the execution of the *gesummv* kernel[2]. We show that for noise rate up to 20%, all the variables of the kernel are successfully identified by the HARDSI method: the initial pattern (excluding noise) is identified. Then for noise rate between 20 and 50%, the pattern of two over the three-considered variables is positively identified.

It is noteworthy that for noise rate higher than 50%, the initial pattern is not the most relevant one to identify. Indeed, for such cases, the noisy memory-accesses represent the majority of accesses. Thus, the noisy accesses become the one that

[2]Similar results are observed using the other kernels.

potentially influence most the performance.

Chapter 7

Conclusion and Future Work Directions

7.1 Summary and Conclusion **87**
7.2 Perspectives and Future Works **88**
7.2.1 Short Term Perspectives 88
7.2.2 Long Term Perspectives 89

7.1 Summary and Conclusion

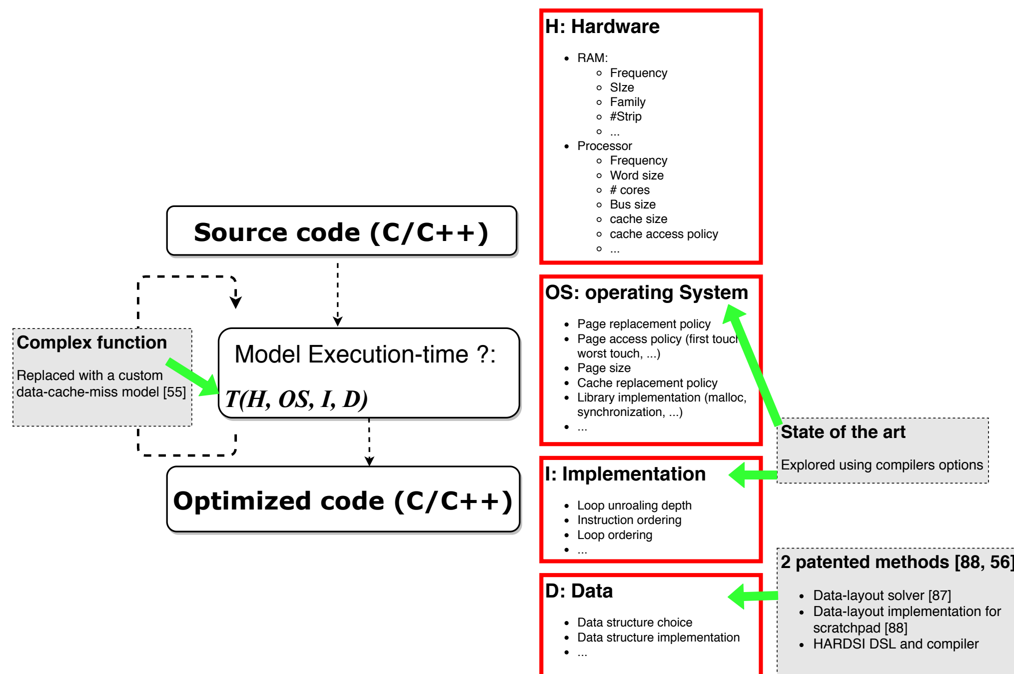


Figure 7.1: Global view of the proposed software-optimization process and the different contributions.

In this thesis, we considered tackling the portability issue that trickles down from modern heterogeneous and application-specific hardware accelerators. In this context, we considered a software-optimization approach aiming at automatically adapting an input source-code to its hardware environment. In Figure 7.1 we identified the time and mathematical issues to solve for implementing the considered

optimization approach. We also show the current implementation status of our custom optimization tool. The first problem considered in this thesis is the complexity of accurately measuring and modeling computational performances. This led us to identify the proper performance measuring tools. Given the specialization of each consider performance tracker, we proposed a custom patch allowing to select the proper implementation for each hardware, OS and software environment.

