Université de Versailles Saint-Quentin-en-Yvelines — École Doctorale "STV"

Généralisation de l'Analyse de Performance Décrémentale vers l'Analyse Différentielle

Generalization of the Decremental Performance Analysis to Differential Analysis

THÈSE

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines
(spécialité informatique)

par

Zakaria Bendifallah

Directeur de thèse :	William Jalby	- Professeur, Université de Versailles, France
Président :	Lee W.Baugh	- Ingénieur, Google, Seattle, USA
Rapporteurs :	Albert Cohen	- Directeur de recherche, INRIA, France
	Michel Krajecki	- Professeur, Université de Reims, France
Examinateurs :	Edouard Audit	- Directeur, Maison de la Simulation, France
	Andry Razafinjatovo	- Ingénieur, Bull, France
	Jean-Thomas Acquaviva	- Ingénieur, DataDirect Networks, France

LRC IT@CA — Exascale computing research

Remerciements

À mes parents et mes soeurs

Résumé :

Une des étapes les plus cruciales dans le processus d'analyse des performances d'une application est la détection des goulets d'étranglement. Un goulet étant tout évènement qui contribue Ã l'allongement temps d'exécution, la détection de ses causes est importante pour les développeurs d'applications afin de comprendre les défauts de conception et de génération de code.

Cependant, la détection de goulets devient un art difficile. Dans le passé, des techniques qui reposaient sur le comptage du nombre d'évènements, arrivaient facilement à trouver les goulets. Maintenant, la complexité accrue des micro-architectures modernes et l'introduction de plusieurs niveaux de parallélisme ont rendu ces techniques beaucoup moins efficaces. Par conséquent, il y a un réel besoin de réflexion sur de nouvelles approches.

Notre travail porte sur le développement d'outils d'évaluation de performance des boucles de calculs issues d'applications scientifiques. Nous travaillons sur DECAN, un outil d'analyse de performance qui présente une approche intéressante et prometteuse appelée l'Analyse Décrémentale. DECAN repose sur l'idée d'effectuer des changements contrôlés sur les boucles du programme et de comparer la version obtenue (appelée variante) avec la version originale, permettant ainsi de détecter la présence ou pas de goulets d'étranglement.

Tout d'abord, nous avons enrichi DECAN avec de nouvelles variantes, que nous avons conçues, testées et validées. Ces variantes sont, par la suite, intégrées dans une analyse de performance poussée appelée l'Analyse Différentielle. Nous avons intégré l'outil et l'analyse dans une méthodologie d'analyse de performance plus globale appelée PAMDA.

Nous décrirons aussi les différents apports à l'outil DECAN. Sont particulièrement détaillées les techniques de préservation des structures de contrôle du programme, ainsi que l'ajout du support pour les programmes parallèles.

Finalement, nous effectuons une étude statistique qui permet de vérifier la possibilité d'utiliser des compteurs d'évènements, autres que le temps d'exécution, comme métriques de comparaison entre les variantes DECAN.

Mots clés : analyse de performances, réécriture binaire, analyse dynamique, analyse statique du code, optimization de code, parallélisme, accès mémoire, compteurs matérielle.

Abstract:

A crucial step in the process of application performance analysis is the accurate detection of program bottlenecks. A bottleneck is any event which contributes to extend the execution time. Determining their cause is important for application developpers as it enable them to detect code design and generation flaws.

Bottleneck detection is becoming a difficult art. Techniques such as event counts, which succeeded to find bottlenecks easily in the past, became less efficient because of the increasing complexity of modern micro-processors, and because of the introduction of parallelism at several levels. Consequently, a real need for new analysis approaches is present in order to face these challenges.

Our work focuses on performance analysis and bottleneck detection of compute intensive loops in scientific applications. We work on DECAN, a performance analysis and bottleneck detection tool, which offers an interesting and promising approach called Decremental Analysis. The tool, which operates at binary level, is based on the idea of performing controlled modifications on the instructions of a loop, and comparing the new version (called variant) to the original one. The goal is to assess the cost of specific events, and thus the existence or not of bottlenecks.

Our first contribution, consists of extending DECAN with new variants that we designed, tested and validated. Based on these variants, we developed analysis methods which we used to characterize hot loops and find their bottlenecks. We later, integrated the tool into a performance analysis methodology (PAMDA) which coordinates several analysis tools in order to achieve a more efficient application performance analysis.

Second, we introduce several improvements on the DECAN tool. Techniques developed to preserve the control flow of the modified programs, allowed to use the tool on real applications instead of extracted kernels. Support for parallel programs (thread and process based) was also added.

Finally, our tool primarily relying on execution time as the main concern for its analysis process, we study the opportunity of also using other hardware generated events, through a study of their stability, precision and overhead.

Keywords: performance analysis, binary rewriting, dynamic code analysis, static code analysis, code optimization, parallelism, memory accesses, hardware counters.

Contents

1 Introduction **1**

2 Background on Micro-Processor Architecture **5**

2.1 Uni-core Design details . . . 5
2.1.1 Pipeline . . . 5
2.1.2 Multiple Issue Processors . . . 7
2.1.3 Vector Extensions . . . 7
2.1.4 Out-Of-Order Execution . . . 8
2.1.5 Caches . . . 9
2.2 Multi-core Designs . . . 11
2.2.1 Multiple Cores[92] . . . 11
2.2.2 Cache Organization[21] . . . 12
2.2.3 Shared Memory Support[89] . . . 13
2.3 GPUs and Many-Core Designs . . . 13
2.3.1 Graphical Processing Units (GPUs) . . . 13
2.3.2 Many-cores . . . 14
2.4 Design Example: Intel Sandy-Bridge Architecture . . . 14
2.5 Summary . . . 16

3 Application Performance Analysis **17**

3.1 Performance Life-cycle . . . 17
3.2 Performance Evaluation . . . 18
3.2.1 Techniques . . . 18
3.2.2 Performance Pathologies and detection methods . . . 19
3.3 Performance Evaluation Tools . . . 23
3.4 Discussion . . . 24
3.5 summary . . . 25

4 Differential Analysis **27**

4.1 Introduction . . . 27
4.2 Motivating Example . . . 28
4.3 DECAN: Practical Design . . . 30
4.3.1 Principle . . . 30
4.3.2 DECAN Target . . . 31
4.3.3 Variants . . . 31
4.3.4 Semantic Alteration . . . 32
4.3.5 Performance Monitoring . . . 32
4.3.6 Parallel Codes . . . 32
4.4 DECAN Variants Design . . . 32
4.4.1 Instruction Subsets . . . 33
4.4.2 Transformations . . . 35
4.4.3 DECAN Variants . . . 36
4.5 Differential Analysis: Main Analysis Methods and Metrics . . . 38
4.5.1 Observable Events . . . 39
4.5.2 Saturation . . . 39

4.5.3 LS/FP Analysis . . . 40
4.5.4 Data Location and Return On Investment . . . 42
4.5.5 Expensive Instructions . . . 43
4.5.6 Array Cost Analysis . . . 44
4.6 Case Studies . . . 46
4.6.1 Application Characterization and Analysis: RTM application 47
4.6.2 ACA: EUFLUXm Application . . . 49
4.6.3 L1 Load Bandwidth Evaluation . . . 50
4.7 Summary . . . 52

5 Pamda: Performance Assessment using Maqao toolset and Differential Analysis 53
5.1 Introduction . . . 53
5.2 Motivating Example . . . 54
5.3 Ingredients: Main Tool Set Components . . . 57
5.3.1 MicroTools: Microbenchmarking the Architecture . . . 58
5.3.2 Cqa: Code Quality Analyzer . . . 58
5.3.3 Decan: Differential Analysis . . . 58
5.3.4 Mtl: Memory Tracing Library . . . 59
5.4 Recipe: Pamda Tool Chain . . . 60
5.4.1 Hotspot identification . . . 60
5.4.2 Performance overview . . . 60
5.4.3 Loop structure check . . . 61
5.4.4 CPU evaluation . . . 61
5.4.5 Bandwidth measurement . . . 62
5.4.6 Memory evaluation . . . 63
5.4.7 OpenMP evaluation . . . 63
5.5 Experimental results . . . 63
5.5.1 PN . . . 64
5.5.2 RTM . . . 65
5.6 Related Work . . . 67
5.7 Summary . . . 68

6 Decan: Assembly Level Re-writing Challenges and Limitations 69
6.1 Introduction . . . 69
6.2 Decan Technical Design . . . 69
6.2.1 General Overview of the Maqao Framework . . . 70
6.2.2 Decan architecture . . . 73
6.3 Dealing with Control Flow issues in Decan . . . 74
6.3.1 Data Dependent Control Flow . . . 75
6.3.2 In-vitro Mode . . . 75
6.3.3 In-vivo Mode . . . 76
6.4 Extensions for Parallel Applications . . . 78
6.4.1 Shared Memory Codes . . . 78
6.4.2 Distributed Memory Codes . . . 80
6.5 Code Alteration Side Effects and Workarounds . . . 80
6.5.1 Code Layout Sensitivity . . . 81
6.5.2 Data Dependence Alteration . . . 83
6.5.3 Instructions with variable Latencies . . . 85

6.5.4 Instrumentation Side Effects . . . 86
6.5.5 Floating-point Exceptions . . . 86
6.5.6 Wrap-up: Side Effects Sources . . . 87
6.6 Summary . . . 88

7 Tackling Measurement Precision, Stability and Probe Intrusiveness 89
7.1 Introduction . . . 89
7.2 Events of Interest . . . 90
7.3 Experimental Methodology . . . 90
7.4 Measurement Stability . . . 91
7.4.1 Estimation . . . 92
7.5 Measurement Precision . . . 94
7.5.1 Small Regions . . . 94
7.5.2 Probes Accuracy . . . 96
7.6 Probe Intrusiveness . . . 99
7.6.1 Relationship Between Event Type and Probe Intrusiveness . . . 99
7.6.2 Reducing Probe Intrusiveness With Decan . . . 100
7.6.3 Experimental Results . . . 101
7.7 Summary . . . 103

8 Conclusions 105
8.1 Perspectives . . . 106
8.1.1 Tool Development . . . 106
8.1.2 Research Topics . . . 107

Bibliography 109

List of Figures

2.1 Five stages pipeline [49] 6
2.2 Vector operations on Intel architectures. The two technologies illustrated are SSE(Streaming SIMD Extensions) and Advanced Vector Extensions (AVX) 7
2.3 Processor with level three cache[13] 10
2.4 Common cache organizations, with two or three levels of cache . . . 12
2.5 Intel Sandy Bridge μarchitecture pipeline functionality [7] 15

4.1 Memory and Floating-point streams analysis with the variants `LS` and `FP`. The experiments are performed on 4 cores 29
4.2 Division and reduction impact analysis with the variants `NO_DIV` and `NO_RED`. The experiments are performed on 4 cores 29
4.3 Execution time and saturation curves for NR codelet `mprove` on different data sizes for a four cores execution. 40
4.4 Execution time and saturation curves for NR codelet `mprove` on different data sizes. Results are for a four cores execution. 42
4.5 Execution time and saturation curves for NR codelet `mprove` on different data sizes for a four cores execution. 43
4.6 Execution Slowdown of a version of the loop instrumented to perform FMT over its original version for `BALAN_3` codelet on different data sizes. The four levels of slowdown correspond from left to right to data being in L1, L2, L3 caches and RAM. 47
4.7 Execution time in cycles for the hottest loops of the RTM kernel. . 48
4.8 LS/FP saturations for the hottest loops of the RTM kernel 48
4.9 DL1 saturations for the hottest loops of the RTM kernel 49
4.10 The left figure illustrates the source code of the matrix-vector product in EUFLUXm. The right figure shows the individual contribution in the overall execution time of memory instructions targeting each array of the EUFLUXm routine. Results are presented for 2 and 4 cores. . 50

5.1 A Fortran source code sample and its main performance pathologies highlighted in pink. 56
5.2 Comparing static estimates obtained by CQA with dynamic measurements performed on different code variants generated by DECAN of both the original and the vectorized versions: **REF** is the reference binary loop (no binary modifications introduced by DECAN), **FP** (resp. **LS**) is the DECAN binary loop variant in which all of the Load-/Store (resp. FP) instructions have been suppressed, **REF_NSD** (resp. **FP_NSD**) is the DECAN binary loop variant in which only FP sqrt and div instructions (resp. all of the Load/Store and FP sqrt/div instructions) have been suppressed. The y-axis represents the number of cycles per source iteration: lower is better. 56
5.3 CQA output. 59
5.4 PAMDA overview. 60

5.5 Performance investigation overview. T means the condition is True, otherwise it is False (F) . . . 60
5.6 Detecting structural issues. . . . 61
5.7 DL1 subtree: CPU performance evaluation. . . . 62
5.8 LS subtree: Memory performance evaluation. . . . 63
5.9 OpenMP performance tree: STD represents the standard deviation between threads while the OVH branch stands for OpenMP Overhead evaluation. . . . 64
5.10 Streams analysis on PN. The REF curve corresponds to performance of the original code. The LS (resp. DL1) curve corresponds to the DECAN variant where all FP instructions have been suppressed (resp. all data accesses are forced to come out of L1). . . . 64
5.11 Group cost analysis on PN. Each group curve corresponds to performance of the loop while the target group is deleted. The original code performance (REF) is used as a reference. . . . 65
5.12 Evaluation of the cost of cache coherence protocol. The S2L variants show similar performance as their corresponding reference versions. The NO_STORE variants also show similar performances, except for two loops which present a relatively non negligible store cost. . . . 66

6.1 The MAQAO framework architecture . . . 70
6.2 DECAN tool workflow . . . 73
6.3 Flowchart showing the logic of the code generated by DECAN for the instance mode. As long as the loop call i is not reached the original version of the loop is executed. Once the loop call is reached, the transformed version of the loop is activated. The program is ended at loop exit . . . 77
6.4 Flowchart showing the logic of the code generated by DECAN for the recovery loop mode. . . . 78
6.5 Flowcharts showing the logic of the code generated by DECAN for the two openMP operatory modes. Flowchart (A) illustrates the case where all threads execute the same variant, and Flowchart (B) illustrates the case where different threads may executes different DECAN variants . . . 79
6.6 `DL1,FP` and `LS` variants on the four of the hot loops of the BT benchmark. . . . 80
6.7 `Stream` variants on 32 processes of the PN application . . . 81
6.8 Performance of the `REF`, `LS(ANMB)`, `LS(AN1B)` and `LS` variants for the NR codelet `toeplz_4` on a low frequency execution . . . 82
6.9 Performance of the `REF`, `LS(ANMB)`, `LS(AN1B)` and `LS` variants for the NR codelet `toeplz_4` on a high frequency execution . . . 82

7.1 For each event, the stability of all data size points for all the NR codelets of the test suite . . . 94
7.2 hqr_12 codelet source and binary codes. . . . 95
7.3 Evolution of the number of measured `L1D_REPLACEMENT` events per iteration following data size for `balanc` codelet. . . . 96

7.4 Evolution of the number of measured `MEMLOAD_UOPS_RETIRED_L1_HIT` events per iteration following data size for `balanc` codelet. 96
7.5 For each event, the distribution of the smallest event count starting from where the real and reference measures match 97
7.6 Minimum cycles count for which the real measure matches the reference measure for heavy measurement method that accesses the CPU_CLK_UNHALTED_CORE counter (**CCUC**) and light measurement method using **rdtsc**. Results are shown for 17 NR codelets. 98
7.7 Average error reduction and error increase after the subtraction of events generated by the probes 101
7.8 Ratio of successful overhead corrections (relative error <0,05) among error reductions (error increase cases are not taken into account) . . 102

List of Algorithms

1 Code example 28
2 LS/FP_analysis 41
3 Polaris (MD) loop 2937 source code 51
4 Reference measures 91
5 Real measures 91

List of Tables

3.1 A few typical performance pathologies. 20
3.2 Performance pathologies detection with static analysis techniques. . . 21
3.3 Performance pathologies diagnosis with simulation techniques 21
3.4 Performance pathologies diagnosis with dynamic time profiling techniques. 21
3.5 Performance pathologies detection with hardware counters based techniques 22

4.1 Assembly code of NR codelet SVDCMP_13 (SSE version). 33
4.2 Advanced data-flow analysis tracks the symbolic values of registers. Based on registers value it is then possible to infer which instructions are targeting the same data structure. Such instructions are coalesced within *groups* 35
4.3 Operands order change for a store instruction (the instruction is transformed into a load). 35
4.4 Source operand deletion of a vector multiplication instruction (the instruction is transformed into a load). 36
4.5 Vector multiplication instruction with a load operand transformed into a simple load instruction. 36
4.6 Vector multiplication instruction with a load operand to which a prefetch instruction is added. 36
4.7 Group detection results through both grouping static analysis and FMT runtime analysis 46
4.8 Results of group reconstruction for EUFLUX application with the use of both static and dynamic (FMT) analyzes. The cost of each analysis in slowdown over original execution time is shown 50
4.9 Stream analysis for loop 2937 of POLARIS (MD) 51
4.10 Effect of L1 cache bandwidth reduction on performance for loop 2937 of POLARIS (MD) 52

5.1 A few typical performance pathologies. 55
5.2 Decan variants and transformations. 59
5.3 Bytes per cycle for each memory level (Sandy Bridge E5-2680). . . . 62
5.4 PN Mtl results for the three most relevant instruction groups. . . . 65

6.1 Example of transformation which alters data dependency between instructions. The resulting instruction (V1) adds new dependencies whereas (V2) preserves the original ones 83
6.2 Code fragment of a loop extracted from POLARIS application. With an emphasize on initial and added dependencies on some instructions after transformation application 84
6.3 Comparison between the performance of REF variant and two versions of FP variant 85
6.4 Creation of an FP variant on a code which contains a division operation 85
6.5 Application of FP stream transformation on a code which contains a division operation 86

7.1 List of hardware counters of interest, available in the Intel Sandy-Bridge micro-architecture 90
7.2 Expected values for the approximation of l_val following the type of interaction between probe and loop events. 101

Chapter 1

Introduction

General context

How to complete the execution of a piece of code as fast as possible has always been a primary concern since the emergence of computing machines. This goal is tackled from several perspectives by different actors: hardware vendors seek to provide designs that can be efficiently exploited, programmers focus on algorithms more efficient in the exploitation of underlying architecture, compiler developers attempt to polish and optimize the written code and performance analysts try to find and fix performance issues affecting the final code. We subscribe to the later category and we also focus on *HPC (high Performance Computing)*. HPC is characterized,on the software side, by major simulation programs for various domains(crash simulations, thermodynamics, molecular chemistry, *etc*), and on the hardware side, by the use of supercomputers, the biggest computing machines ever made.

The time to completion (time to finish the task or a program) is important in HPC. In his presentation, *Tadashi Watanabe*, Project Leader of Next-Generation Supercomputer R&D Center RIKEN and designer of the *K supercomputer*(the world´s fastest supercomputer in 2011 and 4th fastest at the time of this writing) stated, two months after the *Fukushima nuclear disaster* in Japan, that the simulation of the tsunami wave took *2 hours* with their fastest supercomputer at the time of the catastrophe, and that the same simulation only takes *10 minutes* with the *K* supercomputer. Knowing that it took the tsunami approximatively 50 minutes to reach the nuclear power plant, he argued that the use of *K* would have given some time to start reacting to it.

However, recent years recorded new turns in processor evolution, uni-core processors were deprecated in favor of multi-core processors. Modern day supercomputers are clusters of processors, connected with either standard or dedicated communication networks, hence allowing thousands of processor cores to be exploited in parallel. This raised new challenges: energy consumption, for example, becomes a primary concern (the most powerful supercomputer at this time consumes near *17,808* KW [16]). Therefore, the energy consumed to complete the task becomes another goal which usually holds a trade-off relationship with the first one.

On the application side, several programming paradigms emerged. We thus find a widespread use of distributed memory models, notably with the use of the Message Passing Interface MPI [47]). They aim to exploit the parallelism offered by a high number of processors connected with a communication network. At node level, the use of shared memory models is common notably with technologies such as OpenMP [30] and Intel TBB [84]. Finally, at core level, we heavily rely on optimizing technology of compilers such as GCC [6] or ICC [10]. The objective is there to design a code generation scheme which allows to make the best use of the power offered by the hardware.

Providing powerful hardware is a valuable asset to achieve high performance. However, the aggregation of more computing power is not a solution in itself, due to the severe technical limitations such as communication latency and power consumption, as well as software limitations such as non-scaling algorithms. Additionally, compilers contribute to better performance, but by becoming too complex, they fail to find the optimal combination of optimizations. Performance analysis tools complete the cycle by giving feedback on the nature of performance bottlenecks and their locations.

Application Performance Analysis

Performance bottlenecks in HPC applications can be located at various levels. In a typical parallel application, they are generally ordered according to the granularity at which they occur: *i*) inter-node bottlenecks are at the highest granularity, they mainly concern communications between different processors, load balancing, synchronization, *ii*) intra-node bottlenecks are in a midst granularity and cover all intra-node interactions such as load balancing and data sharing between threads, *iii*) core bottlenecks are found at the lowest granularity and cover the good exploitation of hardware (vectorization, data locality, *etc*).

Each level of granularity has its own set of analysis techniques. For inter-node bottlenecks, the analysis is in general done through source code instrumentation. This does not affect the normal behaviour, the difficulties mainly being to handle the volume of collected data. As far as intra-node and core issues are concerned, the focus is put on fine grain bottlenecks. They are the direct effects of interaction between the code and the underlying architecture. Hardware related bottlenecks are very sensitive. A simple probe which monitors cache effects in the L1, can generate memory references and thus bias the result. Hardware vendors included in their designs what is known as performance monitoring units (PMUs), a set of hardware devices which monitor several micro-architectural actions (cache hits and misses, branch mis-predictions, number of instructions dispatched in each execution port). These made analysis tools less intrusive. However, they present some disadvantages too. The counters differ in their semantics from architecture to architecture and are not well documented. But most importantly, in the majority of cases, they only assess quantitive date, no qualitative such as the weight in terms of performance issues. They answer the question:"how often a particular event occured but give no information on how it affected performance". The number of occurences is an interesting metric to characterize code fragments, but it fails to determine the impact of an event in a context where events may occur in parallel (the impact of an event can be partially or completely masked), which is the case in modern architectures (e.g. out-of-order execution and multi-threading).

Analysis methods are diverse too, in the previous paragraph we supposed that the analyses are performed on the program at runtime, the set of techniques based on this approach are gathered under the *measurement* category. It is also the category to which our work subscribes. The other two notable categories include *simulation* and *modeling*.

Contributions

Our work also includes measurement techniques. Traditional measurement and

instrumentation techniques rely on pure observation of the events. We believe that, at such small scale, it is difficult for a performance tool, given the complexity of modern architectures, to associate a weight (or cost) to a particular event (that could be a bottleneck). Our work starts from the base idea of event idealization, where a version of the code is created in which the event (e.g. memory reference) is idealized (suppressed), the performance change between the modified and original versions of the code indicates the impact of the event. The idea was first introduced with a tool called DECAN [57]. We propose to continue the development of the concept further to target more performance issues and broaden its use beyond application performance analysis.

With our research we make the following contributions:

New bottleneck identification techniques: we conceived, tested and validated new DECAN variants that enabled us to identify performance bottlenecks. We also, redefined the concepts of the tool in a more abstract way in order to use them more easily on other architectures.

Moving from In-Vitro to In-Vivo mode: the previous version of the tool operated on extracted fragments of code (in-vitro mode). We developed techniques to use the tool at runtime directly on the loops of real industrial applications (in-vivo mode).

Performance assessment methodology: we use the flexibility of the technique and its characterization capabilities to make it play a central role in a methodology, aiming at coordinating several analysis tools and making them collaborate in bottleneck investigation. There is no need for a heavy tool when a lighter one can be used to investigate an issue.

Tackling measurement issues: we demonstrate that Differential Analysis can be applied to metrics other than time. To achieve that, we study how accurate and reliable hardware counters can be.

Outline

This dissertation is organized as follows:

- Chapter 2 discusses background and related work in the field. We review the important micro-architectural components in a modern micro-processor.
- Chapter 3 provides material for bottleneck detection by outlining well-known intra-node performance pathologies as well as the common diagnosis techniques.
- In Chapter 4 we introduce the building blocks that enable *Differential Analysis* as well as the analyses it provides.
- Chapter 5 builds an analysis methodology in which tools of different natures are put together, with DECAN as a central tool for coordination.

- In Chapter 6 we present some of the technical challenges a tool such as Decan raises, and discuss how we handled them.
- Chapter 7 addresses measurement precision, stability and probes instrusiveness. It starts with an analysis and a classification of the most commonly monitored events and proposes an overhead reduction technique for the targeted events.
- Future work suggested and motivated by our research is outlined in Chapter 7 and finally, Chapter 8 presents our conclusions.

Chapter 2

Background on Micro-Processor Architecture

Processor design has seen major developments in recent years. The focus on uni-core processors favored the development of complex and powerful designs as well as performant memory hierarchies. A sustained increase in processing speed could be achieved by shrinking transistor size, but problems such as die heat rose, which made further shrinking too costly, Therefore, the design trend shifted toward multi-core designs in order to achieve the desired performance needs. In addition to uni-core mechanisms, Multi-cores add another level of complexity due to inter-core parallelism. Furthermore, processors need to access the data stored in a central memory outside the processor die. This task is critical and has been of primary concern in processor designs too. It leads to the development of smaller memories close to the processor called caches.

The present chapter reviews some important processor micro-architectural details including those of the Intel Sandy-Bridge platform as it is the main architecture on which we tested and validated our work.

We however note that the notions introduced within the chapter are by far not exhaustive, and that specialized books [49] provide richer and more complete material.

2.1 Uni-core Design details

A Uni-core processor is based on a computer architecture that has a single processing unit. In this design, a program counter sequences the work by executing instructions one-by-one. The current instruction has to be finished before the next one starts. This ensures a sequential execution of programs. Program data are stored in an external central memory, and circulate between it and the processor through a bus.

Several improvements were set up to tune the base design in order to achieve better performance. The exploitation of different forms of parallelism allowed to increase instructions throughput. Fast access to data being critical to performance, the addition of several levels of caches shortened the time to access data. In the current section we review some of the most significant improvements.

2.1.1 Pipeline

Pipelining is an implementation technique which consists of dividing the instructions execution process into several stages. It allowes several instructions to be executed at the same time. This is achieved by leting each isntruction to be in a different stage of the pipeline at a given clock cycle. The technique takes advantage of the opportunity of executing in parallel various actions within an instruction. Pipelining offers the advantage of keeping all portions of the processor occupied. By increasing the amount of useful work done, it increases the throughput of instructions.