This thesis presents a custom contribution to solving the Data-Layout Decision problem. Indeed, choosing a proper data-layout implementation is critical for performance. Using the developed and patented HARDSI method and framework, we showed how to pick an adequate implementation with regards to the host hardware memory and access pattern. Our approach also showed an interest in using known and highly engineered data-layout implementations for optimizing new or unrelated code. The picked optimized implementations allowed reaching an execution-time speed-up up to $48.9x$[1] and an L3-miss reduction up to $98.1x$ on a three levels of data-caches architecture.

This thesis also focused on optimizing software code executed using embedded scratchpad memories. We acknowledged the fact that modern scratchpad memories are getting constantly larger. Existing approaches of storing only temporal and intermediate results, do not take full advantage of these fast memories. Consequently, we proposed DDLGS a patented method for dynamically generating scratchpad-dedicated implementations at run-time for matrix data-layouts. Given that these implementations are dynamic, the proposed solution may be adequately refactored depending on the followed memory-access pattern. Similarly, the run-time execution of this method allows to adapt to parameters that ignored at compile time (e.g. matrix and tiles size). The optimized implementations allowed reaching an execution-time speed-up up to $54.2x$[2] compared to a non-scratchpad-adapted optimization. Within the same hardware environment, the code generated using our DDLGS method outperformed the state-of-the-art one by up to $80X$ in terms of CPU-cycles.

7.2 Perspectives and Future Works

7.2.1 Short Term Perspectives

As a short term perspective, we propose to introduce a software-guided preprocessing of the code to optimize. The objective being to split the input code within sections to be optimized independently. Each section accessing the considered data layout following a different access pattern. Indeed we show in this thesis that splitting an input code into such sections allowed to improve the efficiency of the data layout generated by our HARDSI framework. However, this splitting task

[1] On the benchmark and input set presented in section 6.1
[2] On the benchmark and input set presented in section 6.1

has always been done by the programmer, which is only possible when the code is simple (relatively low number of interleaved loops and conditional branches). Indeed, splitting a code according to its different memory-access patterns is an NP-complete problem. A simple algorithm to solve this problem would be to compare for each line of code[3] the current memory signature with all the known signatures. Given the large number of potential access-pattern signatures ($O(N!)$ for a matrix of size N), a much more efficient algorithm needs to be proposed. Nevertheless, we believe that solving this tuning problem is an essential step toward porting the HARDSI approach to a general purpose compiler.

7.2.2 Long Term Perspectives

As a long term perspective, we propose to investigate the usage of the HARDSI method on a variety of high-performance architectures. In this thesis, we developed and deployed our HARDSI framework on single cores. However, we introduced different mathematical tools aiming to eventually guide a deployment on multicore machines. We also investigated the potential usage of the HARDSI method on emerging technologies such as the in-memory computing C-SRAM. One of the main limitation to such a wide-range deployment is linked to finding an adequate split of the code relative to each new architecture. We believe that this code-split needs to be realized automatically (e.g by the compiler). We also believe that solving this problem would allow to use the memory-signatures that we build for the detection of malicious-code injection. By accurately partitioning a source code, the HARDSI framework could be used to extract the memory-signature of some partition. By detecting changes in the signature of security-critical code partitions, we could spot the injection of malicious code. Given the statistical properties of the HARDSI signatures (consistence in between similar executions), this detection would potential improve the accuracy of current malware detectors.

[3] Or following a dichotomy exploration of the lines

Publications

The present thesis has lead to the following publications, patents and communications.

Published papers

- [89] R.Y Sid Lakhdar and H.P. Charles, *Cache-Miss Estimation Method Through Convolution-Based Kerne Analysis*. Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS). 2019. Bayonne, France.

- [57] R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Toward Modeling of Cache-Miss Ratio for Dense-Data-Access-Based Optimization*. 30th International Workshop on Rapid System Prototyping (RSP). 2019. New York, USA.

- [88] R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Data-Layout Optimization based on Memory-Access-Pattern Analysis for Source-Code Performance Improvement*. 23rd International Conference on Software and Compilers for Embedded Systems (SCOPES). 2020. Sankt Goar, Germany.

Patents

- [58] R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Method for constructing a memory-accesses signature by a microprocessor*. 2019. EU, US, JP, FR patent No. 1913348 Submitted on December 2019. .

Submitted Patent (under investigation)

- [59] R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Method of executing a computer program by an electronic computer comprising a main memory and a secondary programable memory*. EU, US, JP, FR. To be submitted in December 2020

Presented Posters

- R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Self-optimizing Scientific-Computation Libraries for High-Performance Computation*. 14th Fourteenth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES). 2018. Rome, Italy.

- R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Data-Cache-Miss Modeling for Self-optimizing Scientific-Computation Libraries at High-Performance Computation*. Google Compiler and Programming Language Summit. 2018. Munich, Germany.

- R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Toward Modeling of Cache-Miss Ratio for Dense-Data-Access-Based Optimization*. 23rd International Conference on Embedded Systems and Performance, Embedded Systems Week(EsWeeK). 2019. New York, USA.
- R.Y Sid Lakhdar, H.P. Charles and M. Kooli, *Solving the Data-Layout-Decision Problem for Code Efficient Hardware Portability*. Google Compiler and Programming Language Summit. 2019. Munich, Germany.

Bibliography

[1] Jeremy Adler and Ingela Parmryd. Quantifying colocalization by correlation: the pearson correlation coefficient is superior to the mander's overlap coefficient. *Cytometry Part A*, 77(8):733–742, 2010.

[2] Sam Ainsworth and Timothy M Jones. Paramedic: Heterogeneous parallel error correction. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–213. IEEE, 2019.

[3] Ayaz Akram and Lina Sawalha. × 86 computer architecture simulators: A comparative study. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 638–645. IEEE, 2016.

[4] David Atienza Alonso, Stylianos Mamagkakis, Christophe Poucet, Miguel Peón-Quirós, Alexandros Bartzas, Francky Catthoor, and Dimitrios Soudris. *Dynamic memory management for embedded systems*. Springer, 2015.

[5] Nasser Alsaedi, Steve Carr, and Alvis Fong. 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, 2018.

[6] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[7] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, 2004.

[8] Mario Badr and Natalie Enright Jerger. Synfull: Synthetic traffic models capturing cache coherent behaviour. *ACM SIGARCH Computer Architecture News*, 42(3):109–120, 2014.

[9] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[10] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*, pages 73–78. IEEE, 2002.

[11] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*. Springer, 2010.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[13] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.

[14] John P Boyd. Defeating the runge phenomenon for equispaced polynomial interpolation via tikhonov regularization. *Applied Mathematics Letters*, 5(6):57–59, 1992.

[15] Doug Burger and Todd M Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH computer architecture news*, 25(3):13–25, 1997.

[16] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy evaluation of gem5 simulator system. In *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*, pages 1–7. IEEE, 2012.

[17] Vincenzo Catania, Andrea Mineo, Salvatore Monteleone, Maurizio Palesi, and Davide Patti. Cycle-accurate network on chip simulation with noxim. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 27(1):1–25, 2016.

[18] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A Endo, and Rémy Gauguey. degoal a tool to embed dynamic code generators into applications. In *International Conference on Compiler Construction*. Springer, 2014.

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

[20] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. Scalapack: A portable linear algebra library for distributed memory computers-design issues and performance. *Computer physics communications*, 97(1-2):1–15, 1996.

[21] William E Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[22] Keith D Cooper and Timothy J Harvey. Compiler-controlled memory. In *SIGOPS OSR*. ACM, 1998.

[23] Intel Corparation. Intel (r) 64 and ia-32 architectures software developers manual. *Combined Volumes, Dec*, 2016.

[24] Johan Cronsioe, Brice Videau, and Vania Marangozova-Martin. Boast: Bringing optimization through automatic source-to-source transformations. In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. IEEE.