Figure 2.1 shows a 5 stages pipeline of a MIPS processor which uses a RISC instruction set [49]. In this architecture, an instruction passes through the following stages to complete execution:

- *Instruction Fetch (IF)*: the next instruction to execute is fetched from memory
- *Instruction Decode (ID)*: decodes the fetched instruction. The decoding recognizes the opcode as well as the operands the instruction uses.
- *Execution (EX)*: Executes the instruction. The execution is handled by the ALU (arithmetic and logic unit), if the opcode corresponds to a memory operation than the effective address is computed, otherwise, the operation is performed on the decoded operands
- *Memory Access (MEM)*: if the operation is a load, the operand is read from memory, an if it is a store, the content of the register operand is written to memory
- *Write-back (WB)*: if the operation produces results, these are written back to the register file.

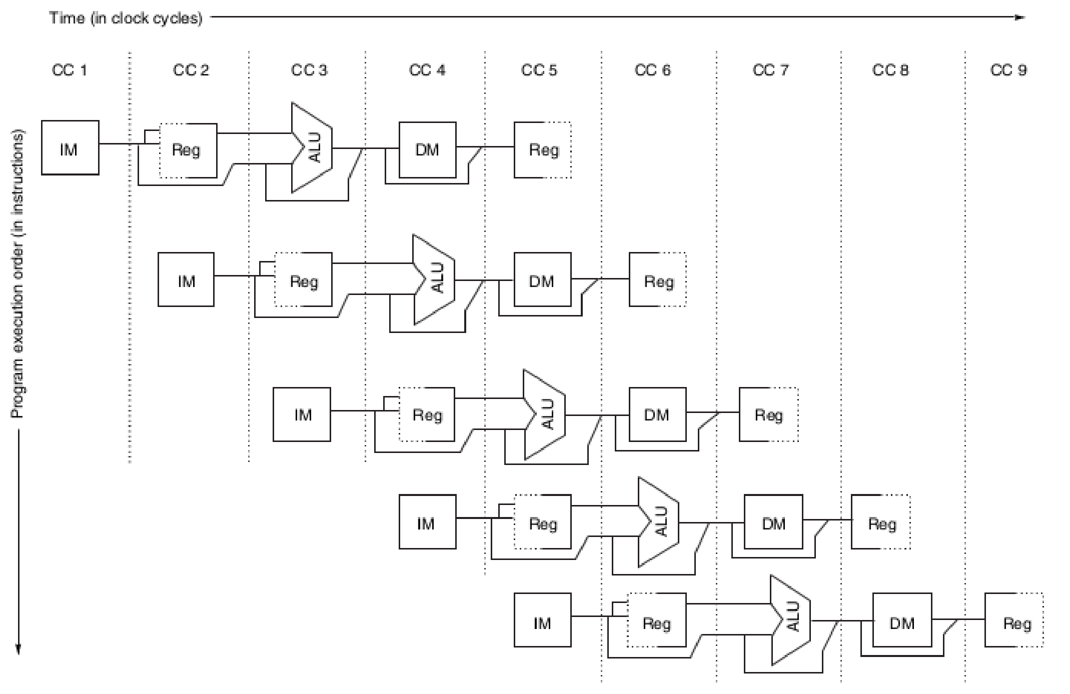


Figure 2.1: Five stages pipeline [49]

The above pipeline example remains simple compared to modern pipelines. These have more execution stages, some of which can be complex, hence having a latency of more than one cycle. The Sandy-Bridge presented in Section 2.4 gives an overview of such pipelines.

But pipelining generally presents disadvantages too, the most notable being[80]:

- Increased hardware complexity which generates resource conflicts, control dependency and data dependency.

- Pipeline stalls: due to what is called pipeline hazards (data hazards, disrupted control, *etc*). They prevent smooth execution of the pipeline by causing flows (bubbles) in it.

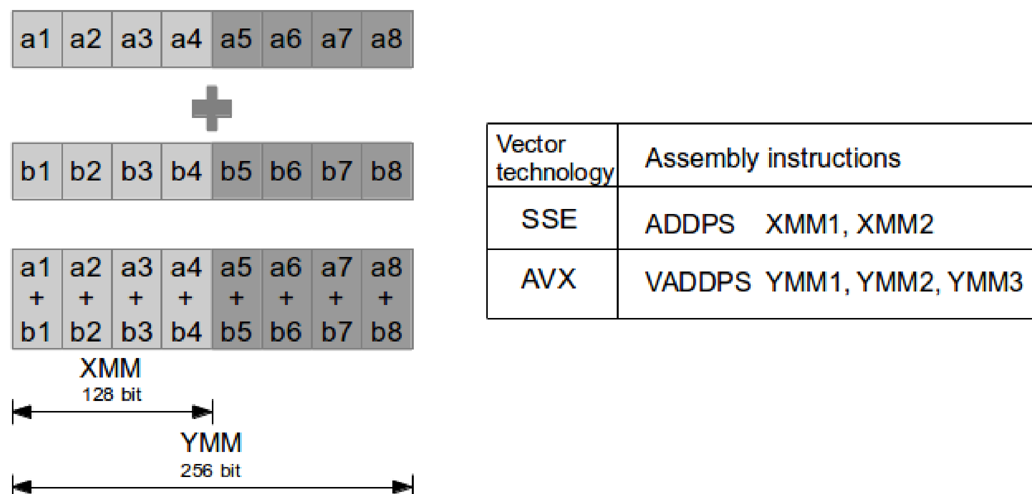
2.1.2 Multiple Issue Processors

Multiple issue processors (also called Superscalar architectures) are architectures where multiple instructions are issued at the same time. The goal is to achieve a bigger *instructions per cycle* (IPC) ratio by exploiting the independence available between instructions, introducing what is called *instruction level parallelism*. In general, multiple issuing is combined with pipelining in order to enable several instructions to be simultaneously initiated within a pipeline stage, in other words a superscalar architecture can be seen as several pipelines working at the same time. The same concept is applied at the pipeline stage level also (e.g. multiple functional units), allowing several operations of the same type to BE executed in parallel. Several types of superscalar architectures have been desgned. We find: *statically scheduled superscalar architecture*, *VLIW (Very Long Instruction Word)* and *dynamically scheduled superscalar architecture*. The later one is the most widespread in current processor designs, it is also the one we focus on within our study.

Superscalar architectures suffer from the same issues as pipelines, namely: resource conflicts, control dependency and data dependency.

2.1.3 Vector Extensions

Vector extensions are parts of the functional units of the processor which work on packed data. A subset of the *Instruction set architecture (ISA)* allows to exploit these units. By doing so, the processor has the ability to exploit data parallelism when possible. In order to expose such parallelism to the processor, the compiler constructs *vectorized loops*. When the vectorization is total, all loads, stores and ALU instructions operate on vectors of data, whereas when it is partial only part of of the instructions operate on vectors of data.



Vector technology	Assembly instructions
SSE	ADDPS XMM1, XMM2
AVX	VADDPS YMM1, YMM2, YMM3

Figure 2.2: Vector operations on Intel architectures. The two technologies illustrated are SSE(Streaming SIMD Extensions) and Advanced Vector Extensions (AVX)

Figure 2.2 illustrates an addition operation performed using two technologies of vector extensions implemented in the Intel x86 micro-architecture: 1)*Streaming SIMD Extensions (SSE)* which works on 128 bit long vectors and is able to pack four elements in single precision (32 bit each) or two elements in double precision (64 bit each), and 2) *Advanced Vector Extensions (AVX)* which is the successor of SSE on newer Intel micro-architectures. and operates on arrays with double the size of those of the former technology, hence allowing up to eight elements in single precision and four in double precision.

Vector extensions technology is present in other architectures under other names. AMD uses both SSE and AVX in addition to its own technology 3D Now [1], whereas ARM developed its own technology called NEON[5].

2.1.4 Out-Of-Order Execution

Out-Of-Order execution is a technology which enables instructions to not follow the *first in first out* (FIFO) strategy. Instead, they are buffurized in what is called an instruction window. Within this window, each instruction whith ready operands is dispatched to free functional units. Hence, non-ready instructions cannot block the ready instructions that arrive after them. Once the execution finishes, the processor ensures an in-order commit (WB) of instructions in order to preserve the correct program semantic. In general, out of order execution starts in the *execution stage (EX)* of the pipeline (see Figure 2.1). The following elements play a key role in an out-of-order engine:

Instruction buffer: when an instruction terminates its passage in a pipeline stage, it is recorded in a temporary register called a*latche*. Latches enable instruction preservation in case of pipeline stalls. In an in-order pipeline one latche is needed in each stage because only one instruction can be active. In an out-of-order context however, several instructions might be active within a single stage (e.g. EX stage). Therefore, a buffer of instructions at the entry of the Out-of-order area substitutes the latche. The buffer may have different designations but its functionality is the same: providing an area where parallel polling of instructions is possible.

Register renaming: one of the big issues of OOO execution is the handling of data dependencies. Of the three possible data dependencies (WAW, WAR and RAW), only Reads after Writes (RAW) are true dependencies, WAW (writes after writes) and WAR (writes after reads) are possible because the same register name refers to totally independent values. This is due to the fact that there are not enough register IDs in the ISA. Register Renaming is a mechanism that allows to rename register IDs in order to eliminate WAW and WAR dependencies. The renaming mechanism has been first described by Tomasulo in [91], and most modern processors use a variant of that process.

Speculation: conditional branch instructions are problematic for the out-of-order engine and the pipeline in general. Until the instruction itself is executed, the next path is not known, therefore it is not possible to know what instructions should enter the pipeline next. In such situations, either no instruction coming after a branch instruction is executed and even prefetched until the outcome of

the branch is known, or the processor speculates that one of the two paths will be taken and acts as if it was really the case. The branch prediction mechanism, called *branch predictor*, helps the prefetcher to determine which instruction stream should be fetched next. However, in the case of a bad prediction (also called mis-prediction), the execution of all instructions which arrived afte rthe branch is invalidated, the pipeline is flushed and execution resumes from the right path. Although instructions executed speculatively are not committed until the outcome of the branch instruction is determined, otherwise, it would not be possible to cancel their effects.

The out-of-order engine introduces issues also. Most notably, if the instruction window (which is a buffer) is full, it becomes impossible to execute new instructions until new slots are free. In the case of long dependency chains between instructions, with costly memory accesses, the waiting time to enter the window can be considerable for the newly decoded instructions.

2.1.5 Caches

As processors evolved and benefited from smaller circuitry, their working frequency has grown faster than the frequency with which memory circuitry operates. Indeed, the two being on different chips, processor technology grew faster than memory technology. As a result, the main memory data access latency became relatively bigger with each technology jump. This problem is commonly known as the *memory wall*[100].

In order to decrease this performance gap, small and fast memories known as *caches* were introduced. These are located between the processor and the main memory, they can be organized in several *levels* and have different possible internal *organizations* and characteristics.

2.1.5.1 Cache Hierarchies

Several levels of cache can be placed between the processor core and the main memory. caches close to the processor core are faster and smaller, those which are far are bigger and slower. Furthermore, a cache can be either *split* or *unified*. In a split cache, instructions are held in an *instructions cache*(*I-cache*) and data in a *data cache*(*D-cache*). On the other hand, a unified cache holds both instructions and data.

A data request from the core is first passed to the L1 cache (the closest to the core). If the data is found there, it is called a *cache hit*, if not, it is called a *cache miss*. In the later case, the request is passed to the L2 cache, if it is found their, then data are passed to the L1, otherwise the request is passed to the L3 cache, the process is repeated until the data are found in a lower cache level, or the request reaches the main memory, and the data is transfered from there to the L2 and L1 caches. Figure 2.3 illustrates this system.

2.1.5.2 Cache characteristics

We enumerate five questions which drive cache caracteristics:

1. *Where to put data within the cache*: we identify two extreme positions, either the data have only one location in the entire cache, or any location can be

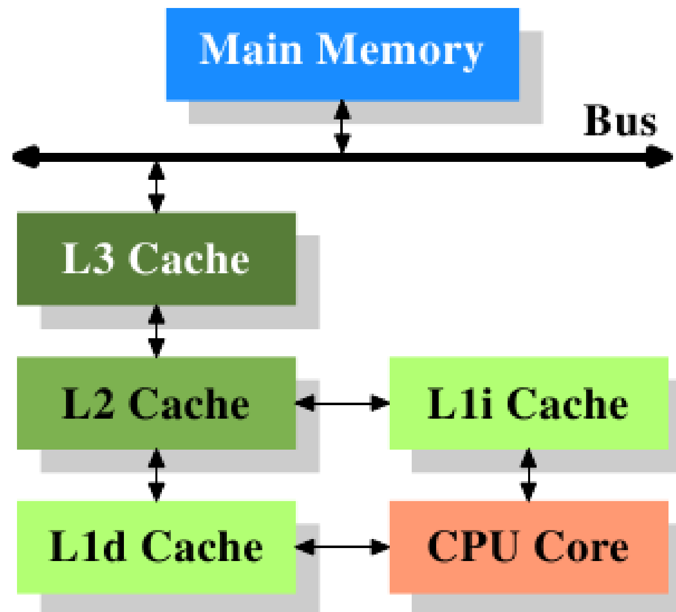


Figure 2.3: Processor with level three cache[13]

used. The former solution is known as *direct mapped* caches, and the second as *fully associative* caches. A third category, known as *set-associative* caches, is a trade-off between the two, and allows a number of caches lines to be placed in one location.

2. *What to do when the cache is full*: when the cache is full and new data need to be placed in it, then an already existing cache line should be evicted. Several eviction techniques are possible: *least recently used (LRU)*, *most recently used (MRU)*, *random replacement (RR)*, *etc.*

3. *How to identify data within the cache*: only part of the address of data is needed for its placement in cache, therefore, the rest of the address is recorded in a *tag* register tied to the cache line. More over, mechanisms to verify the validity of data within the cache lines are also needed.

4. *How to manage a case of write*: unlike a read which requires just to find and bring data, a write includes not only findind data but propagates its new value within the entire memory hierarchy. We enumerate two known write strategies: *write-back* and *write-through*.

5. *How to prefetch data chunks*: includes all the matters of predicting data that is likely to be used: how much to bring, to which cache level data should be prefetched, how many parallel prefetch streams are needed.

2.1.5.3 Data locality

Caches greatly enhance data access latencies. However, as any sophisticated hardware wiring, they constitute a limited resource, which puts more pressure on the software part in order to exploit them efficiently. We identify two important properties the software needs to take into account:

- *temporal locality:* a referenced memory location is likely to be referenced again in the near future.

- *spacial locality:* if a memory location is referenced then it is likely that a nearby location will be referenced in the near future.

2.2 Multi-core Designs

Moore´s law, a prediction dating from early 80's and stating that processor speed would double each 1.5 years was no more verified in early 2000. Processor performance increases have begun to slow down. Chip performance attained a 60 % increase per year in the 1990s but decreased to 40 % per year from 2000 to 2004, We could build a slightly faster chip, but it would cost twice the die area while gaining only a 20 percent speed increase, noted Marc Tremblay, chief architect for Sun Microsystems Scalable Systems Group. The performance increase was sustained because of the possibility to make smaller transistors, however, transistors cannot shrink forever. Even now, as transistor components grow thinner, chip manufacturers have struggled to cap power usage and heat generation, two critical problems. As a result, chip manufacturers started to build chips with multiple cores, less powerful but cooler.

Intrinsically, this added many design parameters: number of cores, symmetricity of the cores, memory organization, core interconnects and power management. Moreover, some software programs needed to be rewritten in order to take advantage of the parallelism offered by multiple processing units, raising new questions: what programming paradigm should be used, and how to transform the initial sequential algorithms into parallel algorithms, which are able to utilize efficiently the multiple processing units.

The existence of multiple cores on a single die introduces new challenges related to internal communications and common resource sharing. First, cores should be able to talk or notify each other, hence some sort of a network should be established between them. Second, in contrast with uni-cores, the external main memory is now shared between cores, which means that memory access management has to be more elaborate. Third, if two or more cores run the same program, they would share the same memory space, moreover, they are likely to access the same memory cases, which would be in their local caches. Therefore, in order to maintain a global coherent state of the memory some advanced memory coherency mechanisms are needed.

2.2.1 Multiple Cores[92]

A multi-core processor can either integrate similar cores (homogeneous), or cores with differen characteristics (heterogeneous). Most mainstream processors are homogeneous, examples include: Intel and AMD processors. Heterogeneous processors, have at least two cores with differences (e.g. ISA, functionalities, performance). IBM's Cell [54] is probably the most known example of it. In general, both organizations contain pipelined, superscalar cores with vector extensions, for a better exploitation of the ILP. Nonetheless, since in some cases it is hard to exploit such parallelism due to dependences, one can argue in the favor of structures with an increased number of less powerful cores, or else in the exploitation of what is called *Simulataneous Multithreading (SMT)*, which consists of running multiple threads on the same core at the same time.

More over, recent advances in power management make it possible to control

the frequency of each core independently. The control over core frequency change enables the user/OS to define custom power consumption policies for the programs

2.2.2 Cache Organization[21]

The multi-core dimension adds some important design decisions to take into account. Over time, the number of cache levels has increased from two levels L1 and L2 to three levels L1, L2 and L3 (also called last level cache *LLC*). Figure 2.4 gives a schematic overview of some common organizations. In general, the last level of cache is *shared* among cores, hence a core looking for a data first checks its local cache, if it does not find it, it goes down in hierarchy until it reaches the LLC. Shared caches bring several advantages: first, the available storage space can be dynamically allocated among multiple cores, second, only one copy of the data is needed within the cache, third, LLC is the level in which coherence misses (see Section 2.2.3) can be resolved. But shared caches imply also that different cores working on different data sets will interfere with each other. A data sought by multiple cores necessarily creates contention problems. In contrast, *private* caches do not generate interferences between cores on resources, and no contention over data leading to performance benefits, but have more replicated data and a mis-use of global memory available if compared with a shared cache.

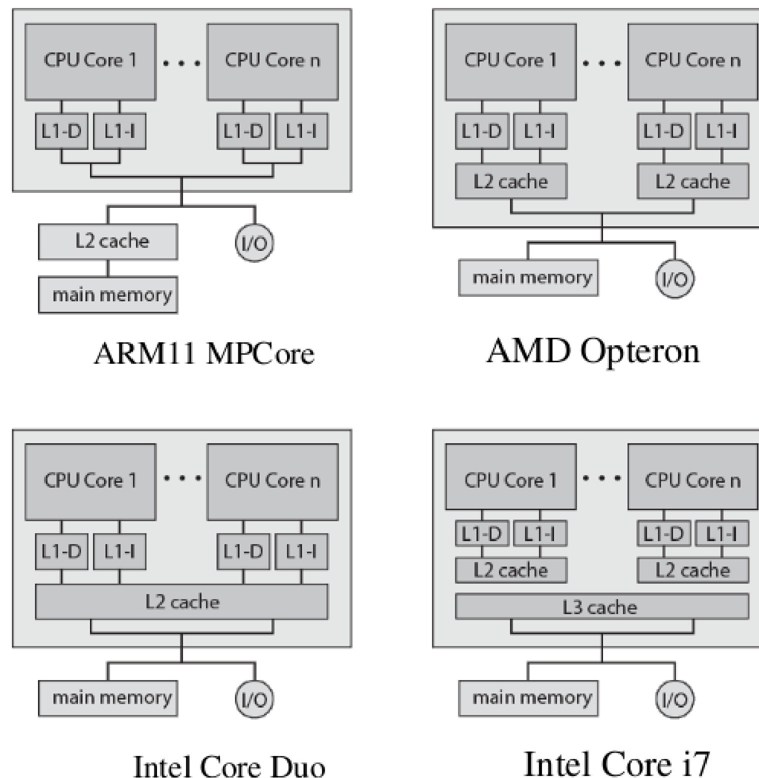


Figure 2.4: Common cache organizations, with two or three levels of cache

Also, caches can be either centralized or distributed. We can take the case of a LLC cache for example, along with its controller, it can represent a single centralized entity from which surrounding cores may extract data, or it can be

physically distributed on the chip into banks, each being close to a specific core. The combination of the core, its private caches and the LLC associated part with it is called a *tile*. An on-chip network is used to connect the tiles. This way, if a private cache of a core performs an access and fails, its request is routed through the network to the tile where the line is expected to be.

2.2.3 Shared Memory Support[89]

Shared memory support makes it possible for multiple cores to access the same shared address space, but it raises what are known as coherency problems. A coherence problem may arise when multiple cores each have a copy of a datum, in their caches for example, and at least one of them accesses it with a write. In such a case, the datum copies from the other core´s caches become stale, resulting in an incoherent situation. Incoherent situations are resolved through a set of rules implemented in different parts on the chip. This set of rules is known as *coherence protocol*. Known coherence protocols include: MESI[49], MOESI[3], MESIF[45] and DRAGON[71].

Coherency is part of memory consistency, a larger set of issues about the correctness of data inside memory. Memory consistency models attempt to define shared memory correctness. In the case of a uniprocessor, this would simply be rules about loads and stores (memory reads and writes) and how they act upon memory. Only one correct result among many incorrect ones is found. In the case of a multi-core processor with shared memory it becomes less obvious. The shared memory consistency model deals with loads and stores of multiple threads, hence multiple correct behaviors are usually allowed. Applying the rules of consistency models involves software assistance, which makes them visible to the program. Additionally, hardware assistance through what is called *coherency mechanisms and protocols* insures more performance. Though, these are neither crucial nor visible to software, they prove to be very effective and important in modern designs.

2.3 GPUs and Many-Core Designs

We briefly reviewed some special architectures composed of several processors but not referred to in the multi-core nomenclature. Classified as accelerators, they kept their own name.

2.3.1 Graphical Processing Units (GPUs)

Late years recorded the introduction of graphical processing units (GPUs) in the HPC field. GPUs are adequate and powerful in processing problems which expose data parallelism. However, they are specialized for that kind of problems, and would fail to achieve the same performance as a general purpose processor. Therefore, current supercomputers integrate GPUs as accelerators. Programmers only use them on kernels which expose a high level of data parallelism.

A GPU is composed of hundreds nay thousands of simple processor cores. The idea is that each core executes the same instruction but on a different bundle of data. The cores benefit from local private caches and distant shared ones. Finally, the main processor communicates with the GPU through the *Peripheral Component Interconnect* (PCI).

2.3.2 Many-cores

This kind of architecture has been released by Intel under the name *Intel Many Integrated Core Architecture* (Intel MIC Architecture). The architecture integrates around 50 to 60 processor cores on the same die. The cores are simpler than those we find on multi-cores, but are much more powerful and integrate 512 bits long vector registers. It is possible to either use a MIC as an accelerator, in the same way a GPU is used, or to execute the entire code on it.

2.4 Design Example: Intel Sandy-Bridge Architecture

Figure 2.5 depicts the pipeline of a processor core based on an Intel Sandy Bridge μarchitecture. All details are extracted from the official Intel optimization manual for the Sandy-Bridge architecture [7].

The Sandy-Bridge pipeline can be cut into three major areas:

- An *In-order* front end which fetches instructions and decodes them into micro-ops (micro operations). The processor being a CISC model, the decoding process transforms instructions into RISC micro-ops. The front end should deliver instructions in a continuous manner to the back end either from the true or the speculated execution path.
- An *out-of-order* engine. able to dispatch up to six micro-ops per cycle. Micro-ops are no more executed following program semantics but following data-flow, where the rename/allocate unit issues a micro-op as soon as its operands are ready and the adequate hardware resources for it are free.
- An *In-order* retirement unit which ensures that micro-ops retire in program order (semantic order).

First, a block of instructions is chosen by the branch prediction unit. The instructions start their execution in the decode block which is composed essentially of the following units:

- **Instruction cache ICACHE**: the first level cache is divided into two independent caches, one for data and the second for instructions. The dedicated instruction cache is able to fetch 16 bytes of data in each cycle.
- **Instruction pre-decode**: scans the 16 bytes chunk fetched from ICACHE in order to determine instruction boundaries. Since X86 instructions can be of variable length, the scanning process becomes more complicated.
- **Instruction decode**: Four decoding units decode instructions into micro-ops. It is worth noting that more than up to four micro-ops can be generated by a single instruction, and only one decoder is able to perform such decoding. The three others are only capable of decoding one micro-op instruction. Decoded instructions are directed both to the micro-op queue and to the decoded ICACHE. Also, decoding in Sandy-Bridge micro-architecture introduces two important concepts:
  - **Micro Fusion**: consists of fusing multiple micro-ops into a single one when possible. This has the advantage of saving bandwidth by reducing

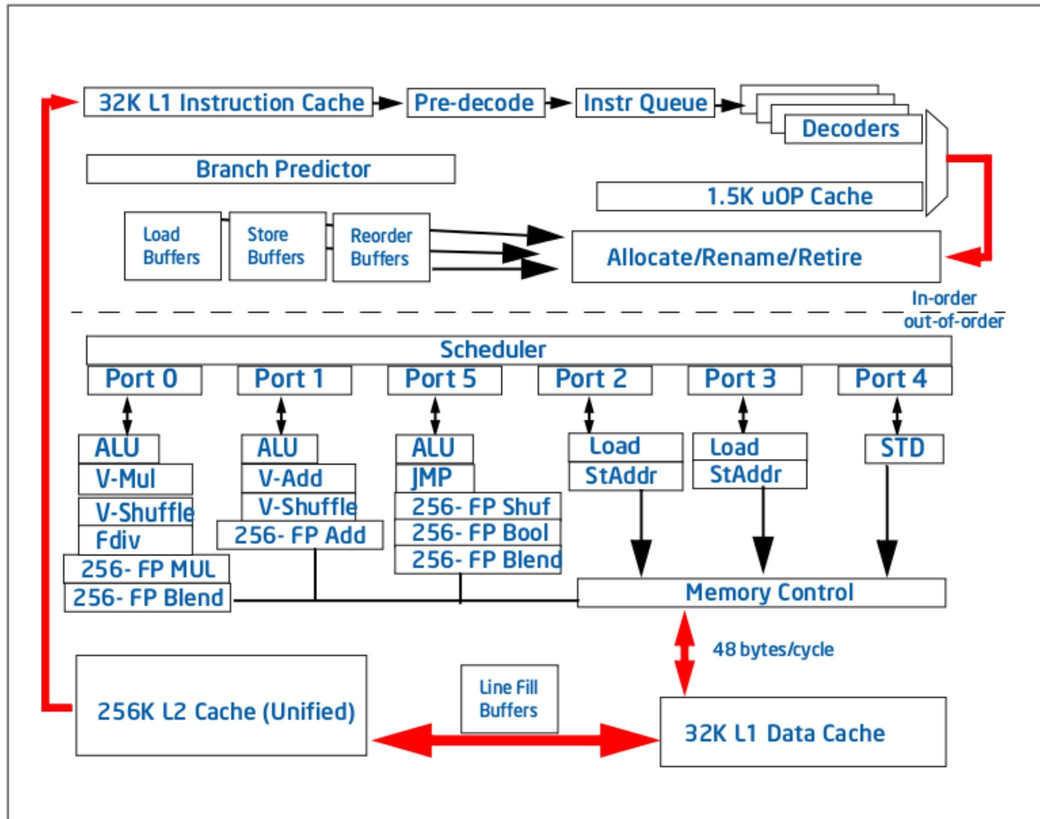


Figure 2.5: Intel Sandy Bridge μarchitecture pipeline functionality [7]

the number of micro-ops forwarded to the OOO engine. An example of micro-op fusion are instructions that combine load and computation operations.

– **Macro Fusion**: consists of merging two instructions into one micro-op until the end of their execution, this technique enables a reduction of latency, energy and hardware resources usage. Macro fusion is subject to some limitations too, for example, if the first instruction comes at the end of a cache line and the second at the beginning of the next one then macro-fusion is not possible.

- **Decoded ICACHE**: is a small cache for micro-ops which stores decoded instructions and makes them available. ICACHE is composed of 32 sets, each having 8 ways with and able to hold up to six micro-ops. Micro-op storing is also subject to some rules, therefore when an instruction cannot be stored in the cache, it is directly delivered from the decoders. ICACHE automatically brings major improvements. First energy can be saved by avoiding to fetch and decode instructions already present in it. Second, in high performance codes where the front-end with its 16 bytes bandwidth has troubles to keep feeding the back-end, the more generous ICACHE bandwidth of 32 bytes brings a big relief.

- **Micro-op queue**: acts as a bridge between the front-end and the OOO engine. The queue holds micro-ops delivered either from decoders or from the ICACHE. The queue helps to hide the disturbances that can happen in the

front-end micro-op delivery process by ensuring that four micro-ops are delivered at each cycle.

- **Loop Stream Detector (LSD)**:is a mechanism that detects small loops in the micro-op queue, once detected, the loop micro-ops are exclusively delivered by the micro-op queue, which enables to shut down the upper pipeline stages (fetch, decode and caches) for the duration of the loop.

At this point, the in-order phase is finished, now, the OOO engine takes micro-op execution in charge. Its main components are:

- **Renamer**: removes false dependencies (WAW and WAR) by renaming source and destination operands of micro-ops with internal micro-architectural sources and destinations. The renamer also offers some execution capabilities, such as delivering up to four micro-ops per cycle.
- **Scheduler**: acts by finding which micro-ops are ready for execution and dispatches them following a priority order. The scheduler pulls its micro-ops from a queue called reservation station and checks, for each operands, whether he source operands are ready or not.

- **Execution core**: in sandy-Bridge the execution core is superscalar, it consists of three execution stacks, each handling one of the following data types: general purpose integer, SIMD integer or floating point and X87 (legacy floating point unit of earlier Intel processor generations). Six execution ports are available, some are dedicated to specific tasks such as port 4 for stores or 2 and 3 for loads/address calculation, the remaining ones (0,1, and 5) are more general and handle several types of data and operations.
- **Retirement**: retires executed micro-ops in program order, and handles faults and exceptions.

2.5 Summary

In the current chapter, we highlighted the design of modern high performance processors, we addressed important components and technologies such as the processor pipeline, the out-of-order engine and memory caches. Second, we introduced multi-core architecture, along with the added complexity of core communications and cache coherency. Finally, we presented the details of the Intel Sandy-Bridge micro-architecture, a widely spread micro-architecture on which we based the majority of our work.

The performance of an application being a function of both hardware and code characteristics, this introduction sets the ground for the hardware part. Our next chapter addresses the issues software faces to overcome hardware complexity and take advantage of its power. The chapter also addresses specific issues of software itself, and offers review of application performance analysis in general.

Chapter 3

Application Performance Analysis

Scientific applications need to make the best use of processor resources in order to achieve better execution time. However, the increasing complexity of modern processors makes this task harder and harder. As a result, more elaborate performance analysis techniques and tools need to be developed and integrated into the application optimization life-cycle.

in this chapter, we review some typical well-known performance pathologies, and indicate how the already available performance analysis techniques proceed to detect them. In section 3.4, we put the light on a number of weaknesses related to bottleneck detection and impact assessment in the available analysis methods.

3.1 Performance Life-cycle

We recognize three major steps in the performance life-cycle:1) finding the locations and sources of performance issues, 2) estimating the potential performance gain following bottleneck removal, and 3) selecting and applying the chosen optimizations. The process is repeated for the new versions of the application.

Given the complexity of the encountered performance pathologies, it is difficult to fully automate the process with tools which detect bottlenecks, assess the potential gain and apply optimizations. More over, the analysis and projection phases can in general be automated but we fail to provide automated solutions for the majority of optimizations. Several factors contribute to this limitation, the three most important being first the lack of analysis tool to provide feedback on the high level code structures involved in performance bottlenecks, second the nature of optimizations themselves which need deep code changes, and finally the quality of the written code which often is poor.

Our work focuses on improving and automating the performance analysis and bottleneck detection step. The later can be divided into two major phases:

- *Bottleneck location identification*: consists of performing a characterization of the application (or a sub part of it) in order to find bottlenecks locations. The parameters of the characterization depend on the granularity. At node level, we primarily seek to separate loop nests and identify the characteristics of each of them. The characterization parameters can be: branches, memory accesses, data location, compute units use. If the parameters are well chosen, they may give a hint on the nature of bottlenecks the phase suffers from.

- *Bottleneck cause detection*: consists, for a given program phase (or code), to determine what is/are the main factor(s) limiting it from performing better. The goal is to tie the bottleneck to its source, for example finding that cache misses are responsible for performance degradation would not be of a help if we cannot identify the array(s) involved.

3.2 Performance Evaluation

Performance evaluation is an integral part of the performance analysis cycle. In an evaluation process, it is important to choose the right technique in order to ensure a quick detection of performance pathologies.

3.2.1 Techniques

Performance analysis techniques can be divided into three categories: *measurement, simulation* and *modeling*. Despite the fact that the categories are distinct and independent from each other, they can sometimes be complementary, hence allowing to build hybrid techniques.

A brief description of each of the three categories is provided below:

3.2.1.1 Measurement

Measurement includes techniques which monitor the various events generated during the execution of a program. The events can be of various natures: execution time, memory references, hardware counters, *etc.* Moreover, the measures of an event can be obtained with two different measurement methods:

- *Tracing*: probes are placed at the entry and exit of a code area in order to measure all occurrences generated by a set of events within the delimited area. Tracing has the advantage of being precise, but has a limit in monitoring small code regions due to the overhead introduced by the probes. As it constitutes also the choice for our measurements, we provide in Chapter 7 a study on the quality of our measures.

- *Sampling*: consists of taking event values only at particular points during program execution. The points can be either defined as time intervals, particular points in the code, event counts (threshold) or a combination of the three. In this method, the probe is located outside of the application code, it is reached through an interrupt when a sampling point is encountered. Compared to tracing, sampling is lightweight because it generates less overhead, however, it also is less precise. Its precision can be controlled to some extent by either increasing or decreasing the sampling frequency. Decreased frequency provides low precision and low overhead, whereas increased frequency provides better precision but adds considerable overhead.

3.2.1.2 Modeling

Consists of mathematical modeling of performance and design issues. Modeling has many uses: understanding how an application performance will scale given different problem sizes [68], predicting how an application will perform on other architectures [68], designing and exploring micro-architecture [72, 41, 55], managing energy consumption [56].

Within a performance analysis context, modeling is mostly used in application characterization, because it enables to compute performance without re-simulation [78], additionally, model parameters enable fast projections and more insight. However, it presents limitations too, and for some very complex problems the established model proves to be inaccurate.

3.2.1.3 Simulation

Simulation includes performance analysis techniques built on top of computer architecture simulators. A simulator aims at reproducing the entire internal working of the hardware (and optionally OS), thus it allows to have a full control over what to observe. Known simulators include: Simplescalar [27] and simic[67].

Simulations of a sub-part of the hardware are also common. Complex issues related to memory behaviour are for example investigated through a simulation of the memory subsystem, where a trace of memory references is given in entry to the simulator.

Simulation may be performed at different accuracy levels depending on the goals. It is often combined with modeling to produce fast simulators. We distinguish the following trade-offs[14] in the design of a simulator:

- *Full system vs micro-architecture simulators*: micro-arcihtecture simulators only model the micro-architecture, whereas full system simulators model not only the micro-architecture but the OS as well; they offer a more complete picture but are slower.

- *Functional vs performance simulators*: performance simulators provide a more accurate modeling of the architecture but are slower than functional simulators.

- *Trace driven vs execution driven*: trace driven simulators run pre-recorded traces of instructions, which allow a deterministic simulation, whereas execution driven simulators allow the exploration of speculative execution and OS effects.

Simulation is used for different purposes such as micro-architecture design and exploration, debugging and performance analysis. In the later category, simulation techniques face the following challenges:

- Cycle accurate simulation is considerably slower than native execution. Whereas simulating a single CPU is thousands of times slower, multiprocessor simulation is up to a million times slower [99]. Still, Several works targeted the reduction of simulation time .

- Hardware designers omit to provide the full specifications of their designs, this directly echoes on the design of the simulators. The tool may give more or less trustworthy results.

3.2.2 Performance Pathologies and detection methods

We are interested in our study on node level performance bottlenecks. Table 5.1 provides a list of typical performance pathologies of different causes. We generally use the term pathology to designate an abnormal situation which may or may not be a limiting factor for performance. The list represents a base ground on which we can discuss the abilities of bottleneck detection techniques and tools.

3.2.2.1 Static Analysis

Static inspection of the code can be efficient in finding a number of performance pathologies. The advantage of static analysis is that it is cheap compared to dynamic

No	Pathology	Detail
		CPU
1	ADD/MUL balance	ADD/MUL parallel execution (of fused multiply add unit) underused
2	Non pipelined execution units	Presence of non pipelined instructions: div, sqrt
3	Lack of loop unrolling	Significant loop overhead, lack of instruction-level parallelism
4	Short loop trip count	Significant loop overhead, control instructions are costly
5	Vectorization	Unvectorized loop
6	Complex control flow graph in innermost loops	Prevents loop vectorization
7	code alignment	If the start of a short loop is not aligned for example, it may lead to performance loss. Same goes for succesive branches (See [50]).
8	Full buffers	Some critical buffers within the micro processor lead to performance losses if they are full because of the pipeline stalls they induce.
		Memory
9	Unaligned memory access	Presence of vector-unaligned load/store instructions
10	Bad spatial locality and/or non stride 1	Loss of bandwidth and cache space
11	Bad temporal locality	Loss of perf. due to avoidable capacity misses
12	4K aliasing	Unneeded serialization of memory accesses
13	Associativity conflict	Loss of performance due to avoidable conflict misses
14	High number of memory streams	Too many streams for hardware prefetcher or conflict miss issues
		Multi-thread
15	Load unbalance	Loss of parallel perf. due to waiting nodes
16	Lock waiting	Loss of performance , due to some threads spin waiting for locks
17	Bad affinity	Loss of parallel perf. due to conflict for shared resources
		Multi-thread - Memory
18	False sharing	Loss of bandwidth due to coherence traffic and higher latency access
19	Cache leaking	Loss of bandwidth and cache space due to poor physical-virtual mapping

Table 3.1: A few typical performance pathologies.

analysis in terms of time. It fails, however, to assess the performance loss. A number of static analysis tools exist, among which we cite: IACA[9], MAO[50] and CQA[35]. The later is part of the MAQAO framework on which we base our tools and is also used in the performance analysis methodology we introduce in Chapter 5.

Table 3.2 indicates how static analysis techniques detect some of the pathologies we enumerated in Table 5.1.

Pathologies	diagnoses
2,3,5,6,7	Can be detected through a simple analysis of the generated code.
3,5,6	These can be returned by the compiler as feedback on the optimizations it performed (or failed to). An example of it is the `opt-report` option of the ICC compiler[10]

Table 3.2: Performance pathologies detection with static analysis techniques.

3.2.2.2 Simulation Techniques

Simulation techniques are powerful and succeed in detecting the majority if not all of the pathologies. However, some pathologies either can be investigated through a cheaper technique or are not critical enough to require the use of a simulator. We summarize in Table 3.3 the issues most frequently targeted by simulation.

Pathologies	diagnoses
8,	Can be investigated with a detailed simulation of the flow of instructions within the micro-architecture.
10, 11, 12, 13, 14 17,18	Can be investigated through a simulation of the memory subsystem behavior. A trace of memory references from the program is introduced into the simulator. [17] [2].

Table 3.3: Performance pathologies diagnosis with simulation techniques

3.2.2.3 Dynamic time profiling

Measurement techniques based on time profiles are useful to diagnose a number of pathologies. Profiling consists of instrumenting the code in order to be able, at runtime, to capture program behavior.

Table 3.4 indicates how some of the pathologies introduced in Table 5.1 are diagnosed through dynamic code profiling.

Pathologies	Diagnoses
15, 16	sampling and attribution of idleness to causes[65]
17	source instrumentation or sampling of parallel regions

Table 3.4: Performance pathologies diagnosis with dynamic time profiling techniques.

3.2.2.4 Hardware Performance Counters Based Techniques

For two decades, hardware counters/events have attracted a large deal of attention [73]. Many tools and analysis methods rely on them to analyze intra-node related

performance issues. They are especially helpful to debug fine grain pathologies since they have the ability to monitor various events occurring within the processor at very low overhead and without altering the numerical output of the system. Hardware counters are consequently used in a large number of tools; they now are natively supported in the Linux kernel since the version 2.6 through the *perf* subsystem [37].

Table 5.1 can be diagnosed with hardware counters. We can describe mildly the types of counters involved in such process in Table 3.5

Pathologies	**diagnoses**
1,2	Counters that monitor the number of `uops` dispatched in each execution unit
8	Counters that count stalls due to full buffer (e.g. `RESOURCE_STALLS_ROB` or `LD_BLOCKS` on Sandy-Bridge)
17	Counters that count the occurrences when a load hits a modified cache line in another core (e.g. counter `MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS` on Sandy-bridge [8])
12	Counters that count occurrences of load blocked by preceding stores (e.g. `LOAD_BLOCK.OVERLAP_STORE` [8])

Table 3.5: Performance pathologies detection with hardware counters based techniques

We also make the following observations on the limitations, hardware counters based techniques suffer from:

- Hardware events are excellent at capturing how a given piece of hardware is used but, very often, they fail in evaluating the exact performance impact: for example, hardware events can count cache misses but what matters is not the number of cache misses but the total impact of cache misses on performance (i.e. the product of the number of cache misses and the average cost of cache misses. In general, hardware counters fail at evaluating accurately such average costs).

- Another difficulty with hardware counters is the correlation between source code and hardware event counts. Hardware counters can be triggered on and off at the beginning and the end of a loop structure or a function but the results are globally reported for the whole loop/function and not for individual source code statements.

- One final difficulty with hardware performance events is their complexity. First, their number is fairly high (in general over a thousand) making them hard to use. Secondly, many of them refer to low-level microarchitectural details which are not publicly available making counter information hard to decipher. For example, knowing that the reservation station is full and generates partial stalls in the front end pipeline does not give a precise clue of what to do to optimize the code.

3.3 Performance Evaluation Tools

Performance analysis tools are used to investigate and understand the precises causes of bad performance for a particular piece of code on a particular platform. They generally integrate one or several techniques among those we cited in the previous paragraph. They also focus either on a specific granularity or handle several. Some of the well established and effective tools are:

Scalasca: Scalasca [44] mainly focuses on MPI programs and is very efficient for quickly identifying communication problems such as late sender-early receiver. For OpenMP programs, Scalasca can identify load balancing issues in data parallel loops and synchronization issues. Scalasca uses source level instrumentation which is well suited for the communication problems listed above (minimal interference with the compiler). However for more general performance bottlenecks, Scalasca does not provide any specific exploration technique besides hardware performance counters.

TAU: Tuning and Analysis Utilities (TAU) Performance System [86, 85] is a performance profiling and tracing framework. As such, it offers much more flexibility in the performance investigation techniques than Scalasca. The TAU framework addresses performance problems on three levels: instrumentation, measurement, and analysis. It provides instrumentation at different levels and performs tracing on parallel programs. Although TAU offers many possibilities of using (inserting / triggering) various performance counters, it basically inherits all the key limitations of hardware performance counters and in many cases, it will not be very helpful for performance bottleneck investigation.

PerfExpert: [28] goes a step further by trying to analyze performance bottlenecks and provide optimization guidelines. Again, it mostly relies on hardware performance counters to evaluate performance problems and suffers from the same problems as the others. However, the approach of synthetizing hardware counter information to derive performance optimization is very powerful.

XE Amplifier: XE Amplifier [4] is an Intel tool for performance analysis. It has different features, including stack sampling, thread analysis and hardware event sampling. Some traditional features, such as identifying the hottest modules and functions in a whole application or tracking call sequences, are also supported. XE Amplifier leverages hardware counters for in-depth analysis of the memory system and architectural tuning and associates performance issues with the source code. If no symbol sources are found in the binary, XE Amplifier navigates through the disassembled code at a basic bloc granularity.

Cachegrind: Cachegrind [17] is the cache profiler included in the Valgrind instrumentation framework. When using Valgrind, the original instructions never run on the host processor. Instead, Valgrind converts instructions on-the-fly to an intermediate representation. Valgrind companion tools can easily and directly manipulate the intermediate representation which is then recompiled for the target architecture. Cachegrind is based on the simulation of configurable L1I, L1D, and L2 caches. It identifies the number of cache misses for each line of the source code,

with per-function, per-module and whole-program summaries.

ThreadSpotter: Acumem AG [2] offers the commercial product ThreadSpotter specially targeted at analyzing data access issues. It relies on a statistical analysis of address traces to estimate various potential problems with data access (stride, false sharing).

Maqao: Maqao [12] is a performance analysis framework that works at binary level. By disassembling the binary and building an intermediate representation, Maqao offers the base ground to build analysis tools in the form of modules. Existing modules include: a profiling tool for sequential and parallel programs [31], a memory tracing library called Mtl, a static analysis tool called Cqa [35] as well as the tool we develop and work on within our study Decan [58].

3.4 Discussion

Our review of bottleneck detection techniques attracted our attention on some missing areas in the performance analysis cycle that we believe need to be filled. Below are the important points we focused on

Better bottleneck detection

We noticed that the majority of bottleneck detection methods rather provide more a pathology detection than a bottleneck detection. Indeed, highlighting a bottleneck means assessing the importance of the pathology and of its impact on performance, in other words its *cost*. However, the majority of presently known methods fail to deliver accurate information. Hardware counters based techniques provide an information on the quantity of an event rather than on on its cost. Simulation techniques are technically capable of providing the cost such as in [43] but are rarely used for that purpose because the analysis time is too high. Time profiling, on the other hand, is able to assess the cost only if sufficient precision is reached; however, loop level bottlenecks (especially innermost loops) require a level of precision that classical profilers are unable to reach. The reason for this is that instrumenting the code inside a loop may heavily alter its behavior, and instrumentation results would be biased. Therefore, we identify a clear need in a better bottleneck detection through pathology *cost* assessment.

Coordinate the application analysis process

performance characterization consists of finding the parts of code on which optimization effort should be spent, successive characterization steps with a lower analysis granularity at each new step should lead at the end to the identification of the bottlenecks. We then identify the need for an analysis approach which would be based on successive steps of characterization leading to bottleneck detection. At each step, numerous tools might be used. Consequently, a clear analysis methodology would greatly help to answer the recurrent questions of: what tool to use and when to use it.

3.5 summary

Within this chapter we emphasized our interest on application performance analysis issues. We particularly focused on on-core performance pathology detection and performance characterization by reviewing well known pathologies as well as the methods used to investigate them. Finally, we enumerated some flaws within the actual methods and analysis process in general. The next chapter explores the tools and methods we strived to establish in order to complete the performance analysis process.

Chapter 4

Differential Analysis

4.1 Introduction

There is still more room for yet another performance analysis method and tool. Our review of performance analysis techniques in the previous chapter enabled us to draw the following observations:

- Failure to order pathologies following to their costs. Thus, it is less obvious to assess if a pathology effectively is a bottleneck.
- Out-of-order execution on modern micro-processors allows several bottlenecks to have parallel effects. For example, a costly memory reference may overlap with a long latency instruction which will mask its effect. These cases are important to catch in order to focus on the most important bottlenecks.
- The cost of a bottleneck helps to determine the potential gain which would result from its optimization. Thus, it helps to order bottlenecks following to their *return on investment (ROI)* ratio.

A novel technique called *Decremental Analysis*[57] addresses these concerns. Decremental analysis consists in building the performance breakdown of a loop by progressively pinpointing bottlenecks. The analysis is based on a binary transformation tool called Decan [57].

Decan targeted innermost loops at binary level. The tool operated by creating, for a given loop, a version in which specific assembly instructions are suppressed. The performance of the created version, called *variant*, is then compared with those of the original version in order to assess the cost of the instruction. By modifying loop instructions, Decan breaks the semantic of the code. Therefore, in order to avoid random effects during runtime, the tool does not monitor loop performance into its original context, but extracts it into an isolated kernel.

However, Decremental analysis provides the base idea, and needs further refinement. We think that the concept can be pushed further to address a bigger set of performance pathologies. On the tool level, the early version of Decan suffered from several limitations which prevented it from supporting complex codes and providing trustworthy results.

In our work we engaged both in the development of Decan as a tool and *Decremental Analysis* as an analysis technique. The following contributions are detailed in this chapter:

- We designed new Decan variants, which we tested and validated. These are detailed in Section 4.4.
- At the tool level, we brought several structural changes and improvements to Decan, these are briefly reviewed in this chapter, but benefit from more details in Chapters 6 and 7.

- At the analysis level, we broadened *Decremental Analysis* into a larger concept that we named *Differential Analysis* through analyses based on a richer variants catalogue and new use cases. These are detailed in Sections 4.5 and 4.6.

4.2 Motivating Example

A brief study of the following example (extracted from a dense Singular Value Decomposition library) [82] will let us explore some of the concerns raised in the previous section and introduce Differential Analysis.

Algorithm 1 Code example

```
real*8 A(N,16), scal, s(16) {Column oriented storage}
DO i = 1,16 (Parallel loop)
   DO k= 1, N
     A(k,i) = A(k,i)/scal
     s(i) = s(i) + A(k,i) * A(k,i)
   ENDDO
ENDDO
```

Let us consider the piece of code shown in Algorithm 1. We consider different values of N ranging from 200 up to 1000K. The target machine is a 4-core Sandy-Bridge architecture. The parallelization of the outermost loop is ideal and results in a perfect load balance between the cores. The innermost loop contains a stride 1 access to an array. It results in a vectorized code. The only two potential issues are: 1) the reduction operation and 2) the division operation which is very costly on Sandy-Bridge architectures. The division operation can not be factored out of the innermost loop because the use of a reciprocal operation followed by a multiply affects numerical properties.

First, DECAN is used to generate two binary variants to detect, for which data range, the loop is CPU bound or data access bound:

- `LS` is a binary variant in which all of the floating-point (FP) arithmetic operations have been suppressed: only the data access instructions and the address/loop instructions have been kept.
- `FP` is a binary variant in which all of the load/store operations have been suppressed: only the FP instructions and the address/loop instructions have been kept.

Instructions wich contain both memory and arithmetic operations are transformed in a way that keeps only one operation.

By measuring `LS` and `FP` we evaluate the contribution of data access and arithmetic instructions to the overall execution time. Figure 4.1 shows that for values of N less than 400K, the bottleneck is the arithmetic operations, while for N values greater than 400K, the bottleneck is the data access.

To investigate the performance impact of reduction and division operations, two more variants were generated:

- `NO_DIV` is a binary variant in which only the FP division is suppressed.

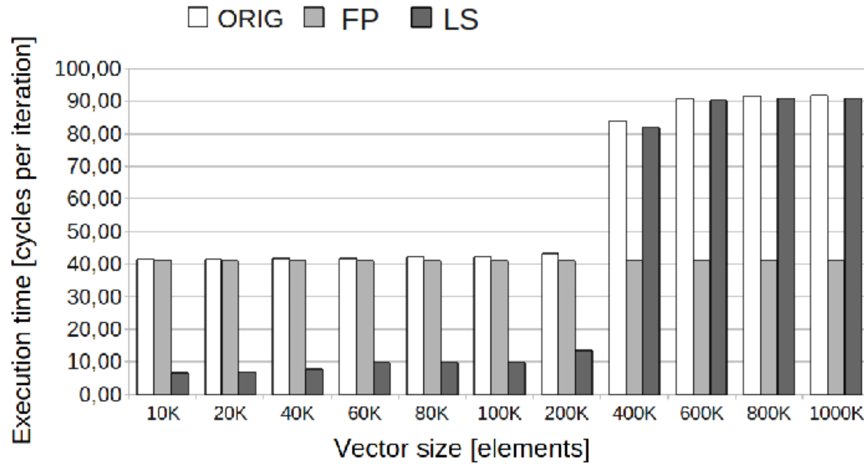


Figure 4.1: Memory and Floating-point streams analysis with the variants `LS` and `FP`. The experiments are performed on 4 cores

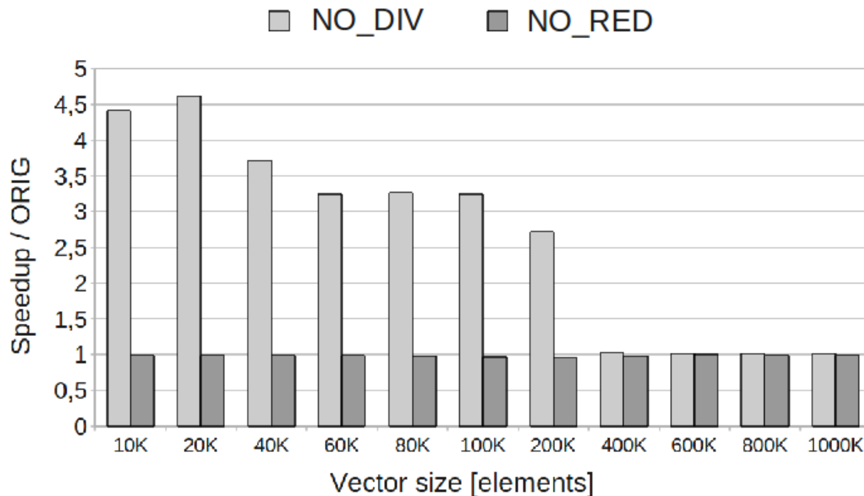


Figure 4.2: Division and reduction impact analysis with the variants `NO_DIV` and `NO_RED`. The experiments are performed on 4 cores

- `NO_RED` is a binary variant in which the dependencies between iterations relative to the addition (second statement of the original loop) are suppressed.