[25] Matthew Curtis-Maury, James Dzierwa, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 157–166, 2006.

[26] Yann Disser and Martin Skutella. The simplex algorithm is np-mighty. *ACM Transactions on Algorithms (TALG)*, 15(1):1–19, 2018.

[27] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, volume 200, 2005.

[28] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4):521–540, 2005.

[29] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, et al. Plasma: Parallel linear algebra software for multicore using openmp. *ACM Transactions on Mathematical Software (TOMS)*, 45(2):1–35, 2019.

[30] Julie Dumas. *Représentation dynamique de la liste des copies pour le passage à l'échelle des protocoles de cohérence de cache*. PhD thesis, 2017.

[31] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustavsson, and Hans Hansson. Towards industry strength worst-case execution time analysis. 1999.

[32] Stephane Eranian. Perfmon2: a flexible performance monitoring interface for linux. In *OLS*, 2006.

[33] Hamed Farbeh, Leila Delshadtehrani, Hyeonggyu Kim, and Soontae Kim. Ecc-united cache: Maximizing efficiency of error detection/correction codes in associative cache memories. *IEEE Transactions on Computers*, 2020.

[34] Keno Fischer and Elliot Saba. Effortless machine learning on tpus with julia.

[35] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98*. IEEE.

[36] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

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

[38] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in mapreduce. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE, 2012.

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

[40] Larry L Havlicek et al. Robustness of the pearson correlation against violations of assumptions. *Perceptual and Motor Skills*, 1976.

[41] G Glenn Henry and Stephan Gaskins. Dynamic cache enlarging by counting evictions, February 12 2019. US Patent 10,204,056.

[42] Shiwen Hu and Lizy K John. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 100–110, 2006.

[43] Wei Hu, Hong Guo, Hongna Geng, Kai Zhang, Jun Liu, and Xiaoming Liu. A novel design of software system on chip for embedded system. *Journal of Signal Processing Systems*, 86(2-3):135–147, 2017.

[44] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2013.

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

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

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

[48] Arun Kannan, Aviral Shrivastava, Amit Pabalkar, and Jong-eun Lee. A software solution for dynamic stack management on scratch pad memory. In *2009 Asia and South Pacific Design Automation Conference*, pages 612–617. IEEE, 2009.

[49] Nujoom Sageer Karat, Spandan Dey, Anoop Thomas, and B Sundar Rajan. An optimal linear error correcting delivery scheme for coded caching with shared caches. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 1217–1221. IEEE, 2019.

[50] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[51] Marta Kolasa. An energy-efficient, parallel neighborhood and adaptation functions for hardware implemented self-organizing maps applied in smart grid. *Energies*, 13(5):1197, 2020.

[52] Maha Kooli. *Analysing and supporting the reliability decision-making process in computing systems with a reliability evaluation framework.* PhD thesis, 2016.

[53] Maha Kooli et al. Smart instruction codes for in-memory computing architectures compatible with standard sram interfaces. In *2018 DATE*. IEEE.

[54] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance evaluation*, 67(8):634–658, 2010.

[55] Ulrich Kremer. Automatic data layout for distributed memory machines. Technical report, 1995.

[56] Riyane Sid Lakhdar. C/c++ implementations of a matrix data structure, https://github.com/simbadsid/datalayoutoptimization/tree/master/matriximplementations.cc.

[57] Riyane Sid Lakhdar, Henri-Pierre Charles, and Maha Kooli. Toward modeling cache-miss ratio for dense-data-access-based optimization. In *Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP'19)*, pages 64–70. ACM, 2019.

[58] Riyane Sid Lakhdar, Henri-Pierre Charles, and Maha Kooli. Method for constructing a memory-accesses signature by a microprocessor, US, EU, JP, FR. Patent No. 1913348, December 2019.

[59] Riyane Sid Lakhdar, Henri-Pierre Charles, and Maha Kooli. Method of executing a computer program by an electronic computer comprising a main memory and a secondary programable memory, US, EU, JP, FR. To be submitted in December 2020.

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

[61] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless dram cache. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 211–222. ACM, 2015.

[62] Alain M Leger et al. Jpeg still picture compression algorithm. *Optical Engineering*, 1991.

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

[64] Y-TS Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 298–307. IEEE, 1995.

[65] Tingyuan Liang, Liang Feng, Sharad Sinha, and Wei Zhang. Paas: A system level simulator for heterogeneous computing architectures. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

[66] Ting-Ru Lin, Drew Penney, Massoud Pedram, and Lizhong Chen. Optimizing routerless network-on-chip designs: an innovative learning-based framework. *arXiv preprint arXiv:1905.04423*, 2019.

[67] Ting-Ru Lin, Drew Penney, Massoud Pedram, and Lizhong Chen. A deep reinforcement learning framework for architectural exploration: A routerless noc case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 99–110. IEEE, 2020.

[68] Yang Liu, Wissam Sid-Lakhdar, Elizaveta Rebrova, Pieter Ghysels, and Xiaoye Sherry Li. A parallel hierarchical blocked adaptive cross approximation algorithm. *The International Journal of High Performance Computing Applications*, 34(4):394–408, 2020.

[69] Mirko Loghi, Massimo Poncino, and Luca Benini. Cache coherence tradeoffs in shared-memory mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):383–407, 2006.

[70] John McCarthy. History of lisp. *ACM Sigplan Notices*.

[71] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

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

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

[74] Gordon Moore. Moore s law. *Electronics Magazine*, 38(8):114, 1965.

[75] Samuel K Moore. Another step toward the end of moore's law: Samsung and tsmc move to 5-nanometer manufacturing-[news]. *IEEE Spectrum*, 56(6):9–10, 2019.

[76] Tipp Moseley, Neil Vachharajani, and William Jalby. Hardware performance monitoring for the rest of us: a position and survey. In *IFIP International Conference on Network and Parallel Computing*, pages 293–312. Springer, 2011.

[77] Shubhendu S Mukherjee and Mark D Hill. An evaluation of directory protocols for medium-scale shared-memory multiprocessors. In *Proceedings of the 8th international conference on Supercomputing*, pages 64–74, 1994.

[78] Abdolmajid Namaki Shoushtari. *Software Assists to On-chip Memory Hierarchy of Manycore Embedded Systems*. PhD thesis, UC Irvine, 2018.

[79] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. Applying the roofline model. pages 76–85, 2014.

[80] Antonio J Peña and Pavan Balaji. A data-oriented profiler to assist in data partitioning and distribution for heterogeneous memory in hpc. *Parallel Computing*, 51:46–55, 2016.

[81] Darko Petrović, Thomas Ropars, and André Schiper. Leveraging hardware message passing for efficient thread synchronization. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 143–154, 2014.

[82] Mikael Pettersson. Perfctr: the linux performance monitoring counters driver, 2005.

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

[84] Dick Price. Pentium fdiv flaw-lessons learned. *IEEE Micro*, 15(2):86–88, 1995.

[85] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Intermittent hardware errors recovery: Modeling and evaluation. In *2012 Ninth International Conference on Quantitative Evaluation of Systems*, pages 220–229. IEEE, 2012.

[86] Basireddy Karunakar Reddy, Matthew J Walker, Domenico Balsamo, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Empirical cpu power modelling and estimation in the gem5 simulator. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017.

[87] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[88] Henri-Pierre Charles Riyane, Sid Lakhdar and Maha Kooli. Data-layout optimization based on memory-access-pattern analysis for source-code performance improvement. In *23rd International Conference on Software and Compilers for Embedded Systems (SCOPES). 2020.* ACM, 2020.

[89] Sid Lakhdar Riyane and Henri-Pierre Charles. Cache-miss estimation method through convolution-based kerne analysis. In *COMPAS*, 2018.