Figure 4.2 presents the relative performance gains of these variants with respect to the original. First, it clearly shows that the reduction operation induces no performance penalties across the whole data range (`NO_RED` performance is identical to the original). Second, the division operation is very costly (i.e. it is the main performance bottleneck) for values of N less than 400K, while the division operation cost is hidden by the data access for larger N values. Therefore, the technique of keeping the division operation within the loop to increase numerical accuracy had a performance impact only for the smaller values of N. A third variant, `NO_RED_NO_DIV`, not shown here, was generated to detect potential interaction between the division and the reduction operations. This third variant had exactly the same performance as the `NO_DIV` variant. This indicates that no interaction is present between them.

With this example we find ourselves in a typical issue of loop level bottleneck detection. First, since memory and arithmetic operations are executed in parallel streams, it is important to know if one of the two dominates the execution. DECAN enabled us to quickly verify it by isolating the two streams and by assessing the proper cost of each one of them. We thus could see two phases: a compute bound phase in small and middle data sizes and a memory bound one for big data sets. We could easily conclude that the memory bound phase was due to data being in RAM. However, we did not know how to interpret the compute phase. Within the code we saw two clear pathologies: divisions and reductions. The following solution were considered:

- the impact of the division operation could have been achieved at the source code level by suppressing it, but the compiler could have then suppressed the original statement from the generated code.
- Another approach is to replace the division by a multiply operation, yet the compiler may still generate an altered code. Operating at the binary level allows "surgical" operations (with minimal intrusion) to be performed, keeping variant code very close to the original target code.
- Suppressing reductions at the source level is much more challenging and in most cases would result in a code very different from the original one.
- Using an analysis tool. The most suitable seemed to be XE Amplifier, because it integrated a feature which pinpointed costly instructions within loops. The problem is that the feature is based on sampling, thus usually fails to attribute the cost to the right instruction.

DECAN enabled us to quickly verify which of the two pathologies was a real bottleneck. We could verify in a reasonable amount of time that the division operation was the main bottleneck. In general DECAN is able to verify a pathology with one variant.

The example studied above allowed us to review DECAN as a tool which allows to create modified version of binary loops called variants. It also enabled us to introduce the analyses which use the variants in order to diagnose performance pathologies.

4.3 DECAN: Practical Design

We review within this section the general concept of DECAN, and summarize the important features we introduced into the tool. The features are detailed separately throughout the manuscript.

4.3.1 Principle

DECAN as a tool modifies parts of a program binary which results into a new binary; it then runs the modified binary to compare various performance metrics with those obtained for the *original* (unmodified) binary. These modified binaries are called *loop variants* or in short *variants*. Variants are **not** semantically equivalent to the original binary: in particular the memory state after running a DECAN variant of a loop is not the same as after running the original version.

4.3.2 DECAN Target

The primary target for DECAN are assembly instructions of innermost binary loops. For a given innermost loop, DECAN can generate several modified binaries according to various transformation rules previously specified.

4.3.2.1 Characteristics of the targeted loops

The binary loops targeted by DECAN have the following characteristics:

- The binary loop must be reducible (have a single entry point but may have multiple exit points). Irreducible loops are non-standard constructs which are difficult to handle by the majority of analysis tools. In practice, over 95% of the innermost loops have a single entry point.

- The loop body can contain several basic blocks and, apriori, can have an arbitrary control flow, except nested loops which are not supported. Regular, one path loops, benefit from a better handling, as we found to be the major hot loops in the applications we studied.

4.3.2.2 Characteristics of the targeted assembly instructions

The current implementation of DECAN handles the Intel X86 *instruction set architecture (ISA)*. Future developments should integrate more ISAs. The implementation has the following characteristics:

- The majority of instructions especially those used in arithmetic calculations and memory operations are integrated. Are of a special interest also vector extensions, where the entire set of SSE and AVX instructions is handled. New instructions are added following the goals of the variants.

- X87 (in particular, all of the instructions manipulating the FP stack) instructions are left untouched/unmodified by DECAN. Therefore, for loops consisting of only X87 instructions, DECAN will correctly run but the variants generated by DECAN will be of little interest.

- Pointer, indirect addressing and complex structures are fully supported by DECAN. They generate instructions on which DECAN can fully operate

4.3.3 Variants

A loop variant is a version of the loop in which assembly instructions have been modified. Two main concepts are involved in the process:

- *Instruction subsets*: instructions are grouped into what we call *subsets*, these are created following specific criteria.

- *Transformations*: a set of transformations that can be applied on assembly instructions. These include: *suppression*, *modification*, *replacement* and *addition*.

A loop variant is created by applying transformations on instruction subsets created from the instructions of the loop.

4.3.4 Semantic Alteration

Decan performs what we qualify as a *controlled alteration* of the code. At runtime, the control flow of the application risks to be changed, a case which should be completely avoided. The earlier version of Decan applied what we call *in-vitro* mode, whereas in our version we applied an *in-vivo* mode. Section 6.3 in Chapter 6 discusses these issues more in depth. Although, we review here the important points of the in-vivo mode we use:

- For any loop, Decan will first identify all the instructions involved (necessary for) in the control flow and these instructions will be blacklisted meaning left untouched by all Decan transformations. Therefore, the control flow of innermost loops is always preserved.
- In order to avoid altering the control flow of the application, Decan applies the *instance mode*. This mode consists of activating the modified version of the loop only for one loop call. The program is normally executed, when the loop call is reached, the modified version of the loop becomes active instead of the original version, the program is ended just after the loop finishes its execution. By applying the process on a number of representative loop calls, a precise image on loop performance can be constructed.

4.3.5 Performance Monitoring

The comparison between variants is based on execution time. Decan is able to monitor the performance of the transformed loops with high accuracy. This is done by injecting probes at the entry and exits of the binary loops. The process enables to compare with precision only the loops which have been modified.

4.3.6 Parallel Codes

Our version of Decan adds support for parallel programs as well. The support is detailed in Section 6.4 in Chapter 6. We review below the important aspects:

- In the case of shared memory models, Decan handles OpenMP based codes. It particularly handles the *parallel for* constructs, where each thread executes the same transformed version of the loop. The only difference with the sequential case is that each thread will have its own probes and performance report.
- In the case of distributed memory models, Decan handles MPI based codes. Only innermost loops without `MPI_send` and `MPI_receive` calls are supported, all processes execute the same transformed version of the loop, but each have its own performance report.

4.4 Decan Variants Design

Decan variants are created by first identifying instruction subsets and then applying the transformations. During our study we defined the concepts of instruction subsets and transformations with the goal of providing a more abstract view, which would be reused when implementations for other ISAs are developed in Decan. We first enumerate instruction subsets and then describe the transformations we developed during our study. Finally, we introduce the variants we developed.

4.4.1 Instruction Subsets

The first step in the transformation process identifies subsets of instructions to transform. We distinguish two operative modes in the decision making for instruction subset creation: *local view* and *global view*.

We use the assembly code shown in Table 4.1 as example from which we extract instruction subsets. The assembly code correspond to the loop in Algorithm 1.

	Assembly code	
	LOOP:	
0	MOVAPS	(%RAX,%RCX,8),%XMM4
1	MOVAPS	0x10(%RAX,%RCX,8),%XMM5
2	DIVPD	%XMM1,%XMM4
3	DIVPD	%XMM1,%XMM5
4	MOVAPS	0x20(%RAX,%RCX,8),%XMM6
5	MOVAPS	0x30(%RAX,%RCX,8),%XMM7
6	DIVPD	%XMM1,%XMM6
7	DIVPD	%XMM1,%XMM7
8	MOVAPS	%XMM4,(%RAX,%RCX,8)
9	MULPD	%XMM4,%XMM4
10	MOVAPS	%XMM5,0x10(%RAX,%RCX,8)
11	MULPD	%XMM5,%XMM5
12	ADDPD	%XMM4,%XMM3
13	ADDPD	%XMM5,%XMM2
14	MOVAPS	%XMM6,0x20(%RAX,%RCX,8)
15	MULPD	%XMM6,%XMM6
16	MOVAPS	%XMM7,0x30(%RAX,%RCX,8)
17	ADD	$ 0x8,%RCX
18	MULPD	%XMM7,%XMM7
19	ADDPD	%XMM6,%XMM3
20	ADDPD	%XMM7,%XMM2
21	CMP	%R14,%RCX
22	JB	LOOP

Table 4.1: Assembly code of NR codelet SVDCMP_13 (SSE version).

4.4.1.1 Instruction subsets created through a local view

The decision to include an instruction in a subset only depends on information collected on the instruction itself, no global analysis result or knowledge is taken into account. The following notable instruction groups can be constructed:

L (Load subset) Corresponds to the set of instructions involved in *load* operations. That is, all instructions having one of their *source* operands coming from memory. The L subset that can be constructed from the assembly example would contain the following instructions: L={0, 1, 4, 5}

S (Store subset) Corresponds to the set of instructions involved in *store* operations. That is, all the SSE/AVX instructions having one of their *destination* operands going to memory. The S subset that can be constructed from the

assembly example would contain the following instructions: S={8, 10, 14, 16}

LS (Load-Store subset) Corresponds to the set of instructions involved in memory (*loads & stores*) operations. We can notice how this subset is the result of the union of the L and S subsets. Therefore, LS subset in the example code is: LS={0, 1, 4, 5, 8, 10, 14, 16}

FP (Floating-Point subset) Corresponds to the set of instructions involved in floating point computations. That is, all instructions which generate a micro-operation that would be executed by an FP unit. The FP subset identified from the example code is: FP={2, 3, 6, 7, 9, 11, 12, 13, 15, 18, 19, 20}

FP-DIV (FP Division subset) Corresponds to the set instructions involved in floating-point division operations. That is, all instructions which generate a micro-operation that would be executed by the FP division unit. The FP-DIV subset identified from the example code is: FP={2, 3, 6, 7}

4.4.1.2 Instruction subsets created through a global view

The decision to include an instruction in a subset depends on information collected with a global analysis on the loop. Usually, it is done through a static analysis on the loop. The following notable instruction groups can be constructed:

RED (Reduction subset) Corresponds to the set of instructions involved in a reduction operation. Reduction operations are identified as being long instruction dependence chains which store their final result in the same location for all iterations. In our example, if we look to the source code of Algorithm 1 we would see only one reduction operation, the accumulator being variable *s*. On the other hand, at assembly level (see Table 4.1), we notice that the code has been unrolled four times, the reduction is accumulated on two registers *%XMM2* and *%XMM3* which will be merged after the loop to construct the variable *s*. In this case, we don´t have just one but two RED subsets, these are: RED_1 = {0, 2, 8, 9, 12, 4, 6, 15, 19 }, RED_2 = {1, 3, 11, 13, 5, 7, 18, 20}.

CTRL subset Corresponds to the set of instructions involved in the control of the looping process. That is, all instructions that participate in the definition of the control flow of the loop. These are grouped by detecting in the *data dependencies graph (DDG)* instructions of the dependence chains which ends are instructions which have exit edges in the *control flow graph (CFG)*. On the example in Table 4.1 this would correspond to instruction *21*, and the resulting subset would be: CTRL={21, 17}.

GR (group) subset Corresponds to the set of instructions that are part of a single data structure. A group is a set of memory accesses to the same data structure. The structure is usually an array, but it can also be a memory area used for spill-fill. Two instructions are considered to belong to the same group if they target an address using the same base and index register values, the only difference being the offset.

As shown in the example depicted in Table 4.2, grouping analysis requires a partial knowledge of the execution context, which is evaluated through advanced data-flow analysis. Indeed, for each used register or memory address, an internal representation is used to keep its possible formal values. Because the analysis exclusively relies on static analysis, it is not always possible to fully detect the entire data structure, in other words, several instruction groups would contain a fragment of it. However, it establishes a first step toward data structure detection. This static detection is completed with a dynamic one (discussed in Section 4.5.6) in order for the analysis to be able to detect entire array data structures.

	Assembly code	**Groups**
0	LOOP:	
1	MOVSS **(%RDI, %R8, 4)**, %XMM0	1 → G1
2	ADDSS **12(%RDI, %R8, 4)**, %XMM0	2 → G1
3	ADDSS **24(%RDI, %R8, 4)**, %XMM0	3 → G1
4	MOVSS %XMM0, *12(%RDX, %R8, 4)*	4 → G2
5	INC %R8	
6	CMP %R9, %R8	
7	JB LOOP	

Table 4.2: Advanced data-flow analysis tracks the symbolic values of registers. Based on registers value it is then possible to infer which instructions are targeting the same data structure. Such instructions are coalesced within *groups*

4.4.2 Transformations

DECAN transforms assembly instructions in one of the following ways:

Deletion: Corresponds to a complete suppression of the instruction.

Modification: Consists of modifying the instruction operands following multiple modification forms:

- **operands order change**: The order of operands is changed. As an example we can take the store instruction *8* of Table 4.1 and transform it into a load instruction. A simple *operand order* change would give the wanted result as shown in Table 4.3.

Original instruction	MOVAPS %XMM4,(%RAX,%RCX,8)
Modified instruction	MOVAPS (%RAX,%RCX,8), %XMM4

Table 4.3: Operands order change for a store instruction (the instruction is transformed into a load).

- **operand deletion**: some of the operands of the instruction are deleted. Table 4.4 illustrates a vector multiplication instruction in which the memory operand (source operand) is deleted. However, two operands (a source and a destination) are required for the instruction to be valid, we copy the destination operand and inject it as a source also.

Original instruction	MULPS (%RAX,%RCX,8), %XMM4
Modified instruction	MULPS %XMM4, %XMM4

Table 4.4: Source operand deletion of a vector multiplication instruction (the instruction is transformed into a load).

Replacement: The instruction is replaced by another one. This is especially helpful when an instruction is part of more than one subset. We can imagine the case of a vector multiplication instruction `mulps` having a memory reference as its source operand; and a transformation process in which only LS subset instructions are kept, a possible solution is to replace the `mulps` by a `movaps` instruction and keep its operands unchanged as shown in Table 4.6.

Original instruction	MULPS	(%RAX,%RCX,8), %XMM4
Modified instruction	MOVAPS	(%RAX,%RCX,8), %XMM4

Table 4.5: Vector multiplication instruction with a load operand transformed into a simple load instruction.

Addition: One or more instructions are added to a specific instruction; we recall how the local view in DECAN shapes its transformation process, the addition process only depends on the actual instruction.

Original instruction	MULPS	(%RAX,%RCX,8), %XMM4
Modified instruction	MULPS	(%RAX,%RCX,8), %XMM4
	PREFETCHT0	(%RAX,%RCX,8)

Table 4.6: Vector multiplication instruction with a load operand to which a prefetch instruction is added.

4.4.2.1 Transformation combinations

Combining transformations on one instruction is also possible, and eventually, the resulting instruction could also be part of the ISA also. For example, it is possible to both change the `opcode` and the operands of an instruction, or modify its operands and add another instruction.

4.4.3 DECAN Variants

Subsets and transformations being introduced, A DECAN variant is obtained by applying a set of transformations on a set of subsets.

Below is the set of DECAN variants we have set up, tested and validated. The variants can be classified in different manners, we chose to group them according to their themes because of the diversity of tasks they cover. The variants list is the result of practical tests and constantly grows as new useful variants are found.

4.4.3.1 DECAN variants targeting load-store and arithmetic instructions

An application code can be divided into two major components: *arithmetic* and *memory* operations. Arithmetic operations are handled by the functional units inside the micro-processor and memory operations are handled by the memory subsystem. These can be seen as the two major streams into which program instructions can be classified. They also determine the trend of a loop. DECAN provides a variant for each stream:

- `LS`: determines the share of memory operations in a loop performance. This can be achieved by creating a loop version in which only memory operations are present. In order to do so, pure floating-point instructions are deleted, mixed instructions are *replaced* by instructions which operate on memory only.

- `FP`: determines the share of floating-point operations in a loop performance. This can be achieved by creating a loop version in which only arithmetic operations are present. In order to do so, pure memory instructions are *deleted*, mixed instructions are *replaced* by floating-point only instructions.

4.4.3.2 DECAN variants targeting expensive instructions

Some assembly instructions, whether memory or floating-point, may have a high cost and can be a bottleneck in the loop they belong to. DECAN variants are quite adequate in the process of detecting such instructions. During our study, we encountered a number of expensive instructions, for which we constructed the following variants:

- `NO_DIV/NO_SQRT`: is used to verify the impact of high latency instructions such as *division* and *square-root* (a latency of 20 to 40 cycles on *x86 Sandy-Bridge architecture* for the division instruction), on performance. The `NO_DIV/NO_SQRT` variant is obtained by a deletion of the FP-DIV/FP-SQRT subset.

- `NO_RED`: is a variant used to verify if reduction operations are severely impacting the performance of the loop. It is obtained by a deletion of the RED subset.

- `NO_CALL`: the cost of a function call can be quantified through a deletion of the later, on the condition that it does not alter the natural looping proces. Also, in cases where the performance of a loop needs to be profiled but where function calls are also present, a transformation in which the CALL instructions are deleted can be helpful.

4.4.3.3 DECAN variants targeting memory operations investigation

Memory sub-system being a major performance limiting part in scientific applications, we developed a number of DECAN variants to investigate memory operations:

- `DL1`: The goal of the DL1 variant is to emulate an ideal case where all data of the loop are in L1 cache. The resulting variant responds to the question: ***how much can we gain from optimizing all memory accesses of the loop***. It also provides a good indicator of data location in the loop. The

variant operates on the LS subset, where each memory operand is set to point towards a unique memory location created for this purpose. Consequently, if initially the memory operand M_i had a footprint of N which corresponds to the number of different memory locations it pointed to at runtime, after the *DL1* transformation, its new footprint would be 1 corresponding to the first access; all subsequent accesses are located in the L1 cache.

- `GROUPING`: Analyzing performance globally for all memory instructions is too coarse. The goal is to refine the scope of the analysis to something more meaningful and isolate delinquent instructions. However, discarding a single instruction is misleading in case of *hit under miss*. For instance, if *A[0]* is a miss, then the next access to *A*, *A[1]*, will be a hit. Discarding the access to *A[0]* will simply shift the miss to *A[1]*. To pinpoint the bottleneck accurately, an aggregation scheme to regroup accesses on a cache line basis (*A[0]*-*A[3]*) has to be designed. Furthermore, aggregation eases measurement interpretation. Compared to a per cache line analysis, a per data structure analysis is more relevant for an application developer. The ability to discard all the accesses to a given data structure at the same time sorts out the different data structures by cost. The variant is created by identifying the GR subsets and deleting them one at a time.

- *`STORE2LOAD`*: Converts a store operation into a load operation but preserves the same targeted memory operand. The main goal of this transformation is to cancel the cache state change induced by the store operation without completely deleting its effect. Indeed, at a micro-architectural level a standard store operation on an x86 machine first generates a load on the memory location after which the store is applied. By only keeping the load operation, cache coherency problems between threads in a multi-core context can be idealized.

4.4.3.4 Hardware-Software interaction

The low granularity with which DECAN operates allows us to clozely analyze some aspects of hardware-software interactions through the following variants:

- `ADD_LOAD_PREFETCH/ADD_RIP_PREFETCH` Adds a load instruction to an already existing one. The new load can be placed before or after the original one. The main goal of this transformation is to reduce the load ports bandwidth.

- *`CTRL`*: The *CTRL* transformation ensures that all instructions contained in the loop are deleted, whether memory or arithmetic. The goal is to have a version of the loop in which only the instructions involved in the control (branching) are present; this enables to evaluate the cost of the loop overhead.

4.5 Differential Analysis: Main Analysis Methods and Metrics

The motivating example in Section 4.2 gave an overview of data interpretation which illustrated the simplicity of Differential Analysis. The current section describes the analyses which use the results of the variants we introduced in the previous section. We first define the events we are more likely to observe in order to have a better idea of what can or cannot be expected from the analysis.

4.5.1 Observable Events

Three important aspects in any application performance analysis tool are: 1) What are the events it can observe or catch ? Second, how to observe the events ? last, what is the accuracy with which the events are observed ? Though we will address the last point in detail in Chapter 7, the first and second points are important to define.

The primary goal of a DECAN variant is to highlight some performance issue or malfunction (or the inverse: the absence of a performance issue). The variant attempts to achieve this by changing the stress applied on the micro-architectural components that are supposed to be directly involved in the malfunction (components that indirectly help are more difficult to detect). However, this is not always possible because of two limiting factors:

- The stress being applied through the code, cannot always be directly applied to the involved component but rather to another component. For example, knowing if the functional unit responsible of division operation is a potential cause of bottleneck can be directly stressed through the `NO_DIV` variant. Memory hierarchy components, on the other hand are more difficult to stress separately, and it is not possible to know if the L3 cache is a potential source of bottlenecks by stressing it with a variant without changing the complete memory stream behavior.
- The main event we rely on in the comparison process in Differential Analysis is execution time, Hardware usually provides very accurate counters for execution time (e.g. *time stamp counter* on X86_64 platforms). Nonetheless, execution time fails in a number of cases to provide a clear view of the issue, for example it does not allow to verify if the change in performance is due to the stressed component itself or to another one which behavior is altered by the change too. Most modern processors provide multiple counters (commonly known as Hardware Counters (HC)) that monitor different micro-architectural components but they suffer from some drawbacks, we started to study their accuracy in Chapter 7 in order to determine if they can be reliable in the comparison process of Differential Analysis.

Additionally, the primary goal is to get feedback and understanding on the potential causes of performance for a loop. We take execution time as the main mean of observation. The choice is also driven by the ubiquity of the means of execution time monitoring among different architectures, as it allows to use the same analysis notions and concepts from one architecture to the other.

Also, a variant not necessarily targets a single event, it can eventually be an aggregation of events. The streams variants `LS` and `FP` are an example of such a case.

4.5.2 Saturation

Saturation is the main metric with which comparisons are performed among DECAN variants. It is the ratio between the execution time of a DECAN variant `VAR` and that of the original version of the loop `REF`, as illustrated in equation (1). The ratio in equation (1) can be read as: the saturation of `VAR` relatively to `REF`.

Since the variant attempts to reduce or idealize an event, its performance should be inferior to those of the original loop. Therefore, saturation can be interpreted as follows: if the ratio is equal to *one*, then the event highlighted in the variant has no impact on performance, if it is close to *zero* then the event has a big impact on performance. The more the ratio is close to *one* the less it is considered critical in terms of performance within the loop.

$$Sat(\texttt{VAR}, \texttt{REF}) = \frac{T(\texttt{VAR})}{T(\texttt{REF})} \qquad (1)$$

Saturation is not necessarily computed between a variant and the original version of the loop, it may as well be calculated between two variants, one of them being considered as a reference for the second. This means that the new `REF` is itself a variant with a transformed loop and the `VAR` variant contains the transformations applied in the new `REF` plus its own ones. As an example, the `LS` variant can be taken as a reference to the `GROUPING` variants where the share of each memory group is quantified relatively to the memory stream only and not to the entire loop performance.

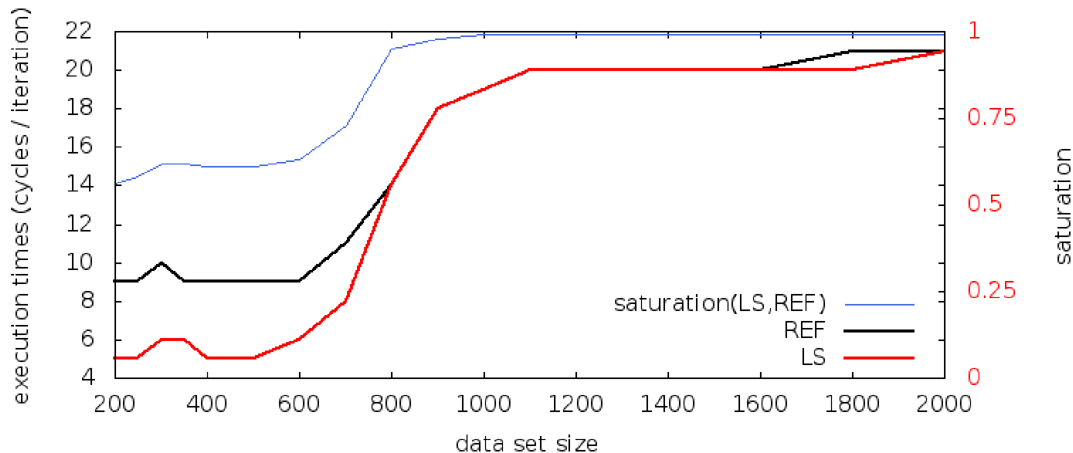


Figure 4.3: Execution time and saturation curves for NR codelet `mprove` on different data sizes for a four cores execution.

Figure 4.3 illustrates the relationship between saturation and execution time. The results are for the NR codelet `mprove_8` on a Sandy-Bridge machine. Two DECAN variants were used: `REF` and `LS`. The `LS` saturation curve (Sat(`LS`,`REF`)) follows the size of the gap between `REF` and `LS`. In data sets of small sizes saturation is between 0.6 and 0.7, whereas for bigger sizes it climbs to a full saturation of 1. The shift in saturation here is explained by the position of data in caches, data are located in higher level caches for small sets of data therefore the latency of memory operations is not the main bottleneck; but it becomes the main one when data are located in lower level caches and notably RAM.

4.5.3 LS/FP Analysis

This analysis refers to a joint study of the saturation of `LS` and `FP` variants. *LS/FP Analysis* (we also call it streams analysis) enables to *characterize* the trend of a

loop and pinpoints whether bottlenecks are part of the memory or of the arithmetic operations.

From a micro-architecture point of view, LS and FP operations share common hardware resources notably the processor front-End but are processed by different functional units in the back-end. Thus, they can ideally be executed in parallel; however, data dependences may be a limiting factor in some extreme cases. Consequently, parallel execution without Front-End and data dependence effects is the optimal case we ought to achieve through *stream analysis*.