[90] Felipe Rosa, Luciano Ost, Ricardo Reis, and Gilles Sassatelli. Instruction-driven timing cpu model for efficient embedded software development using ovp. In *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 855–858. IEEE, 2013.

[91] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[92] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. In *ICCS*, pages 1343–1352, 2015.

[93] Kayla O Seager, Ananta Tiwari, Michael A Laurenzano, Joshua Peraza, Pietro Cicotti, and Laura Carrington. Efficient hpc data motion via scratchpad memory. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 801–805. IEEE, 2012.

[94] Paul Sebexen and Thomas Sohmers. Software techniques for scratchpad memory management. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 98–102, 2015.

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

[96] Jason Sewall, S John Pennycook, Alejandro Duran, Christian Terboven, Xinmin Tian, and Ravi Narayanaswamy. Developments in memory management in openmp. *International Journal of High Performance Computing and Networking*, 13(1):70–85, 2019.

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

[98] Mohamed Wissam Sid Lakhdar. *Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures*. PhD thesis, Lyon, École normale supérieure, 2014.

[99] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[100] Muhammad Refaat Sedky Soliman. Automated compilation framework for scratchpad-based real-time systems. 2019.

[101] Vassos Soteriou, Hangsheng Wang, and L Peh. A statistical traffic model for on-chip interconnection networks. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 104–116. IEEE, 2006.

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

[103] Hossein Tajik, Bryan Donyanavard, Nikil Dutt, Janmartin Jahn, and Jörg Henkel. Spmpool: Runtime spm management for memory-intensive applications in embedded many-cores. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(1):1–27, 2016.

[104] S Tomov, J Dongarra, V Volkov, and J Demmel. Magma library. *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA*, 2009.

[105] Claus Traulsen, Jérôme Cornet, Matthieu Moy, and Florence Maraninchi. A systemc/tlm semantics in promela and its possible applications. In *International SPIN Workshop on Model Checking of Software*, pages 204–222. Springer, 2007.

[106] Richard A Uhlig and Trevor N Mudge. Trace-driven memory simulation: A survey. In *Performance Evaluation: Origins and Directions*, pages 97–139. Springer, 2000.

[107] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):3007–3020, 2017.

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

[109] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Efficient scratchpad allocation algorithms for energy constrained embedded systems. In *International Workshop on Power-Aware Computer Systems*, pages 41–56. Springer, 2003.

[110] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[111] Raymond L Watrous. Learning algorithms for connectionist networks: Applied gradient methods of nonlinear optimization. 1988.

[112] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[113] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.

[114] Brian JN Wylie, David Böhme, Bernd Mohr, Zoltán Szebenyi, and Felix Wolf. Performance analysis of sweep3d on blue gene/p with the scalasca toolset. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

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

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

[117] Yanqin Yang, Haijin Yan, Zili Shao, and Minyi Guo. Compiler-assisted dynamic scratch-pad memory management with space overlapping for embedded systems. *Software: Practice and Experience*, 41(7):737–752, 2011.

[118] Jieming Yin, Subhash Sethumurugan, Yasuko Eckert, Chintan Patel, Alan Smith, Eric Morton, Mark Oskin, Natalie Enright Jerger, and Gabriel H Loh. Experiences with ml-driven design: A noc case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648. IEEE, 2020.

[119] Xiao-Tong Yuan, Bao-Gang Hu, and Ran He. Agglomerative mean-shift clustering. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):209–219, 2010.

[120] Ji Zhang and Xu-chuan Zhou. A parallel algorithm for matrix fast exponentiation based on mpi. In *2018 IEEE 3rd International Conference on Big Data Analysis (ICBDA)*, pages 162–165. IEEE, 2018.

[121] Yangyang Zhao, Yuhang Liu, Wei Li, and Mingyu Chen. Hcma: Supporting high concurrency of memory accesses with scratchpad memory in fpgas. In *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8. IEEE, 2019.