Algorithm 2 LS/FP_analysis

Data: $T(\texttt{LS})$, $T(\texttt{FP})$

begin

 $Sat_{\texttt{LS}} \longleftarrow \frac{T(\texttt{LS})}{T(\texttt{REF})}$; $Sat_{\texttt{FP}} \longleftarrow \frac{T(\texttt{FP})}{T(\texttt{REF})}$;

 if $Sat_{LS} = 1 \land Sat_{FP} = 1$ **then**

  Full saturation: Ideal case, the two streams fully overlap and both fully saturate;

 if $Sat_{LS} \gg Sat_{FP}$ **then**

  The loop is bounded by memory accesses;

  if $Sat_{LS} = 1$ **then**

   Two possible strategies:;

   1. More computations can be introduced to reduce the gap ($Sat_{\texttt{LS}} - Sat_{\texttt{FP}}$);

   2. The LS stream can be optimized in order to reduce the gap ($Sat_{\texttt{LS}} - Sat_{\texttt{FP}}$) ;

 if $Sat_{FP} \gg Sat_{LS}$ **then**

  The loop is bounded by floating-point operations;

  if $Sat_{FP} = 1$ **then**

   1. More memory operations can be introduced to reduce the gap ($Sat_{\texttt{FP}} - Sat_{\texttt{LS}}$);

   2. Computations should be optimized in order to reduce the gap ($Sat_{\texttt{FP}} - Sat_{\texttt{LS}}$);

 if $Sat_{LS} < 1 \land Sat_{FP} < 1$ **then**

  - Unsaturation case: two main reasons are possible;

  1. Heavy interaction (dependencies) between the two streams.;

  2. the core Front-end is a bottleneck because the two streams latencies are not big enough to hide the front-end latency.;

  3. bottleencks on some shared micro-architectural components (front-end buffers, ROB, PRF), have been removed within the variants (less instructions due suppressions)

end

Algorithm 2 illustrates the different combinations of values for the saturations of the two streams along with the appropriate interpretation for each of them. From the algorithm we notice three cases

- The ideal case is reached when the two streams are equal and achieve a full saturation, meaning that somehow their operations overlap, which also means

that neither of the two constitute a bottleneck for the other.

- The opposite case, no full overlap, indicates that there is an imbalance between them, which could be eliminated if either the bottleneck stream (the one with bigger saturation) optimize its operations to lower its saturation, or, if the other stream reduces the slack it.

- The last case is when neither of the two achieves a full saturation. One of the streams can still be a bottleneck, however, it may also reveal either a heavy interaction between the streams leading to a speedup when they are separated, or a relaxation on some components in the micro-processors, because of less instructions in the LS and FP variants.

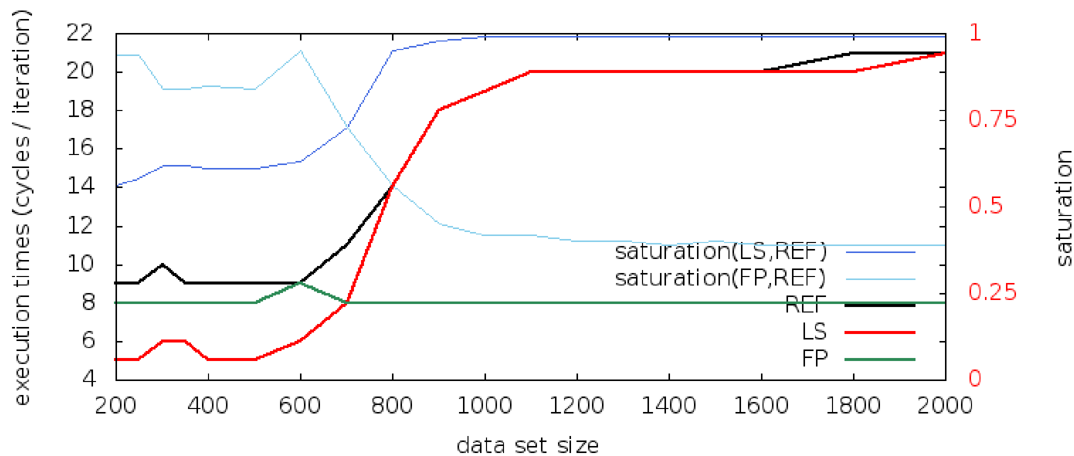


Figure 4.4: Execution time and saturation curves for NR codelet mprove on different data sizes. Results are for a four cores execution.

Figure 4.4 illustrates different configurations for LS and FP saturations for the same NR codelet mprove_8. We notice how the code is FP bound for small data set sizes . More balance between the two streams is achieved when data set of bigger sizes are used. We then reach a perfect balance between the two at a size of 700, however, they are in an unsaturation state. Withing data set of big size, the code is memory bound and gradually the LS variant reaches a full saturation.

4.5.4 Data Location and Return On Investment

In a memory bound loop, the location of loop data greatly determine the performance. Knowing whether data are in memory or in cache is of great help. Moreover, knowing that allows to have a precise idea of the potential return on investment; indicating which loop is worth optimizing first. This can be illustrated through a simple example: suppose that an application has two hot loops A and B respectively representing 30% and 50% of the execution time. Given these numbers, it seems more profitable to focus optimization effort on B, however, if the memory accesses of the two loops are located in L1, then their execution times would be respectively reduced by 90% and 30% respectively. Consequently, it would be more interesting to focus on A since the total gain on its execution time would be 27% instead of

15% for loop B respectively. `DL1` saturation numbers enable us to perform this kind of analysis. The comparison between the original performance of a loop and its performance with an ideal memory behavior, allows to highlight within all the loops of an application those representing the worst memory behavior. Thus, it allows to reorder the priority list of loops to optimize. The initial order usually being dictated by a simple execution time profile.

$$Sat(\texttt{DL1}, \texttt{REF}) = \frac{T(\texttt{DL1})}{T(\texttt{REF})} \qquad (3)$$

Saturation numbers can be interpreted as follows: If the DL1 saturation ratio in equation (3) is equal to 1, then memory accesses are all in L1. Beyond this ratio, the more saturation decreases, the more data are considered within lower level caches. We also noticed that, in general, when the saturation is above 50%, then data are within the L2 cache boundary.

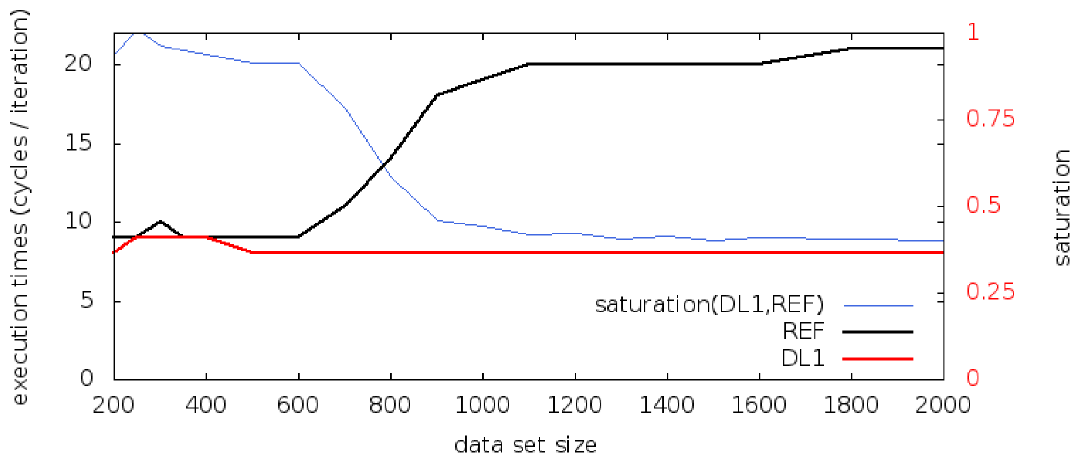


Figure 4.5: Execution time and saturation curves for NR codelet `mprove` on different data sizes for a four cores execution.

Figure 4.5 shows `DL1` saturation for the NR codelet `mprove_8`. Saturation results indicate that for small data sets, there is nothing to do regarding memory operations. It also shows that in bigger data sets (starting from around 700) there is a significant potential of performance improvement (about 60% since `DL1` saturation converges to 40%)

4.5.5 Expensive Instructions

In the case of a loop being bounded by its floating-point operations, it is relatively easy to find the main source of bottleneck (if any). Within the FP stream, instructions with high execution latency are a common source of bottlenecks. These are generally the same instructions in different ISAs and architectures such as *divisions* and *square-roots*. Their execution latency is at a much higher level than the other standard FP operations such as *additions* and *multiplications*.

The share of an expensive instruction can be determined by observing the saturation of the DECAN variants `NO_DIV` and `NO_SQRT`. A saturation close to 1 means that the instruction does not have an impact on performance, whereas a saturation close to zero means that the instruction is an important bottleneck within the loop

4.5.6 Array Cost Analysis

The current section addresses array performance analysis, a well investigated issue in HPC performance analysis through different techniques and tools. We describe how differential analysis, through *Array Cost Analysis*, can come as a support for other techniques (e.g. memory simulation tools) by giving quick assessments on array performance.

Array Cost Analysis is an automatic lightweight performance analysis with the following advantages:

- Provide quick assessment of the share each array has in the loop.
- Enables memory tracers to only focus on the major arrays, and hence decrease their execution overhead
- Provide feedback on the performance of high level structures(arrays) from low level analysis.

4.5.6.1 Principle

The concept of array cost analysis extends directly the DECAN groups analysis introduced in [57, 58] and detailed in Section 4.4.1. Groups analysis consisted basically in deleting the groups one by one in order to find those that cost the most in terms of performance; interactions between groups can also be analyzed by deleting combinations of groups. In *ACA* we keep the same analysis concept but extend it to source level arrays instead of groups.

4.5.6.2 Array Reconstruction: Fast Memory Tracer tool (FMT)

The solution currently used in DECAN groups analysis, called grouping, is based on static analysis. The concept, is to track memory instructions with the same base address, and therefore part of the same array, within the register contents. Static analysis proved to be effective in several cases and provided some good results. Even if the entire array is not recovered, it is gathered into few data fragments called groups. We improved these results through a complementary lightweight dynamic analysis. The idea is to trace memory references and record, for each memory instruction, the range of memory cases they cover.

FMT´s design is driven by constraints of fastness and very low overhead. The design of this tool obey to the following rules:

- No function call is injected inside the loop. All memory references are recorded in buffers created with MADRAS through binary patching.
- To avoid overflows, we ensure that the size of each buffer is equal to the number of iterations of the loop.
- Buffers contents are dumped at the exit of the loop, so that the same buffers are reused in the next loop call
- For each memory instruction, we only record the smallest and the biggest memory address. This eliminates the heavy memory footprint we find in conventional tracers.

- Instructions that reference the same array are grouped according to the following rules:
  1. Two instructions $I1$ with reference range $R1$=[@L1,@H1] (@L1 being the lowest reference addressed by I1 and @H1 being the biggest) and $I2$ with reference range $R2$=[@L2,@H2] are considered part of the same array if: $R1 \cap R2 \neq \{\phi\}$.
  2. An instruction $I1$ is part of a group of instructions $G1$ if there is at least one instruction $I2$ part of $G1$ such that $I1$ and $I2$ satisfy (1).
  3. Two instruction groups $G1$ and $G2$ are merged to form a super group if there is at least one instruction $I1$ in $G1$ and one instruction $I2$ in $G2$ such that $I1$ and $I2$ satisfy (1).
- In order to overcome some cases where no overlap among memory references is detected, program `mallocs` are traced as well. A traced `malloc` is considered as a group G with reference range R=[@REF, @REF + MALLOC_SIZE] (@REF being the address returned by `malloc` and MALLOC_SIZE the size of the allocation), and the same group constraints are applied to it.

It is worth noting, however, that recovering all data structures from a binary is usually performed in a decompilation context. It is a tedious task, and contains a lot of challenges. In our case, the context is different; as we only seek to reocver arrays, we can exploit array elements grouped in a contiguous memory area. Furthermore, the recovery scope is limited to loop level since our study is loop centric.

4.5.6.3 Array reconstruction Results

Table 4.6 shows the results of detection for both the grouping static analysis and FMT runtime analysis on 27 Numerical Recipes codelets.

For our experiments, we chose to work on small compute kernels called *codelets*. They have been notably used by other works [79]. The codeletsare issued from: four important algorithm types from Numerical Recipe[7] codes(see Ch. 2. Solution of Linear Algebraic Equations, Ch.11. Eigensystems, Ch. 12. Fast Fourier Transform, and Ch. 14. Partial Differential Equations). From these chapters, 17 important recipes were selected, covering many functionalities described in these parts of litterature. A codelet extraction tool called *Codelet Finder* automatically extracted 96 codelets from these 17 algorithms. Some recipes yielded up to 26 distinct codelet types, but most have less than 10. Profiling revealed that 27 codelets (the ones we used), represent most of the execution time in the 17 key numerical recipes[1].

on the one hand, we notice that static analysis allows us to achieve 37% of success on the codelet suite. FMT, on the other hand, was able to detect all source arrays. Nonetheless, we recall that there are cases where FMT would fail to reconstruct the entire arrays, in which case we would end-up with fragments such as those returned by static analysis.

4.5.6.4 Runtime overhead

It is important to quantify how much time the refinement offered by FMT is going to cost us. Figure 4.6 shows tracing results on an NR codelet called BALANC_3.

[1] We used some of the codelets in other parts of our work, these appear in several parts throughout the manuscript

Codelet	Groups constructed		source level arrays
	Static analysis	**FMT analysis**	
Balanc_3	1	1	1
Svdcmp_13	1	1	1
Svdcmp_14	2	2	2
Toeplz_1	3	3	3
Four1_2	2	1	1
Hqr_12	2	2	2
Lop_13	5	2	2
Ludcmp_4	9	1	1
Relax2_26	9	2	2
Hqr_15	2	1	1
Jacobi_5	2	1	1
Matadd_16	3	3	3
Mprove_8	3	2	2
Rstrct_29	5	2	2
Svdksb_3	3	2	2
Elmhes_10	2	1	1
Elmhes_11	16	1	1
Hqr_13	2	2	2
Mprove_9	2	2	2
Realft_4	2	1	1
Svdcmp_11	8	1	1
Svdcmp_6 1	7	2	2
Toeplz_2	3	2	2
Toeplz_3	4	3	3
Toeplz_4	4	2	2
Tridag_1	6	6	6
Tridag_2	2	2	2
% **of success**	**37,04%**	**100,00%**	**100,00%**

Table 4.7: Group detection results through both grouping static analysis and FMT runtime analysis

.

The particularity of BALANC_3 is that it is regular in term of memory behavior, meaning that by simply varying the problem size, we can clearly see if data are in L1, L2, L3 or the RAM. This regularity allows us to see the tracing overhead depending on where data are in memory hierarchy. Thus, from the figure, if data are in L1 we approximately have 37 of slowdown on the loop original performance, and as we shift to lower cache levels the overhead contribution also shrinks. In the L2 cache, the overhead drops to 27, whereas in L3 cache it is at approximatively 19. In RAM, we are at about 7.

4.6 Case Studies

The current section discusses some notable applications of differential analysis throughout the study of different codes and for different objectives. Other case studies we worked on are detailed in Chapter 5. The first case shows the use of Differential Analysis in the process of *application characterization*. The second study case illustrates the use of *array cost analysis*. The last study case illustrates an original use of differential analysis to evaluate the hardware adequacy to software needs.

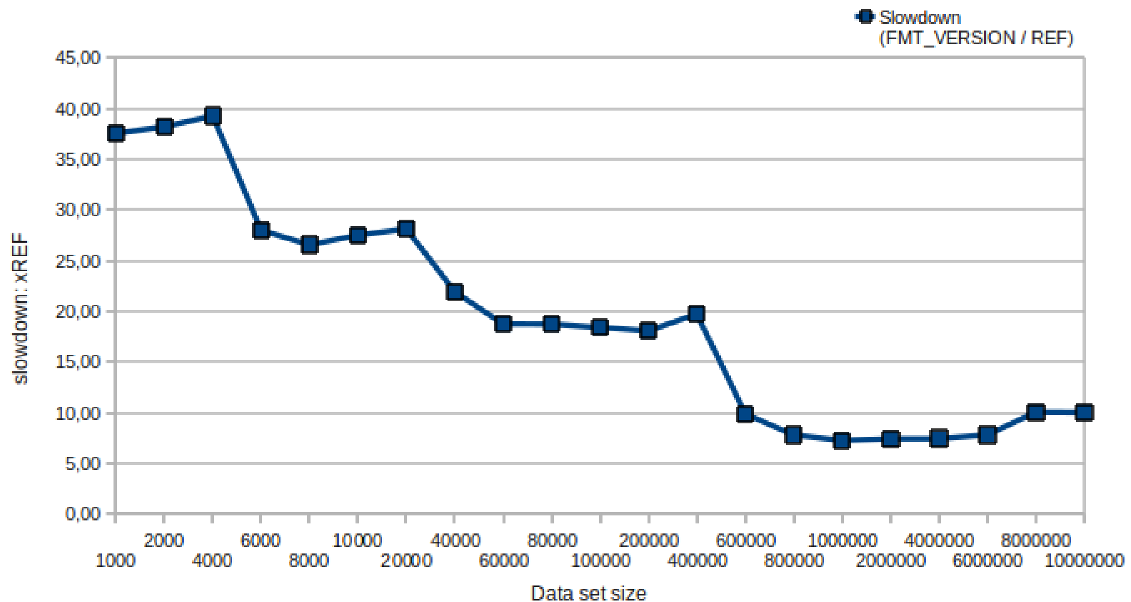


Figure 4.6: Execution Slowdown of a version of the loop instrumented to perform FMT over its original version for `BALAN_3` codelet on different data sizes. The four levels of slowdown correspond from left to right to data being in L1, L2, L3 caches and RAM.

4.6.1 Application Characterization and Analysis: RTM application

Within this section we address RTM, an industrial code that we characterize with *LS/FP*, *Data location* analyses.

Reverse Time Migration (RTM) [23] is a standard algorithm used for geophysical prospection. The code used in this study is an industrial implementation of the RTM algorithm by the oil and gas company TOTAL.

Our RTM code operates on a regular 3D grid. The core of the domain is uniformly processed, but a specific treatment is applied on the borders of the domain, the skin of the domain, to annihilate potential wave reflections. From a performance perspective, more than 90% of the execution time of the application is spent in two functions, *inner* and *damping*. These two functions are executing similar code on two different parts of the domain, *inner* is devoted to the core of the domain, while *damping* is used on the skin of the domain. Standard domain decomposition techniques are used to spread the workload on multi-core target machines. Since the grid is uniform, load balancing is easily achieved by using rectangular sub-domains. We conduct our analysis on the compute kernel which operates inside a node. OpenMP is used to exploit parallelism among cores. Hence, computations on the grid are evenly divided between the cores.

A profile of the kernel resulted in the selection of a set of 11 loops, in which more than 90% of the execution time is spent. Figure 4.7 gives the importance of each loop in respect to their share in the total execution time. The results are ordered from the most to the least time consuming one. We notice that `damping_174` dominates the list with a share of 37% of the total, whereas `inner_132` is the least important one with 5%. These results suggest to focus first and primarily on loops `damping_174` and `inner_126` because of their high share in the execution time.

The loops can be quickly characterized through a *LS/FP analysis*. Figure 4.8

illustrates saturation results for each loop, where we notice a clear memory oriented behavior for all loops. Indeed, LS saturation is between 80 and 100%, whereas FP saturation do not exceed 40%. This perspective would suggest ,for all loops, the following: either to focus on optimizing memory accesses which itself is conditioned by the accesses being bad, or to put more computations in the loops in order to balance between the streams (e.g. merge loops with close memory accesses that may overlap).

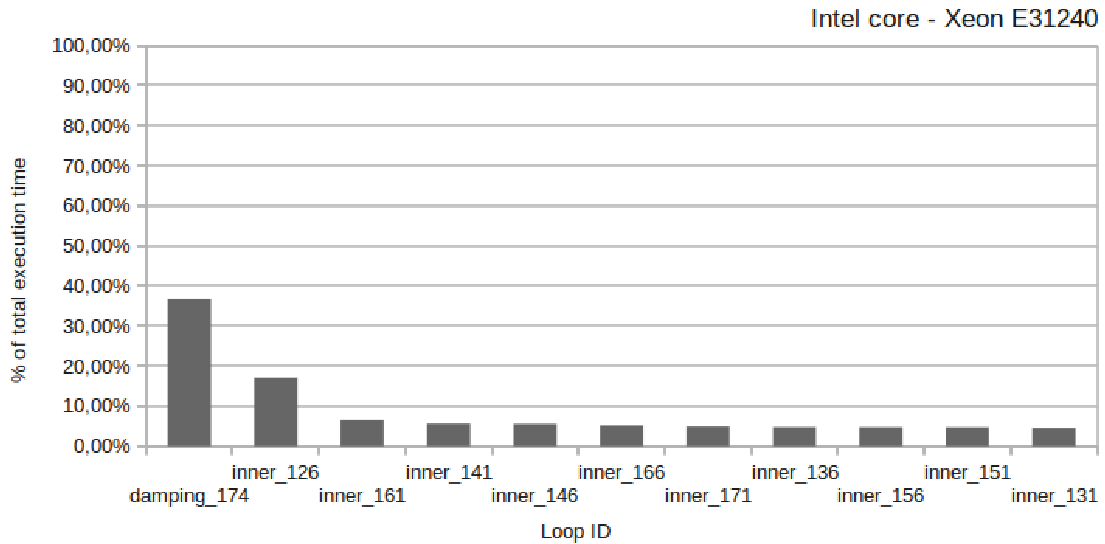


Figure 4.7: Execution time in cycles for the hottest loops of the RTM kernel.

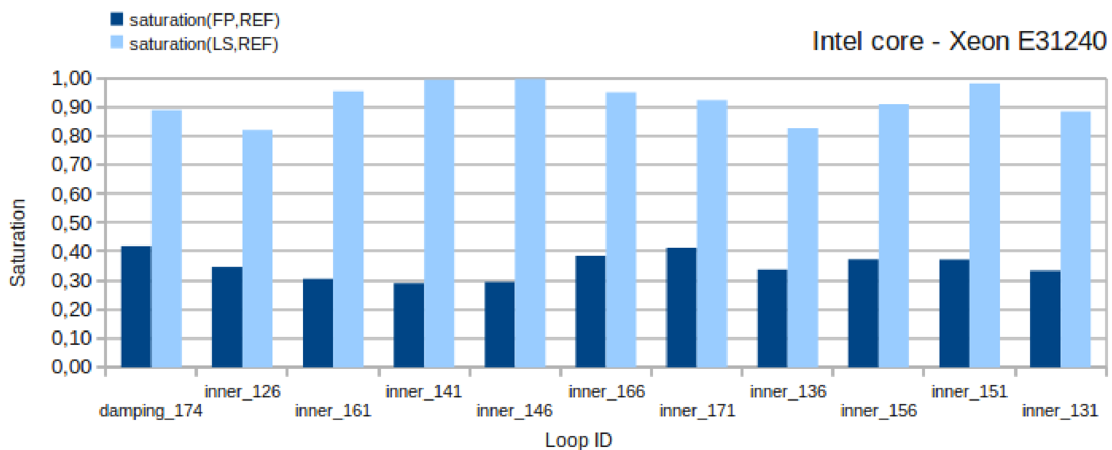


Figure 4.8: LS/FP saturations for the hottest loops of the RTM kernel

In our case, the loops operate on different data, therefore loop merging appears to be inefficient. We turn our attention to the possibility of optimizing memory operations. In order to do that, we analyze the saturation of the DL1 variants, which simulate an optimal memory behavior for the loops. The results shown in Figure 4.9 allow us to draw the following observations:

- The DL1 saturation of the most time consuming loop damping_174 is near 90%, which means that the accesses of the loop are almost all hits in the L1 cache.

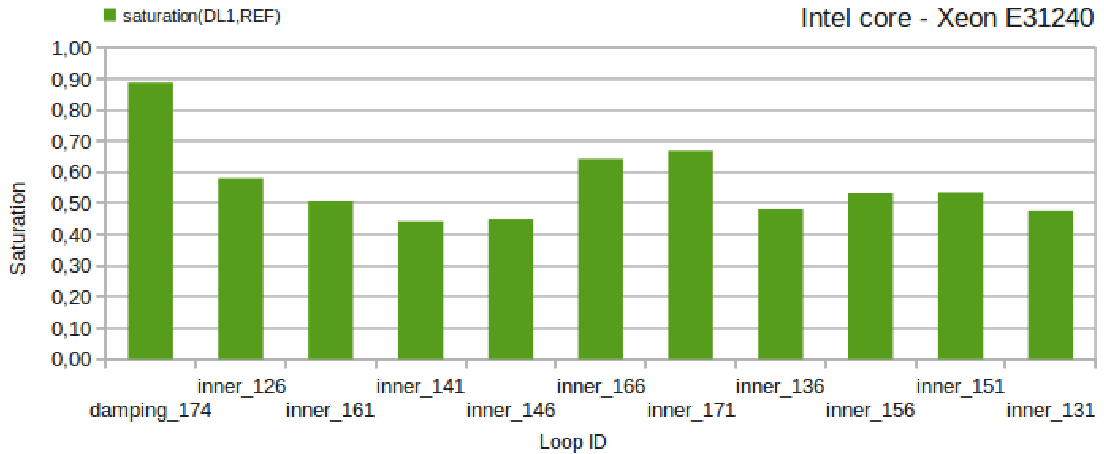


Figure 4.9: DL1 saturations for the hottest loops of the RTM kernel

- The `DL1` saturation of the rest of the loops, are for the most between 40% and 60%, meaning that their memory accesses are roughly within the L2 cache limit.

From these observations we derive two conclusions:1) the priority order established by execution times is misleading, since the most time consuming loop happens to be the one which has the best memory behaviour. `DL1` saturation results suggest instead to focus on loops where the gap is broad, since the gain would be bigger. 2) The fact that the loops have their data within the L2 limits indicates that the chosen blocking already offers a descent locality, since the L1 cache is small and it can be extremely difficult to fit data into it, especially in case when several arrays are present (which is the case in the loops of RTM).

4.6.2 ACA: EUFLUXm Application

EUFLUXm is a 3D finite element CFD application from Dassault [38], ITRSOL (the iterative solver) represents over 80% of the execution and is the most time consuming routine is EUFLUXm. The EUFLUXm routine implements a sparse matrix-vector product in a quadruply nested loop. Among the four different arrays, three of them, *VECX*(2D), *OMPU*(3D) and *OMPL*(3D), are read-only and the last one *VECY*(2D) is read and written. The code of EUFLUXm is presented in Figure 4.10.

A quick inspection of the code reveals that most of the array accesses suffer from bad stride access. The two innermost loops could be interchanged or, alternatively, arrays could be restructured (transposed). Restructuring an array is a complex and expensive code modification because it has to be propagated throughout the whole application. Therefore the issue we are interested in, is how to determine which arrays should be prioritized for restructuring.

We tried first, to use static analysis to detect the four arrays VECX, VECY, OMPU and OMPL as it does not have a cost. The analysis managed to construct 10 groups out of 14 memory references which is a bit far from our target number. FMT, on the other hand, succeeded in detecting the four arrays as shown in Table 4.8, but it cost us an execution 12 times slower than the original one.

Analysis	Number of groups detected	Analysis cost (Slowdown over original)
Static analysis	10	no slowdown (significant speedup)
FMT	4	12.27

Table 4.8: Results of group reconstruction for EUFLUX application with the use of both static and dynamic (FMT) analyzes. The cost of each analysis in slowdown over original execution time is shown

```
Do icb=1,ncbt
  Igp     = isg
  Isg     = icolb( icb+1)
  Igt     = isg – igp
  !$OMP  PARALLEL DO
  !$OMP& SHARED(igt, igp, nnbar, vecy, vecx, ompu, ompl, ndof)
  !$OMP& PRIVATE(ig, e, i, j, k, l)
   Do ig=1,igt
       e      = ig + igp
       i      = nnbar( e, 1)
       j      = nnbar( e, 2)
       Do k=1,ndof
          Do l=1,ndof
                  Vecy( i, k) = vecy( i, k) + ompu( e, k, l) * vecx( j, l)
                  Vecy( j, k) = vecy( j, k) + ompl( e, k, l) * vecx( i, l)
          enddo
        enddo
   enddo
   !$OMP END PARALLEL DO
enddo
```

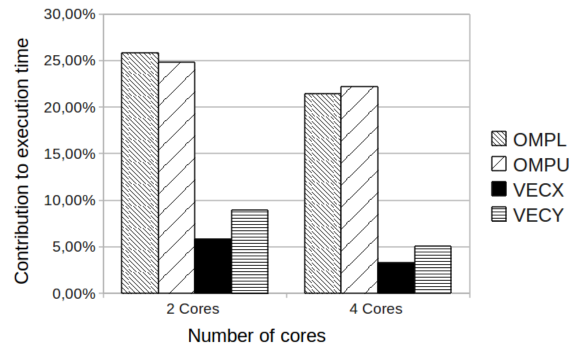


Figure 4.10: The left figure illustrates the source code of the matrix-vector product in EUFLUXm. The right figure shows the individual contribution in the overall execution time of memory instructions targeting each array of the EUFLUXm routine. Results are presented for 2 and 4 cores.

The arrays being detected at assembly level, we could match them with source level arrays with the help of *debug informations* (obtained by compiling the program with a `-g`), which enabled to associate each assembly instruction with its source line and column. We generated four DECAN variants to assess their costs. Each variant had the memory references of one of the suppressed arrays. The difference in execution times between each variant and the original program indicated the cost of each array. Figure 4.10 shows that among the 4 arrays, the accesses to *OMPU* and *OMPL* are the most time consuming. The two arrays represent more than 40% of the total execution time, followed by *VECY* and *VECX* with an individual time share lower than 10%. Therefore, we concluded that any optimization effort has to consider that performance issues in the loop are essentially tied to the *OMPU* and *OMPL* arrays.

4.6.3 L1 Load Bandwidth Evaluation

Another register in which Differential Analysis might be helpful is the exploration of Micro-architectural features of the processor. Eventually, the analyzes one can think of are not as elaborated and precise as those we find in [41, 40, 42], but we still have enough oportunities of obtaining meaningful insights on software-hardware interactions.

The idea here is to evaluate the adequacy of L1 cache bandwidth of the core for real life applications. An appropriate L1 bandwidth is the one just big enough to service the hot loop needs without any contention problems. The whole bandwidth hasto be used, meaning that the bandwidth is not oversized.

We can evaluate a bandwidth with Differential Analysis by overloading the memory stream. This can be achieved by the add of more memory instructions in the loop. The `PREFETCH` transformation makes this possible, by adding after each memory instruction of the loop a software `prefetch` instruction. The memory references of the added instructions point to the same memory reference, hence ensuring L1 accesses. The advantage of the use of `prefetch` instructions instead of a duplication of the loads is that they would exactly act like memory instructions without any alteration of the semantics of the program.

Algorithm 3 Polaris (MD) loop 2937 source code

```
DO j = ni+nvalue1,nagroua
    nfj = nfi + nvalue2*(j-ni_ref)
    nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2+nvalue1
    u1 = x11-x(nj1) ; u2 = x12-x(nj2) ; u3 = x13-x(nj3)
    p21 = pm1(nj1) ; p22 = pm1(nj2) ; p23 = pm1(nj3)
    psr1 = u1*p21 + u2*p22 + u3*p23
    psr2 = u1*p11 + u2*p12 + u3*p13
    rij3 = dtemp(nfj) ; rij5 = dtemp(nfj+nvalue1)
    rij15 = psr1*rij5 ; rij25 = psr2*rij5
    pf1 = pf1 + u1*rij15 + p21*rij3
    pf2 = pf2 + u2*rij15 + p22*rij3
    pf3 = pf3 + u3*rij15 + p23*rij3
    pm2(nj1,thread_num) = pm2(nj1,thread_num) + u1*rij25 + p11*rij3
    pm2(nj2,thread_num) = pm2(nj2,thread_num) + u2*rij25 + p12*rij3
    pm2(nj3,thread_num) = pm2(nj3,thread_num) + u3*rij25 + p13*rij3
END DO
```

	Variant		
	DL1	**FP**	**LS**
Saturation	79%	80%	81%

Table 4.9: Stream analysis for loop 2937 of POLARIS (MD)

We took as a test example for the analysis a loop from an industrial code named POLARIS (MD). Its code is shown in Algorithm 3. Saturations of the `LS`, `FP` and `DL1` variants of the loop are summarized in Table 4.9. We performed gradual overloads that reduced the bandwidth by 10% up to 50%, where a 10% reduction means that a `prefetch` instruction is placed for each ten memory instructions, and so forth. Therefore, a 50% reduction means that the bandwidth is divided by two.

The effects of bandwidth reduction on the loop are shown in Table 4.10. The results can be interpreted as follows: if bandwidth is reduced by 10% than perfor-

mance will drop by 2.33%, if it is reduced by 30% for example then there would be a drop of 11.41%. It is interesting to notice that if the bandwidth of the L1 cache is divided by two, we end up with only 17.32% of global slowdown on loop performance.

This case opens the door for Differential Analysis to be considered in fields other than application performance analysis; we show here that it is possible to use the tool to evaluate some micro-architectural aspects through a simple and lightweight process. Consequently, we think that it is possible to use the approach in software-hardware codesign matters.

	% of reduction of L1 cache bandwidth					
	50%	**40%**	**30%**	**20%**	**10%**	**REFERENCE**
Average overhead	17,32%	14,04%	11,41%	7,00%	2,33%	0,00%

Table 4.10: Effect of L1 cache bandwidth reduction on performance for loop 2937 of POLARIS (MD)

4.7 Summary

Within this chapter, we introduced Differential Analysis through a typical bottleneck detection issue example. We saw that it is based on comparing different versions of the same loop, each version, also called *variant*, introduces controlled changes to the assembly instructions of the loop. The goal is to be able, with a variant, to diagnose at least one pathology. We then described the variant creation process: *instruction subsets* detection and *instruction transformations*. The variants being introduced, we illustrated how their performance scores are interpreted to get meaningful insight on different performance trends and issues. At the end, we saw some cases where the analyzes are used to understand the performance of real life industrial applications. The last case study also showed that the analysis can be applied to evaluate details of the micro-architecture, opening the door for other uses worth studying.

Chapter 5

Pamda: Performance Assessment using Maqao toolset and Differential Analysis

5.1 Introduction

The recent progress of high performance architectures generate new challenges for performance evaluation tools: more complex processors (larger vectors, manycores), more complex memory systems (multiple memory levels including NUMA, multiple level prefetch mechanisms), more complex systems (large increase in core counts up to several hundreds of thousands now) are all key issues which need to be simultaneously optimized to get a decent performance level.

To work properly, all of these mechanisms require specific properties from the target code. For example, good exploitation of memory hierarchies relies on good spatial and temporal locality within the target code. The lack of such properties induces variable performance penalties: such combinations (mismatch between hardware and software) are denoted as performance pathologies. Most of them have been identified (cf. Table 5.1) and efficient workarounds are well known. The current generation of performance tools (TAU [87], PerfExpert [29], VTune [51], Acumem [18], Scalasca [44], Vampir [77]) is excellent at detecting such pathologies although some are fairly specialized: for instance, Scalasca/Vampir mainly addresses MPI/OpenMP issues, requiring the combined use of several tools to get a global overview of all of the performance pathologies present in an application.

Most of the current tools do not provide any direct insight on the potential cost of a pathology. Furthermore, the user has no idea about what the potential benefit of optimizing his code to fix a given pathology is. These two points prevent him from focusing on the right issue. For example, let us consider a program containing two hot routines A and B, respectively consuming 40 % and 20 % of the total execution time. Let us further assume that the potential achievable performance gain on A is 10 % while on B it is up to 60 %. The overall performance impact on B can reach 60 % * 20 % = 12 % while on routine A, it is at best 10 % * 40 % = 4 %. As a consequence, it is preferable to focus on routine B. Additionally, the user has no clue of what the current performance level is, compared with the best achievable one, i.e. he may not know when optimizing is worth the investment.

In general, the situation is even worse since a simple loop may simultaneously exhibit several performance pathologies. In such cases, most of the above cite tools give the user no hint of which ones are dominant and really worth fixing. For instance, a loop can suffer from both a high miss rate and the presence of costly Floating-Point (FP) operations such as div/sqrt: trying to improve the hit rate does not improve the performance if the dominant bottleneck is the div/sqrt operations.

In this chapter, we propose a coherent set of tools (MicroTools [24], Cqa [32],

DECAN [58], MTL [33]) to address this lack of user guidance in the tedious and difficult task of program optimization. These tools are integrated in a unified environment (PAMDA) to help the user to quickly identify performance pathologies and to assess their cost and their impact on the global performance. The different techniques (static analysis, value profiling, dynamic analysis) appear to be more appropriate and give a more accurate answer depending upon which performance pathologies have to be fixed: for example, detecting a badly strided access is immediate through value tracing of array addresses, while the same task is extremely tedious when only using static analysis or hardware counters. Anyway, such array access tracing should only be triggered when really necessary due to its high cost. In this paper, we focus on providing performance insight at core level and parallel OpenMP structures. Our analysis can be combined with MPI analysis provided by tools such as Scalasca, TAU or Vampir.

Through the integrated environment PAMDA, we aim at providing the following contributions:

- To get a global hierarchical view of performance pathologies/bottlenecks
- To Get an estimate of the impact of a given performance pathology taking into account all other present pathologies
- To demonstrate that different specialized tools can be used for pathology detection and analysis
- To perform an hierarchical exploration of bottlenecks according to their cost: the more precise but expensive tools are only used on specific well chosen cases.

Section 5.2 presents a motivating example in detail. Section 5.3 details the various key components of PAMDA while Section 5.5 describes the combined use of these different tools. Section 5.5 describes some experimental use of PAMDA. Section 5.6 gives an overview of related works and the added value of the PAMDA system. Finally, Section 5.6 gives conclusions and future directions for improvement.

5.2 Motivating Example

Figure 5.1 presents the source code of one of the hottest loops extracted from POLARIS(MD) [83]: a molecular dynamics application developed at CEA DSV. POLARIS(MD) is a multiscale code based on Newton equations: it has been successfully used to model Factor Xa involved in thrombosis.

This loop simultaneously presents a few interesting potential pathologies:

- Variable loop trip count.
- Fairly complex loop body which might lead to inefficient code generation by the compiler.
- Presence of div/sqrt operations.
- Strided and indirect access to arrays (scatter/gather type).
- Multiple simultaneous reduction operations leading to inter-iteration dependencies.

Table 5.1: A few typical performance pathologies.

Pathologies	**Issues**	**Workarounds**
ADD/MUL balance	ADD/MUL parallel execution (of fused multiply add unit) underused	Loop fusion, code rewriting e.g. Use distributivity
Non pipelined execution units	Presence of non pipelined instructions: div, sqrt	Loop hoisting, rewriting code to use other instructions eg. x86: div and sqrt
Vectorization	Unvectorized loop	Use another compiler, check option driving vectorization, use pragmas to help compiler, manual source rewriting
Complex control flow graph in innermost loops	Prevents vectorization	Loop hoisting or code specialization
Unaligned memory access	Presence of vector-unaligned load/store instructions	Data padding, use pragma and/or attributes to force the compiler
Bad spatial locality and/or non stride 1	Loss of bandwidth and cache space	Rearrange data structures or loop interchange
Bad temporal locality	Loss of perf. due to avoidable capacity misses	Loop blocking or data restructuring
4K aliasing	Unneeded serialization of memory accesses	Adding offset during allocation, data padding
Associativity conflict	Loss of performance due to avoidable conflict misses	Loop distribution, rearrange data structures
False sharing	Loss of bandwidth due to coherence traffic and higher latency access	Data padding or rearrange data structures
Cache leaking	Loss of bandwidth and cache space due to poor physical-virtual mapping	Use bigger pages, blocking
Load unbalance	Loss of parallel perf. due to waiting nodes	Balance work among threads or remove unnecessary lock
Bad affinity	Loss of parallel perf. due to conflict for shared resources	Use numactl to pin threads on physical CPUs
High number of memory streams	Too many streams for hardware prefetcher or conflict miss issues	See conflict misses
Lack of loop unrolling	Significant loop overhead, lack of instruction-level parallelism	Try different unrolling factors, unroll and jam for loops nest, try classical affinities (compact, scatter, etc.)

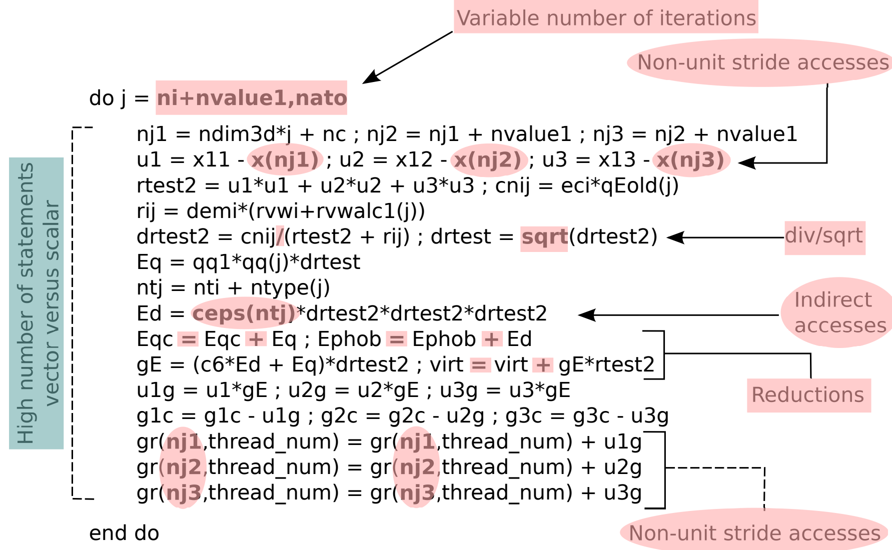


Figure 5.1: A Fortran source code sample and its main performance pathologies highlighted in pink.

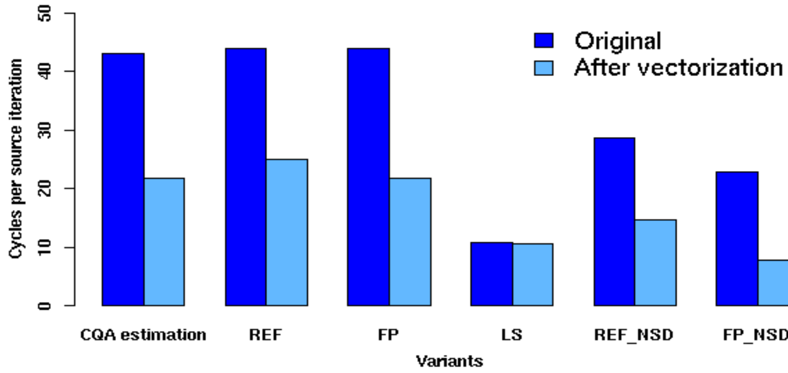


Figure 5.2: Comparing static estimates obtained by Cqa with dynamic measurements performed on different code variants generated by DECAN of both the original and the vectorized versions: **REF** is the reference binary loop (no binary modifications introduced by DECAN), **FP** (resp. **LS**) is the DECAN binary loop variant in which all of the Load/-Store (resp. FP) instructions have been suppressed, **REF_NSD** (resp. **FP_NSD**) is the DECAN binary loop variant in which only FP sqrt and div instructions (resp. all of the Load/Store and FP sqrt/div instructions) have been suppressed. The y-axis represents the number of cycles per source iteration: lower is better.

All these pathologies can be directly identified by simple analysis of the source code. The major difficulty is to assess the cost of each of them and therefore to decide which should be worked on.

A first value profiling of the loop iteration count reveals that the trip count is widely varying between 1 and 2000. However the amount of time spent in the trip count instances of small (less than 150 iterations) loop remains limited to less than 10 %. The remaining interval of loop trip counts is further divided into 10 deciles and one representative instance is selected for each of them. Further timings while analyzing loop trip count impacts, indicate that the average cost per iteration globally remains constant independently from the trip count. Therefore, the data size variation seems to have no impact on performance: the same optimization techniques could be applied for instances having a loop trip count between 150 and

2000.

The static analyzer (see Figure 5.3) provides us with the following key information: in the original version, neither Load/Store (LS) operations nor FP ones are vectorized. It further indicates that due to the presence of div/sqrt operations, the FP operations are the main bottlenecks. It also points out that even if the FP operations were vectorized, the bottlenecks due to sqrt/div operations would remain. However this information has to be taken with caution since the static analyzer assumes that all data accesses are ideal, i.e. performed from L1.

Dynamic analysis using code variants generated by DECAN is presented in Figure 5.2. Initially, the original code (in dark blue bars) shows that FP operations (see `FP` versus `LS` DECAN variants) clearly are the dominating bottlenecks. Furthermore, the good match between CQA and `REF` clearly indicates that analysis made by CQA is valid and pertinent. Optimizing this loop is simply obtained by inserting the SIMD pragma `'!DEC$ VECTOR ALWAYS'`, which forces the compiler to vectorize FP operations. However, the compiler does not vectorize loads and stores due to the presence of strides and indirect accesses. Rerunning DECAN variants of this optimized version (see light blue bars in Figure 5.2) shows that, even for this optimized version, FP operations still remain the key bottleneck (comparison between LS and FP). Therefore, there is no point in optimizing data access, the only hope of optimization lies in improving div/sqrt operations: for example SP instead of DP. Unfortunately, in this case such a change would alter the numerical stability of the code and cannot be used.

The major lesson to be drawn from this case study is that a combined use of CQA and DECAN allows us to quickly identify the optimization to be performed and also gives us a clear halt on tackling other pathologies without impacting overall performance.

5.3 Ingredients: Main Tool Set Components

Performance assessment issues require robust methodologies and tools. Therefore, in order to systematically provide programmers with a performance pathology hierarchy and its related costs, the current work considers two toolsets: MicroTools, for microbenchmarking the architecture, and the MAQAO [22, 59, 12] framework, which is a performance analysis and optimization tool suite.

The goal of MAQAO is to analyze binary codes and to provide application developers with reports to optimize their code. The tool mixes both static (code quality evaluation) and dynamic (profiling, characterization) analyses based on the ability to reconstruct low level (basic blocks, instructions, etc.) and high level structures such as functions and loops. Another MAQAO key feature is its extensibility. Users can easily write plugins thanks to an embedded scripting language (Lua), which allows fast prototyping of new MAQAO-tools.

From MAQAO, PAMDA extensively uses three tools including the Code Quality Analyzer tool (CQA) exposed in section 5.3.2, the Differential Analysis framework (DECAN) presented in section 5.3.3, and finally the Memory Tracing Library (MTL) in section 5.3.4. We briefly present how these three tools contribute to PAMDA and then describe their major characteristics.

5.3.1 MicroTools: Microbenchmarking the Architecture

Microbenchmarking [90, 64, 20] is an essential tool to investigate the real potential of a given architecture: more precisely, in Pamda, microbenchmarking is first used to determine both FP units performance and achievable peak bandwidth of various hardware components such as cache/RAM levels. Second, to estimate the potential cost of various pathologies (unaligned access, 4K aliasing, high miss rate, etc.).

For achieving these goals, Pamda relies on `MicroTools`, consisting of two main components: `MicroCreator` tool automatically generates a set of benchmark programs, while `MicroLauncher` framework executes them in a stable and closed environment.

5.3.2 Cqa: Code Quality Analyzer

In Pamda, the Cqa framework is used first for providing a performance target under ideal data access conditions (all operands are supposed to be in L1), second for providing a bottleneck hierarchy analysis between the various hardware components of the core (FP units, load/store ports, etc.) and third for detecting some performance pathologies (presence of inter iterations dependencies, div/sqrt operations) which are worth investigating via specialised Decan variants. The ideal assumption (all operands in L1) is essential for CPU bound codes such as the POLARIS(MD) loop studied in the previous section. For memory bound loops, a further dynamic analysis is necessary.

Cqa is a static analysis tool directly dealing with binary code. It extracts key characteristics, and detects potential inefficiencies. It provides users with general code metrics such as details on basic loop characteristics, the number of instructions, μ`ops`, and the number of XMM/YMM vector registers used. Cqa also allows users to obtain more in-depth information on the loop execution on the target architecture. For example, the tool provides a reliable front-end pipeline execution report, which is an estimated number of cycles spent during each front-end pipeline stage. The tool gives the same type of report for the back-end. Finally, Cqa provides a cycle estimate of loop body performance under ideal conditions: all operands in L1, no branches and infinite loop count (steady state behavior).

Cqa is able to report both low and high level metrics/reports (figure 5.3). For example, when a loop is not fully vectorized, the high level report provides a potential speedup (reachable if all instructions were vectorized) and corresponding hints (compiler flags and source transformations). For the same loop, some low level metrics/reports show the breakdown of vectorization ratios per instruction type (loads, stores, ADDs, etc.) giving the user a more in-depth view of the issue.

Cqa is operational on Intel 64 micro-architectures from Core 2 to Ivy Bridge.

5.3.3 Decan: Differential Analysis

In Pamda, Decan is used for quantitatively assessing performance pathologies impact. The general idea is fairly simple: a given pathology is associated with the presence of a given subset of instructions, for example div/sqrt operations, then Decan generate a binary version of the loop in which the corresponding instructions are deleted or properly modified. This altered binary is measured and compared with the original unmodified version.

```
Unroll factor: 1 or NA

************************************
            Back-end
************************************
      P0     P1    P2   P3   P4   P5
FU    FP ×/÷ FP +  LD1  LD2  ST   OTH.
Uops  18.00  17.00 9.50 9.50 3.00 6.00
Cycles 43.00 17.00 9.50 9.50 3.00 6.00

Cycles executing div or sqrt
instructions: 20-43 (second value used
for L1 performances)
Longest recurrence chain latency
(RecMII): 3.00

************************************
       Vectorization ratios
************************************
All      : 0%
Load     : 0%
Store    : 0%
```

```
Mul      : 0%
add_sub  : 0%
Other    : 0%

************************************
      Vector efficiency ratios
************************************
All      : 25%
Load     : 25%
Store    : 25%
Mul      : 25%
add_sub  : 25%
Other    : 25%

************************************
          If all data in L1
************************************
cycles: 43.00
FP operations per cycle: 0.81 (GFLOPS
at 1 GHz)
(…)
Cycles if fully vectorized: 21.50
```

Figure 5.3: Cqa output.

Table 5.2: Decan variants and transformations.

Variant	Type of SSE/AVX instructions involved	Transformation
LS	All arithmetic instructions	Instruction deleted
FP	All memory instructions	Instruction deleted
DL1	All memory instructions	Instruction operands modified to target a unique address
NODIV	All division instructions	Instruction operands modified to target a unique addresses
NORED	All reduction instructions	Instruction deleted
S2L	All store instructions	Converted into load instructions
NO_STORE	All store instructions	Instructions are deleted

Using Decan features, Pamda generates altered binaries, thereby splitting performance problems between CPU, memory, and OpenMP issues. Table 5.2 presents a range of loop variants used within the methodology discussed in Section 5.4. The variants are described in more details in Chapter 4.

5.3.4 Mtl: Memory Tracing Library

Within Pamda, Mtl provides specific analysis of pathologies related to data access patterns in particular stride values, alignment characteristics, data sharing issues in multi threaded codes, etc. Mtl works by tracing addresses and by generating compact representations of data access patterns. Mtl is not limited to innermost loops but directly deals with multiple nested loops, allowing to detect more subtle pathologies: for example, row major instead of column major accesses for a Fortran array (stored column wise) are automatically detected. To perform these analysis, Mtl uses the Maqao Instrumentation Language (Mil) [34]. This language makes the development of program analysis tools based on static binary instrumentation easier. In fact, MIL is a specific language for object-oriented and event-directed domains to perform binary instrumentation at a high level of abstraction using structural objects (functions, loops, etc.), events, filters, and probes.

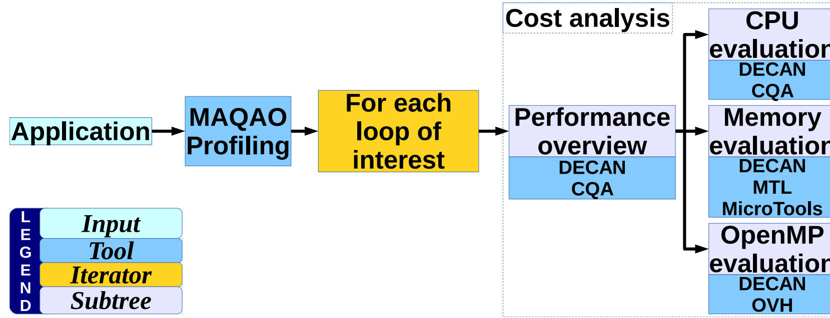


Figure 5.4: PAMDA overview.

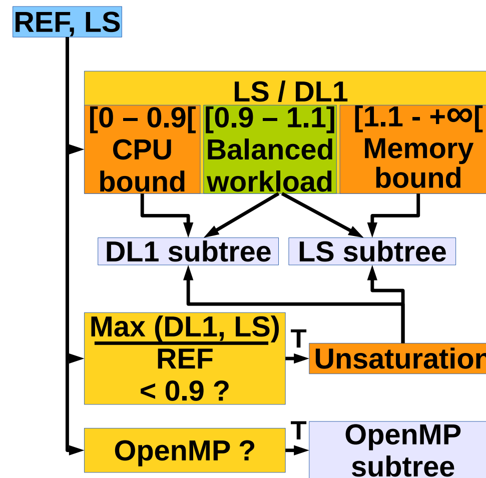


Figure 5.5: Performance investigation overview. T means the condition is True, otherwise it is False (F)

5.4 Recipe: PAMDA Tool Chain

Individual tools are the building blocks that PAMDA glue together through a set of scripts (cf all the diagrams). These scripts are under development but most of the principles have already been evaluated. Figure 5.4 presents PAMDA overall organization, which includes application profiling, cost analysis, structural checks, CPU and memory subsystems evaluation, and finally OpenMP evaluation for parallel applications. The current section describes PAMDA components.

5.4.1 Hotspot identification

To limit the processing cost, we focus on the most time consuming portions of the code. Our target loops are defined as the loops with a cumulated execution time exceeding 80% of the total execution time. It should be noted that with such an aggregated measurement, we can end up with a large number of loops with small individual contributions. Such target loops are identified using `MAQAO sampling`.

5.4.2 Performance overview

The PAMDA approach divides performance bottlenecks into two main categories (Figure 5.5): memory subsystem and CPU. Then, their respective contribution to the overall execution time is quantified using DECAN transformations `LS` (assessing memory subsystem performance) and `DL1` (assessing CPU subsystem performance).

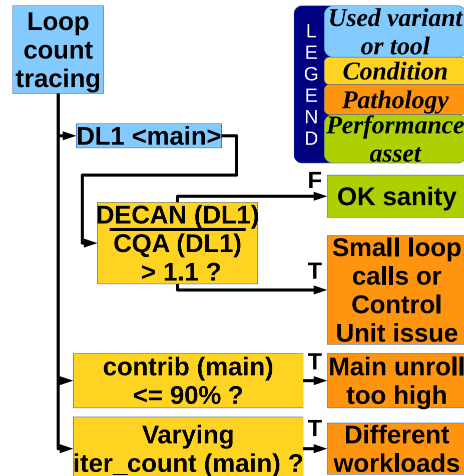


Figure 5.6: Detecting structural issues.

The ratio of these contributions reveals whether the loop is memory or/and CPU bound. Ideally, pipeline and out of order mechanisms insure that cycles spent for memory accesses and for arithmetic operations perfectly overlap: as a result, the time taken by `REF` should be the maximum time taken either by `LS` or `DL1`. In such a case, only the slower component needs optimizing. If the time taken by `LS` and `DL1` is similar, the workload is said to be balanced: optimizing both components is necessary to improve the loop's performance. Finally, when cycles taken by the memory and CPU components are poorly covered by one another (`unsaturation`), optimizing either of them can be sufficient to gain overall performance.

5.4.3 Loop structure check

Loop structure issues can be detrimental to performance, and may be detected using Decan loop trip counting feature. Indeed, in the case of unrolling or vectorization, peel and tail scalar codes may have to be generated to cover the remaining iterations. If too much time is spent in these peel and tail codes, this might indicate the unroll factor is too high in respect to the source loop iteration count. To detect such cases, loop trip counts for each version (peel/tail/main) are determined, and we can check whether the main loop is processing at least 90 % of the source code iterations.

In some cases, the number of iterations per loop instance may not be large enough to fully benefit from unrolling or vectorizing. This is easily highlighted by comparing the dynamic execution time of the DL1 Decan variant with the Cqa estimate, as the latter assume an infinite trip count. The difficulty to optimize such loops is exacerbated when the loop trip counts are not constant.

5.4.4 CPU evaluation

Besides data accesses, CPU performance may be limited by other pathologies such as long dependency chains (`deps`), reductions (texttRED), scalar instructions or long latency floating point operations (`div`): these pathologies can be detected through the combined use of Cqa and Decan (Figure 5.7). The front-end can also slow down the execution by failing to provide the back-end with micro-operations at a sufficient rate. Comparing their contribution to L1 performance (`DL1`) is a cost-effective way to identify such problems. Finally, Cqa can provide us with estimations of the

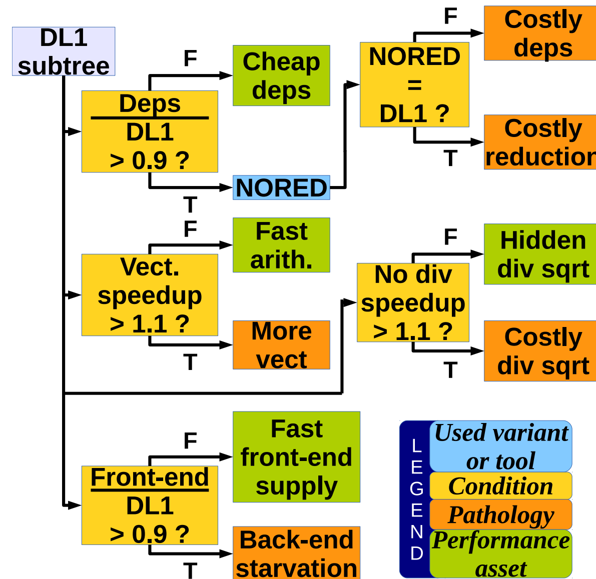


Figure 5.7: DL1 subtree: CPU performance evaluation.

Table 5.3: Bytes per cycle for each memory level (Sandy Bridge E5-2680).

Instruction	L1	L2	L3	RAM
vmovaps	31.74	15.05	10.81	5.10
vmovups	31.73	14.96	10.81	5.10
movaps	30.72	18.16	10.80	5.14
movups	29.53	17.07	10.79	5.23
movsd	15.67	11.55	10.61	5.36
movss	7.91	6.65	6.39	4.97

effect of vectorizing a loop. We precisely quantify CPU related issues, enabling us to reliably assess potential for optimizations such as getting rid of divisions, suppressing dependencies or vectorizing. This information can guide the user's optimization decisions.

5.4.5 Bandwidth measurement

Data access rates from different cache levels/RAM highly depend on several factors, such as the instructions used or the access pattern.

To this end, we generate microkernels loading data in an ideal stream case, testing different configurations for load operations, with or without various software prefetch instructions, and/or splitting the accessed data in streams to be accessed in parallel. We also force misaligned addressing for `vmovups` and `movups`. Finally, we use `Microlaunch` to run these benchmarks for each level of the memory hierarchy.

On our target architecture, 128-bit SSE load instructions could roughly achieve the same bandwidth as 256-bit AVX (Table 5.3) throughout the whole memory hierarchy. Except for `movss`, all instructions could attain similar bandwidths in L3 and RAM: only the type of instruction really matters for data accesses from L1 or L2, and data alignment is not as relevant as it once was.

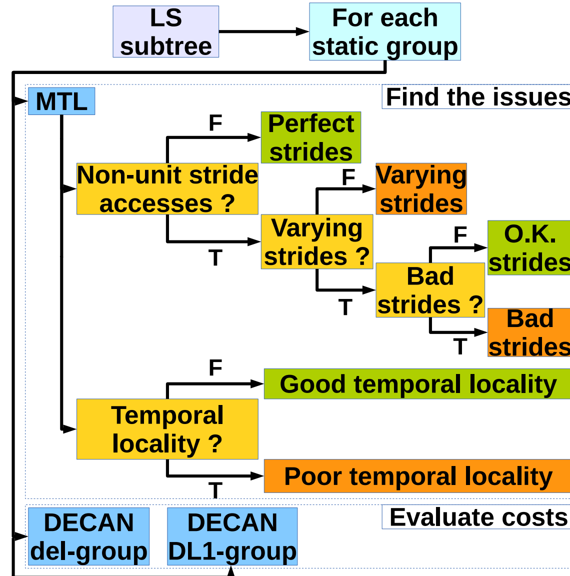


Figure 5.8: LS subtree: Memory performance evaluation.

5.4.6 Memory evaluation

Memory performance can be quite complex to evaluate. We use MTL to find the different access patterns and strides for each memory group (as defined by the grouping analysis [58]). Memory accesses typically are more efficient when targeting contiguous bytes, while discontiguous accesses reduce the spatial locality of data. The worst case scenario is having large and unpredictable strides, as hardware prefetchers may not be able to function properly. MTL also provides the data reuse distance, allowing the temporal locality evaluation of groups.

Once potential performance caveats are identified, we can use DECAN transformation `del-group` to single out offending groups and quantify their contribution to the `LS` variant global time. Comparing the bandwidth measured for each group with the bandwidth obtained in ideal conditions during the measurement phase may then provide us with an upper limit on achievable performance.

5.4.7 OpenMP evaluation

Some issues are specific to parallel programs using OpenMP (Figure 5.9). The standard deviation (`STD`) of the execution time for each thread points out workload imbalances. It is particularly important that no thread takes significantly more time than others to compute its working set, as loop barriers may then highly penalize stalls. Another issue is excessive cache coherency traffic generated by store operations on shared data. Transformation `S2L` converts all stores to loads: we can quantify coherency penalties by comparing `S2L` with `REF`. Furthermore, the OpenMP Overhead (OVH) module of MAQAO is able to measure the portion of time spent in OpenMP routines, providing an OpenMP overhead metric.

5.5 Experimental results

We applied our methodology on two scientific applications: PN and RTM. The analysis processes and test results are presented below.

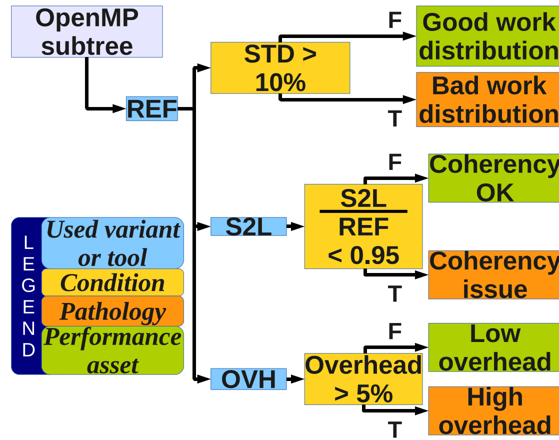


Figure 5.9: OpenMP performance tree: STD represents the standard deviation between threads while the OVH branch stands for OpenMP Overhead evaluation.

5.5.1 PN

PN is an OpenMP/MPI kernel used at CEA (French Department of Atomic Energy). Hot loops are memory bound and are ideal to stress tools dedicated to memory optimizations.

All tests are performed on a two-socket Sandy-Bridge machine, composed of two Intel E5-2680 processors with 8 physical cores each.

The profiling done on the initial MPI version of the code presents four loops, each consuming more than 8% of the global execution time each. Because of a lack of space, we only study the first one, but the three other loops have a similar behavior.

According to the methodology, the next step consists in gaining more insight on the loop characteristics through *performance overview*, hence, the `LS` and `DL1` Decan variants are used. The corresponding results shown in Figure 5.10 indicate a strong domination of data accesses, with the `LS` curve being well over the `DL1` curve and matching the `REF` one. Consequently, the investigation follows the `LS` subtree.

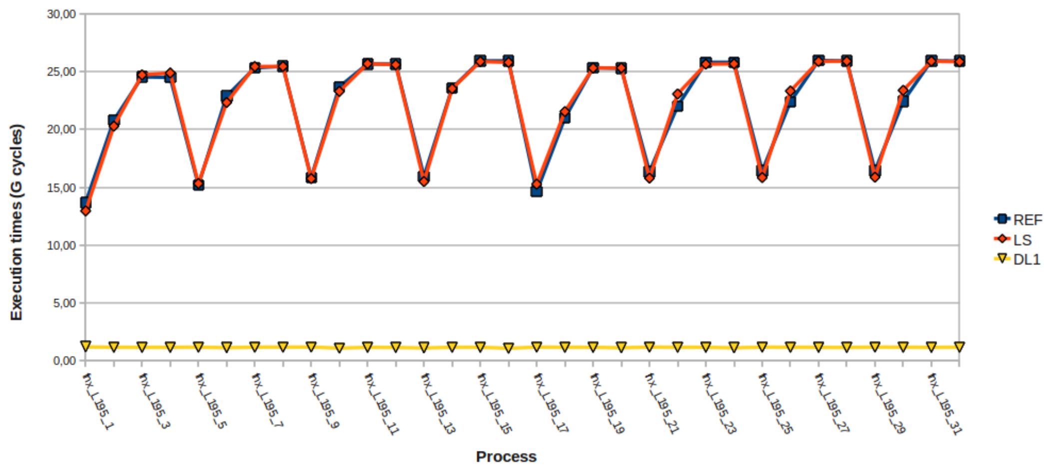


Figure 5.10: Streams analysis on PN. The REF curve corresponds to performance of the original code. The LS (resp. DL1) curve corresponds to the Decan variant where all FP instructions have been suppressed (resp. all data accesses are forced to come out of L1).

Table 5.4: PN MTL results for the three most relevant instruction groups.

Group	Instructions	Pattern
G1	Load (Double)	8*i1
G6	Load (Double)	8*i1+217600*i2+1088*i3
G5	Store (Double)	8*i1+218688*i2+1088*i3

In order to get more information on data accesses, we use MTL. Six instruction groups are detected but only three of them contain relevant SSE instructions dealing with FP arrays. The MTL results shown in Table 5.4 indicate a simple access pattern for group G1 (stride 1) and, for groups G6 and G5, more complicated patterns which need to be optimized. As a result, we are able in this step to characterize our memory accesses with precision. However, it leaves us with two accesses and no possibility to know which one is the most important. At this point, we return to our notion of ROI provided by Differential analysis and apply the DECAN `del-group` transformation for each of the three selected groups. The `del-group` results shown in Figure 5.11 clearly indicate that G6 is by far the most costly group: it should be our first optimization target.

With the finding of the delinquent instruction group, the analysis phase comes to its end. The next logical step is to try and optimize the targeted memory access. Fortunately, the information given by MTL reveals an interesting pathology. The access pattern of the instruction of interest has a big stride in the innermost loop (1088*i3) and a small one in the outermost loop (8*i1). In order to diminish the access penalty, we perform a loop interchange between the two loops, which results in a considerable performance gain with a speedup of *7.7*x at loop level and consequently a speedup of *1.4*x on the overall performance of the application.

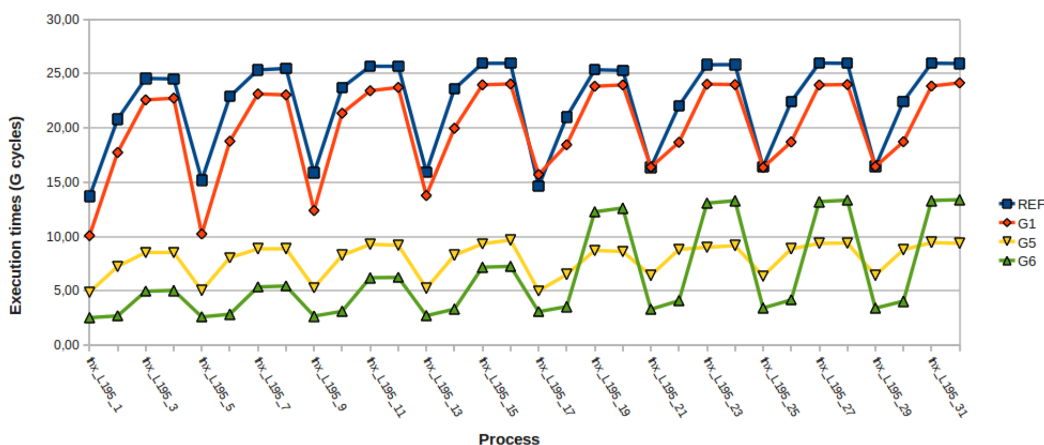


Figure 5.11: Group cost analysis on PN. Each group curve corresponds to performance of the loop while the target group is deleted. The original code performance (REF) is used as a reference.

5.5.2 RTM

Reverse Time Migration (RTM) [23] is a standard algorithm used for geophysical prospection. The code used in this study is an industrial implementation of the RTM algorithm by the oil & gas company TOTAL.

Our RTM code operates on a regular 3D grid. More than 90% of the application

execution time is spent in two functions, `Inner` and `Damping`, which execute similar codes on two different parts of the domain: `Inner` is devoted to the core of the domain while `Damping` is used on the skin of the domain. Standard domain decomposition techniques are used to spread the workload on multicore target machines. Since the grid is uniform, load balancing can be easily tuned by using rectangular sub-domain decomposition and by properly adjusting the sub-domain size.

All experiments are done on a single socket machine, which contains a quad-core Intel Xeon E3-1240 processor with a cache hierarchy of 32KB (L1), 256KB (L2) and 8MB (shared L3).

Step 1: The original version of the code is provided with a default non-optimized blocking. The first analysis on the OpenMP subtree reveals an imbalanced work sharing. A second analysis done at the level of the *performance overview* subtree shows that the code is highly bounded by memory operations. In order to fix this, we focus on the blocking strategy. As a result it turns out that the default block size is responsible for both the load imbalance between threads and the bad memory behavior. We can then select a strategy which provides a good balance at work sharing level as well as a good trade-off between the LS and FP streams. However, we note that, to obtain an optimal strategy, a more dedicated tool should be used.

Step 2: The second step of the analysis consists in going further into the OpenMP subtree and checking how the RTM code performs in term of coherency. As explained earlier, the structure of the code induces a non-negligible coherency traffic. Figure 5.12 shows experimental results after applying the S2L transformation on RTM. While the x-axis details loops respectively identified from `Inner` and `Damping`, the y-axis represents speedups over the original loops. The results indicate a negligible gain due to canceling potential coherency modifiers and a minimal gain, observed on two loops, due to a complete deletion of the stores. Consequently, we can conclude that maintaining the overall coherency state remains negligible, therefore, there would be no point in going further in this direction.

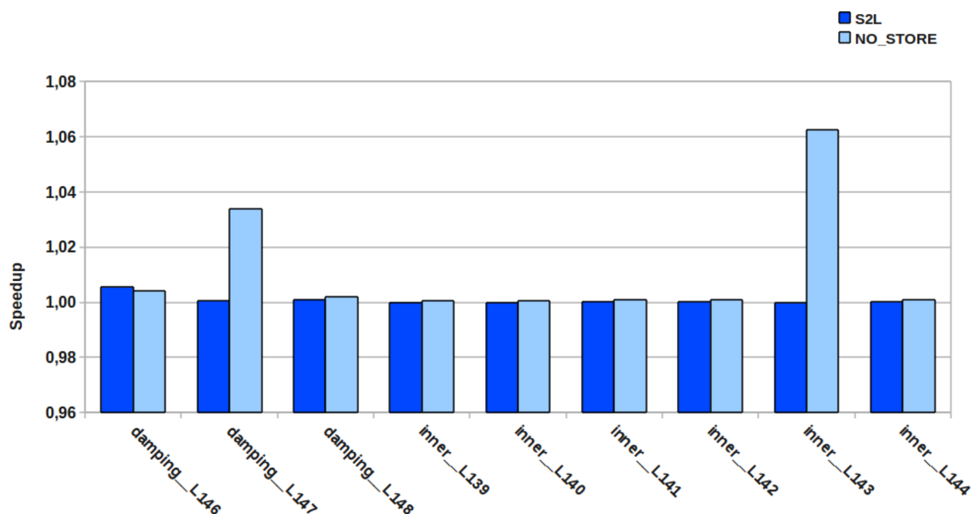


Figure 5.12: Evaluation of the cost of cache coherence protocol. The S2L variants show similar performance as their corresponding reference versions. The NO_STORE variants also show similar performances, except for two loops which present a relatively non negligible store cost.

5.6 Related Work

Improving an application efficiency requires identifying performance problems through measurement and analysis. Assessing bottlenecks impact on performance is much harder. To achieve that, most researchers consider a qualitative approach. TAU [87] represents a parallel performance system that addresses diverse requirements for the analysis and the observation of performance. Although performance evaluation issues require robust methodologies and tools, TAU only offers support to the performance analysis in various ways, including instrumentation, profiling and trace measurements.

Tools such as Intel VTune [51], GNU profiler (Gprof) [46], Oprofile [63], MemSpy [69], VAMPIR [77], and Scalasca [44] provide considerable insight on the profile of an application. In term of methodology Scalasca, for instance, proposes an incremental performance-analysis procedure that integrates runtime summaries based on event tracing. While these tools help hardware and software engineers find performance pathologies, significant manual or human operations still remain to improve software performance, for example to select instructions in particular part of a program.

PerfExpert [29], HPCToolkit [19], and AutoSCOPE [88] pinpoint performance bottlenecks using performance monitoring events. Furthermore, while PerfExpert suggests performance optimizations, AutoSCOPE extends PerfExpert by automatically determining appropriate source-code optimizations and compiler flags. Contrary to Pamda, the considered tools do not provide a methodology presenting the cost related to the identified bottleneck. ThreadSpotter also helps a programmer by presenting a list of high level advice without addressing return on investment issues: what to do in case of multiple bottlenecks? How much do bottlenecks cost?

Interestingly in [101, 102], the authors present an automated system that fingerprints the pathological patterns of the hardware performance events and identifies the pathologies in applications, allowing programmers to reap the architectural insights. The proposed technique is close to the current work and includes pathology description through microbenchmarks as well as pathology identification using a decision tree. However, in order to evaluate usual performance pathologies, Pamda additionally integrates pathology cost analysis.

The above survey indicates that performance evaluation requires a robust methodology, but traditional methods do not help much with coping with the overall hardware complexity and with guiding the optimization effort. Also, previous works focus on performance bottleneck identification providing optimization advice without providing potential gains. The previous factors motivate to consider Pamda as the only methodology combining both qualitative and quantitative approaches to drive the optimization process.

5.7 Summary

Application performance analysis is a constantly evolving art. The rapid changes in the hardware mixed with new coding paradigms force analysis tools to handle as many pathologies as possible. This can only be achieved at the expense of usability. At the end, application developers work with extremely powerful tools but they have to face significant differences and difficulties to use them.

This chapter illustrates the usefulness of performance assessment combining static analysis, value profiling and dynamic analysis. The proposed tool chain, Pamda, helps the user to quickly identify performance pathologies and assess their cost and impact on the global performance.

The goal in using Pamda is to make sure that the right effort is spent at each step of the analysis and on the right part of the code. Furthermore, we try to create some synergy between different tools by combining them in a unified environment. We provide some case studies to illustrate the overall analysis and optimization process. Experimental results clearly demonstrate the benefits introduced by Pamda.

Chapter 6

Decan: Assembly Level Re-writing Challenges and Limitations

6.1 Introduction

Decan is built on top of a software stack called Maqao. Maqao as a framework, targets the binary code of an application, it operates by disassembling the binary and building an intermediate representation (IR). Thus, Maqao enables to build performance evaluation tools as modules which exploit the intermediate representation. We identify two types of tools: 1) static analysis tools which exploit only the IR, and 2) dynamic analysis tools such as profilers which exploit both the IR and the binary rewriting capabilities provided by Maqao in order to instrument the code. Decan as a dynamic analysis tool makes use of both the IR and binary rewriting, and even pushes the later beyond what is usually done within classic tools.

Dynamic performance evaluation tools, whether they are trace based or sampling based, have only to worry about their intrusiveness and precision of measure, this is tied to their nature as tools for pure observation, their disturbance of the program they watch is limited to the intrusiveness of their probes. Decan acts differently. It performs what we qualify as *controlled alteration* of the code of the program in order to highlight performance issues. Indeed, as a binary transformation software, Decan cannot avoid introducing unwanted distortions within the variants it creates, the effect of which would be seen at runtime. Some are easy to pinpoint, but some others can be very tricky to highlight. In either case, it is necessary to keep their effect as small as possible at runtime, otherwise the results of the variants could be biased, and that would invalidate the comparison process.

This chapter summarizes the technical difficuties a tool such as Decan is likely to raise. In what follows we:

- Describe the architectures of both Maqao framework and Decan.
- Describe how we handled the passage from In-vitro mode to In-vivo mode.
- Describe how we handled parallel codes with Decan.
- Enumerate and categorize the most important side effects we encountered and dealt with during the design of Decan.

6.2 Decan Technical Design

Decan is built upon a binary manipulation framework called Maqao. The framework works at the binary level by reverse engineering the binary code in a similar

way as in a decompilation process. However, unlike the decompilation, which tries to reproduce the original source code, MAQAO stops after the construction of an *intermediate representation (IR)*. It then provides an API to manipulate the IR.

6.2.1 General Overview of the MAQAO Framework

Figure 6.1 depicts a general view on MAQAO architecture. We can identify three major blocs, namely: MADRAS, MAQAO *core* and a *modules* bloc. These are described below.

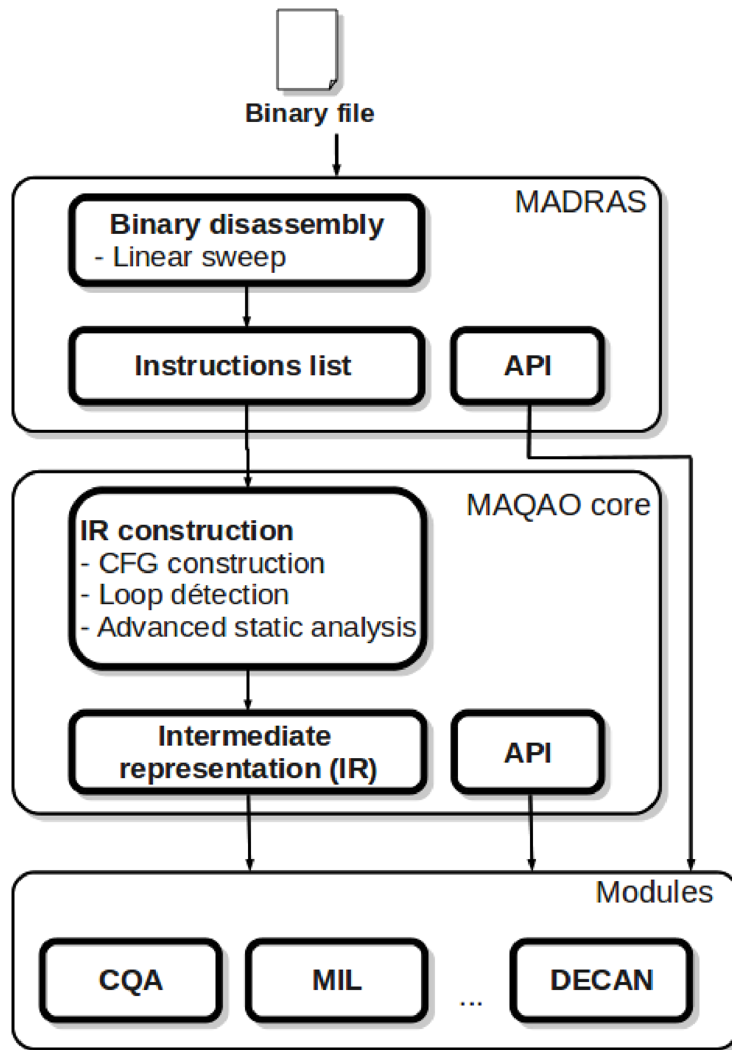


Figure 6.1: The MAQAO framework architecture

6.2.1.1 MADRAS:

We introduced in Chapter 4 MADRAS as the binary static rewriting tool DECAN uses to perform its transformations.We did not however justify that choice. Given that binary patching tools can be grouped into two families: *static* and *dynamic*, we could have chosen a dynamic binary rewriting tool as well. The reasons for discarding the second family lies in the opposition that exist between their properties and the constraints which need to be respected in order to be able to perform such analysis.

We enumerate two major dynamic rewriting frameworks, DynamoRIO [25] and Pin [66] which are built slightly differently. The major difference between the two being that, DynamoRIO copies each basic bloc of the application into a code cache it fully controls, and that the code is exclusively executed from that code cache. Pin, on the other hand, recompiles each code sequence (straight line of code) with a JIT compiler and puts the newly generated sequence in a code cache. The rest of the processing is pretty much similar except for optimization details that each framework uses in an attempt to mask its overhead. In a nutshell, a dynamic instrumentation framework runs at the same time as the target application. It also takes the leadover the control flow of the application, and this makes them intrusive and affects both execution time and memory state (caches). Moreover, it is difficult to know when this intrusiveness happens during the application lifetime, at the end, the use of dynamic patching to generate Decan variants is likely to introduce a significant amount of noise to seriously affect the comparison process. Still, the possibility to use these tools in future developments should not be discarded, as they are making substantial advancement into more transparency and less intrusiveness [26, 94].

In a different approach, static rewriting tools modify the binary code statically and generate a new modified version to be executed. The modified program is only active at runtime, the patching tool is no more needed. Therefore, the generated overhead generated is minimal compared to dynamic rewriting tools. Our choice to work with Madras is primarily justified by the fact that we were close to its development and could easily include our needs and specifications for Decan into its intrumentation capabilities. Other static binary rewriting tools are available such as PEBIL [62], and basically have the same capabilities as Madras.

Madras (Multi Architecture Disassembler, Rewriter and ASsembler)[93] is the binary code disassembler of Maqao, The tool translates binary code into a list of instructions that can be analyzed. Madras relies on a static disassembly method called *linear sweep*, the method consists of sequential disassembly of the executable, hence ensuring a full recovery of the binary content. Though, one serious drawback of the method is its lack of handling obfuscated codes (e.g. interleaving of data and execution code).

The instruction stream returned by Madras permits a great flexibility in instruction manipulation, in addition of offering access to all the parts of an instruction, it is possible to query on the properties of the instruction (e.g. load or store, branch or not, vector or scalar).

Madras also features a binary rewriting (also called patching) functionnality, which is widely used by Decan in its instruction transformation process. Below is an overview of its capabilities:

- Instruction insertion, deletion and modification: modification includes both opcode and operands changing.
- Function call insertion: the code is injected at binary level, thus the function call may change the contents of registers. In order to overcome this, Madras records the contents of all the registers before the call and restores them after it (spill-fill). It is also possible to have full control over the call injection and avoid the spill-fill of all registers.
- Global variables insertion: includes the possibility to inject new global variables in the code and to reference them through modified or newly inserted

instructions.

It is worth noting however, that the patching process does not come without complications. Probably, the most naticeable one occurs in cases where either the newly inserted instructions, function calls or modified instructions cannot be injected within the code area itself due to a lack of space. For example, if we attempt to replace a 4 bytes long instruction in the middle of the code (between two other meaningful instructions), with two 4 bytes long instructions, the part of the code located after the original instruction needs to be shifted by 4 bytes. The problem with shifting big chunks of the code, is the need to recalculate all the references. Within it, which is a tedious task prone to several mistakes due to the limits of static disassemblers. Madras avoids the shift with the following workaround:

- The basic block containing the non-fitting code is moved to the end of the binary, in its original location an unconditionnal branch is placed to point to the relocated block. The remaining place in the original location is filled with `nop` instructions.

- At the end of the newly relocated block an unconditionnal branch instruction is placed to redirect the control to its normal flow (the next basic block)

6.2.1.2 Maqao core

Maqao core groups the high level analyses which construct the *intermediate representation*. The following analyses are performed to construct the IR:

- Control flow analysis: takes the instruction stream returned by Madras and, by following the branch instructions, constructs the *control flow graph (CFG)* for each function. It also follows functions calls and constructs the *Call Graph (CG)*.

- Loop detection analysis: gets the *CFG* as entry and performs a Depth-first search (DFS) tranversal of it to construct the loop hierarchy. The algorithm used in the analysis is described in [98]. It detects both reducible and irreducible loops. The majority of analyses based on Maqao are loop centric including Decan.

- Advanced static analysis: Consists of several static analyses with the aim of polishing the CFG and CG, such as indirect branches resolution, static single assignment (SSA) and and groups detection which produces the groups described in Section 4.4.1.

Several useful data structures results from this module, these include: the Control Flow Graph (CFG), the Loop Hierarchy Graph (LHG) and the Call Graph (CG). A rich API is also exposed for the extension modules to exploit these structures.

6.2.1.3 Modules

Modules are performance analysis softwares which exploit the data structures and APIs offered by the lower parts of Maqao (Madras and Maqao core), Decan, being one of them, is built at this level. Two of the most noticeable modules are:

- *CQA[32]*: Is a static analysis tool; CQA analyses the quality of the generated code, and provides feedback on potential optimizations as well as a projection of new performances if these are applied.
- *MIL[31]*: Is a meta-language that enables fast prototyping of analysis tools such as profilers.
- *MTL*: MTL is a memory tracing library with advanced compression techniques.

We used a number of modules built on top of MAQAO, along with DECAN, within a unified performance analysis methodology which we introduced in Chapter 5.

6.2.2 DECAN architecture

DECAN is built as a module on top of MAQAO; its entry is the binary file of the target program as well as a configuration file that sets several parameters such as: the transformations to apply, the profiling method and the choice of the recovery strategy. Figure 6.2 illustrates the workflow of the tool. We can identify two major phases:

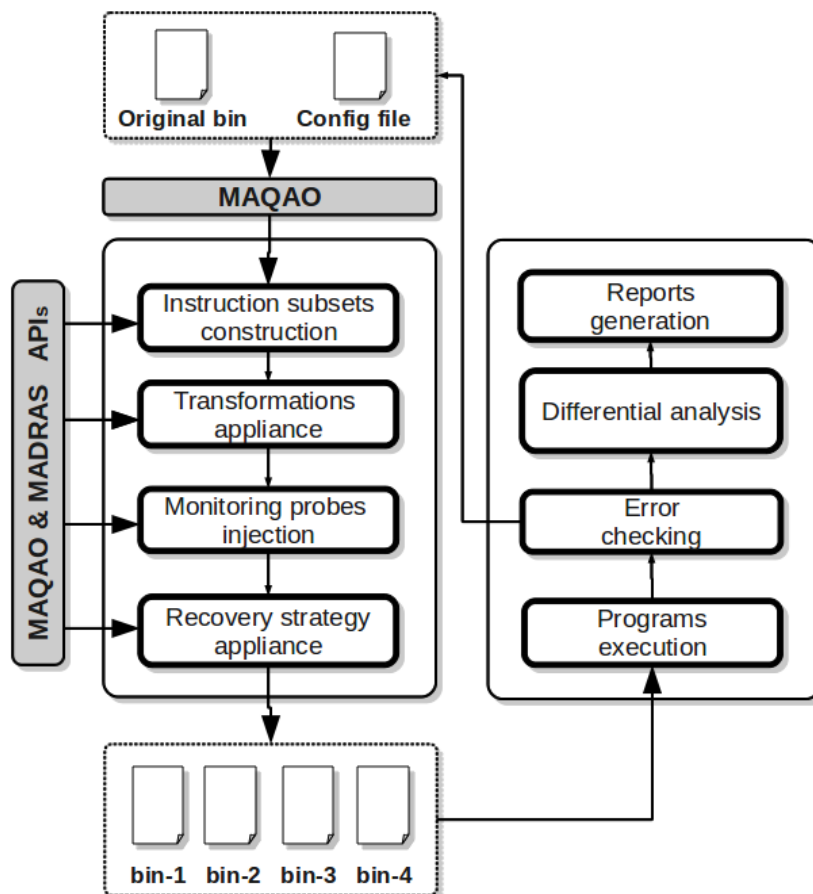


Figure 6.2: DECAN tool workflow

1) Variants generation is achieved by means of the following steps

- **Instruction subsets construction:** DECAN scans the target loop instructions and constructs instruction subsets. This construction is discussed in detail in Section 4.4.1.

- **Transformation:** with the help of MADRAS API, DECAN applies the appropriate transformations on the instruction subsets. Base transformations are described in Section 4.4.2.

- **Monitoring probes injection:** for its comparison process, DECAN has to collect various events (the principal one being execution time).DECAN implements its own runtime profile library. Monitoring probes are injected after the transformation process at the *entry* and *exits* of the loop. Measurement related issues are discussed in detail in Chapter 7. For each variant, the following events can be collected:

  - *Elapsed cycles*: we implement our elapsed cycles monitoring probe on the base of the `rdtsc` assembly instruction, an instruction provided in the Intel platforms which reads the value of a cycles hardware counter called *time stamp counter (TSC)*.
  - *Value profiling*: we implement probes which perform a value profiling on the loop where the numbers of *loop calls* and *iterations per call* are recorded.
  - *Hardware Counters*: DECAN uses a dedicated hardware monitoring library built within MAQAO to monitor counter values. The library is built on top of the Linux perf_event interface [15].

- **Recovery strategy appliance:** DECAN transformations, by altering the semantic of the code, may also alter its control flow and its normal behavior. In order to avoid such situations we set up two recovery strategies: *Instance mode* and *Recovery loop*. Recovery methods are discussed more in detail in Section 6.3

2) variants execution and results processing. The major steps of each are:

- **Variants execution**: the generated DECAN variants are executed following a vigorous experimental setup (described in detail in Chapter 7) in order to have reliable results for comparisons.

- **Error checking**: an error checking process determines whether the obtained results are coherent or not. Two examples of such situations are the case of a `LS` variant (with only memory operations) in which the execution time is greater than that of the `REF` variant, and second two variants of the same loop but with different iteration counts. Non-coherent results usually reveal the presence of previously unseen side effects which should be handled.

- **Differential analysis**: implements the core analyses which perform comparisons between events generated within variants.

- **Reports generation**: depending on the goals of the analyses, appropriate reports with feedback are generated. Data are also uploaded into a database for further use.

6.3 Dealing with Control Flow issues in DECAN

DECAN performs what we qualify as *controlled alteration* of the code. If not taken into account, this may cause serious issues at runtime, since the semantic alteration

is likely to change the control flow of the program from its normal behaviour into a random one. The consequences of a random control flow are critical both to the program (program crash, infinit loops, *etc*) and the analysis method (number of iterations of the loop may change between variants).

6.3.1 Data Dependent Control Flow

The main issue with semantic alteration is when the control flow is data dependent. DECAN performs well on loops with constant number of occurrences or of iterations, which makes the tool very suitable in the case of stencil codes. In such cases, modifying the core of the loop with DECAN has no impact, neither on the number of iterations nor on the number of occurrences (loop calls). However, other types of encountered codes present a different behavior. Indeed, for a particular code fragment, if the control structures (*if* statements and *loops*) depend on data computed during the execution of the code and not on some invariant constants, the alteration of computation can change the original orientations of those structures.

We differentiate two types of control alterations which may arise either separately or at the same time:

- *inner control flow alteration:* corresponds to the case where the inner control flow of the loop transformed by DECAN is altered, which causes its number of iterations to change.
- *outer control flow alteration:* corresponds to the case where the DECAN transformations on the intructions of a loop causes the control flow of the rest of the program to change.

6.3.2 In-vitro Mode

The earlier version of DECAN performed what we qualify as *In-vitro* execution, where the routine containing the target loop is extracted from its original context (the application) and executed separately. The following process was used[57]:

1. Dumping the memory context of the routine
2. Dumping the addresses of all the parameters of the routine.
3. Building a loader that maps the memory context of the routine and passes the parameters of the routine to the stack/registers.
4. Patching the application in order to directly connect to the loader and avoid the execution of the whole application.

The approach was interesting but we found some drawbacks:

- The process contained some flaws which led to program crash in several occasions.
- The context dump of the memory context does not restore the original state of caches. Therefore, the target loop may have a different behaviour than in its original context.
- The process does not prevent control flow problems inside the routine.

These drawbacks prompted us into developping other techniques to deal with control flow issues. These are discussed in the following section.

6.3.3 In-vivo Mode

The approach we adopted for our updated version of DECAN relies on the concept of *in-vivo* execution. This means that the target loop is executed within its original context. We had to handle both *inner control flow* and *outer control flow* issues. For that, we used the techniques described below.

6.3.3.1 Instruction black lists

In cases where data corruption is due to *inner control flow alteration*, we attempt to detect the instructions responsible for the data corruption and prevent them from being transformed. The detection method is based on a static analysis which detects the instruction chain involved in the control process, this is done through a buttom-up search on the loop DDG, starting from the jump instruction involved in the backward edge. All the instructions part of the dependency chain ending with the jump instruction are put into a black list. All the instructions of the black list are not transformed.

Because we only handled loops with a single basic block with DECAN, the analysis stays fairly simple to perform. Newer versions of the tool handle multi-block loops and perform a more advanced analysis.

The methods suffers from the drawbacks of static analyses, data manipulation in memory cannot be accurately tracked, which causes the analysis to fail if such cases happens.

6.3.3.2 Instance mode

In cases where data corruption is due to *outer control flow alteration*, we activate the transformed loop only at a particular loop call (loop instance). This technique, coupled with instructions black lists, is used in the majority of our analyses. It operates as follows:

- First, a profiling is done on the loop to capture the number of iterations of each loop call. We order the recorded trace and delimit deciles (we generally use deciles but it depends on the wanted precision). Thus, we obtain ten buckets.
- Second, the program is patched in what we call *instance mode*, it follows the logic shown in Figure 6.3. A test block is created before entering the loop. The block holds a counters which is incremented with each loop call. A specific loop call is considered as a threshold, as long as the threshold is not reached the original version of the loop is executed and the program continues its execution. When the threshold is reached, the transformed version of the loop is activated (with its probes). Once, it finishes its execution, the program is ended.
- In order to have a detailed image on the bahaviour of the loop, we take a loop call from each of the created buckets and use it as a threshold for the binary generated in instance mode.

This technique enable us to avoid dealing with outer control flow isses. The sampling over loop call allows to have a good coverage on the performance of the loop. The only drawback is that depending on the variation of the number of iterations, the process can take more or less time. If the number of iterations of the loop does not

change for all loop calls, then only one execution is needed, whereas if it varies a lot, we need to take several loop calls in order to preserve our precision over loop behaviour.

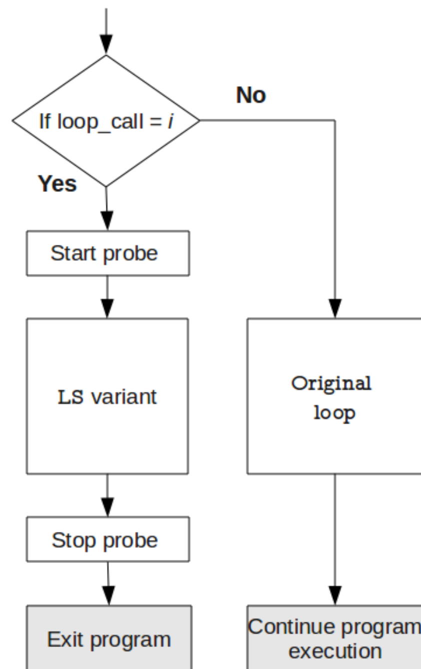


Figure 6.3: Flowchart showing the logic of the code generated by DECAN for the instance mode. As long as the loop call i is not reached the original version of the loop is executed. Once the loop call is reached, the transformed version of the loop is activated. The program is ended at loop exit

6.3.3.3 Recovery loop

Another approach to tackle *outer control flow alteration* problems is the injection of a recovery loop to restore the correct state of registers and memory. Basically, the original loop is not only replaced by the transformed one, but also duplicated; when the modified loop call is finished, the original version is called. The code generated by DECAN follows the logic described in Figure 6.4. The mechanism is as follows:

- *Context saving*: All the registers (general purpose, vector, flags and FPU) are saved.
- *Start probe*: The probe at the entry of the loop is activated.
- *Modified loop execution*: the transformed version of the loop is executed.
- *Stop probe*: The probe is deactivated at the exit of the loop.
- *Context restore*: All the registers are restored.
- *Original loop execution*: the original version of the loop is executed to recover the correct state of registers and memory.

This technique is particularly efficient, since its cost is just twice the cost of the loop. However, it supposes that the transformed loop does not change the memory

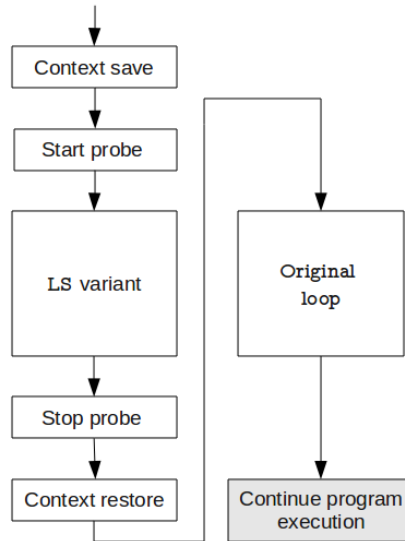


Figure 6.4: Flowchart showing the logic of the code generated by DECAN for the recovery loop mode.

state, otherwise the original version of the loop would work on corrupted data. Thus, stores need to be completely suppressed from the transformed version of the loop, in order to keep the memory state unchanged.

6.4 Extensions for Parallel Applications

Despite targeting primarily on-core performance issues, we came during the development of DECAN to analyze some parallel codes. Thus, we had to extend DECAN in order to handle them.

DECAN is agnostic to the programming model used in the application, what it really sees, are assembly instructions, threads and processes. The Differential analysis concept can be applied in different manners on a parallel code depending on the granularity at which the concept is to be applied: instruction, thread or process.

6.4.1 Shared Memory Codes

It consists of applying transformation process on loops inside parallel regions. The `OMP PARALLEL` directive of the OpenMP standard [30] is a typical example of such regions where each openMP thread executes a number of the iterations of the loop. For such construct, we set up two operatory modes:

- **Homogeneous processing of threads** The same transformed loop is executed by all threads. The flowchart (A) illustrated in Figure 6.5 shows the logic of the code generated by DECAN, each thread i has its own start and stop probes, but all execute the same `LS` variant. This is a forward extension of the sequential case, the only difference being that the monitored events are captured for each thread separately.

  Figure 6.6 shows the results of applying this operatory mode on four hot loops of the BT benchmark (from the NPB 3.0 [53]). Three DECAN variants (`DL1, FP` and `LS`) were generated and a parallel execution on four threads were

performed. We notice that each thread has its own reports on saturation. In the majority of cases, the threads have close saturations, thus we generally synthetize the results by taking the saturations of only one of them.

- **Heterogeneous processing of threads** Threads execute different variants of the transformed loop. An example of this is illustrated within the flowchart (B) in Figure 6.5, in which thread i executes the `DL1` variant whereas the other threads execute the `LS` variant.

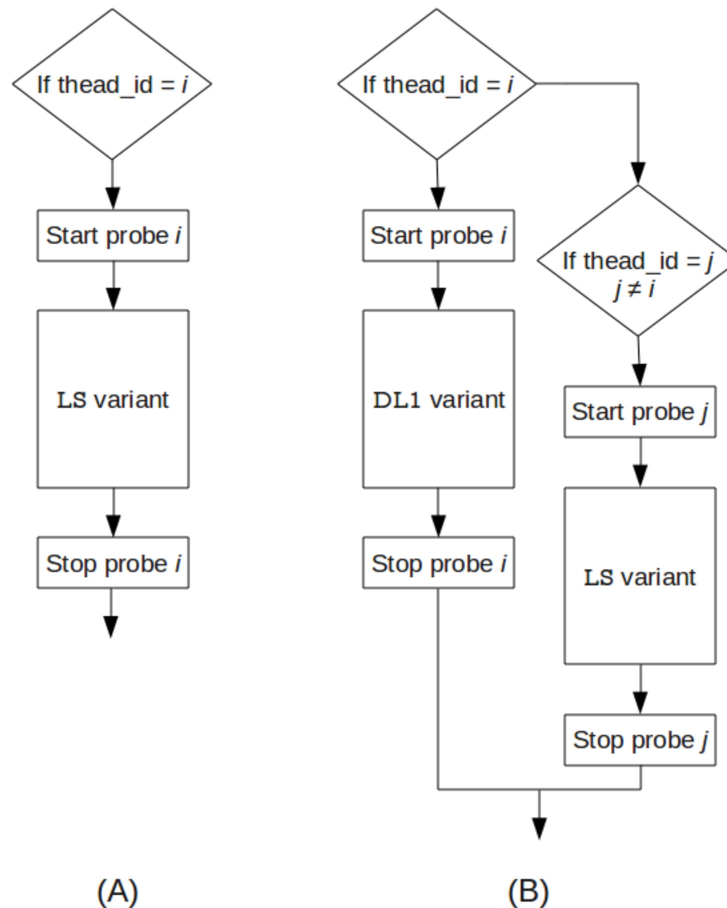


Figure 6.5: Flowcharts showing the logic of the code generated by DECAN for the two openMP operatory modes. Flowchart (A) illustrates the case where all threads execute the same variant, and Flowchart (B) illustrates the case where different threads may executes different DECAN variants

In our experiments on parallel codes we use the *homogeneous processing of threads* operatory mode. The second mode, was just set up as a prototype and could be the subject of future research.

DECAN should work on the majority of shared memory parallel models and frameworks. However, there are some subtelties that need to be handled differently for each framework. In the actual implementation of the tool, we added support for the OpenMP model. For that we managed the following details:

- DECAN handles only loops wrapped inside parallel constructs and without barriers, since it cannot handle well function calls inside the loop. The only parallel construct it handles for the moment is the `Parallel for`

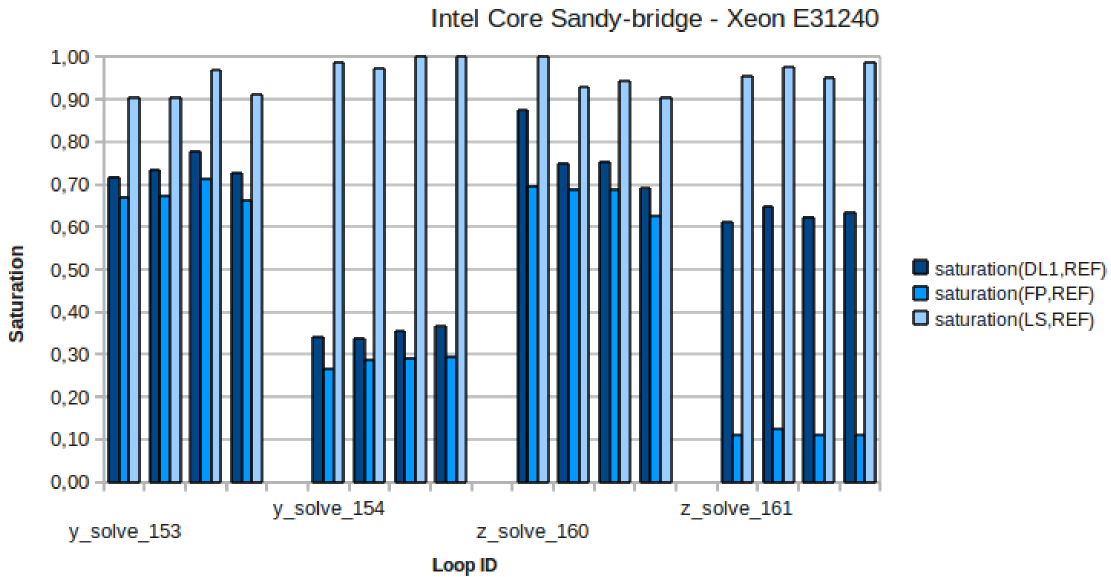


Figure 6.6: `DL1`,`FP` and `LS` variants on the four of the hot loops of the BT benchmark.

- Both in GCC and ICC, the loops wrapped inside the `parallel for` construct are extracted into independent functions and the OpenMP runtime manages the control flow redirection to them. These functions are easily identified by DECAN, in ICC a special label is attributed to each function and enables to identify its original function, whereas in GCC the functions are injected without labels just after the one they are extracted from.

6.4.2 Distributed Memory Codes

DECAN handles MPI applications in the same manner as it handles sequential ones. The main difference lies in the results reporting mechanism: in the MPI case, a report file is generated for each process in the loop. Still, loops containing standard MPI statements such as SEND and RECEIVE are skipped due to the delicate handling of function calls.

Figure 6.7 illustrates the results of applying DECAN on PN, an OpenMP/MPI kernel used at CEA (French Department of Energy). Each of the 32 processes of the loop has its own reports of the `stream` variant.

6.5 Code Alteration Side Effects and Workarounds

A side effect is any unwanted change in program behaviour introduced by a DECAN transformation. The intensity of the runtime distortion caused by the alteration is an indication of its harmfulness.

The detection of a side effect is not always obvious, we sometimes recognize already some potential side effects just by looking at the nature of the DECAN transformation itself. In other cases, only an abnormal runtime behaviour reveals the side effect.

The current section summarizes the main side effects we dealt with during the development of DECAN.

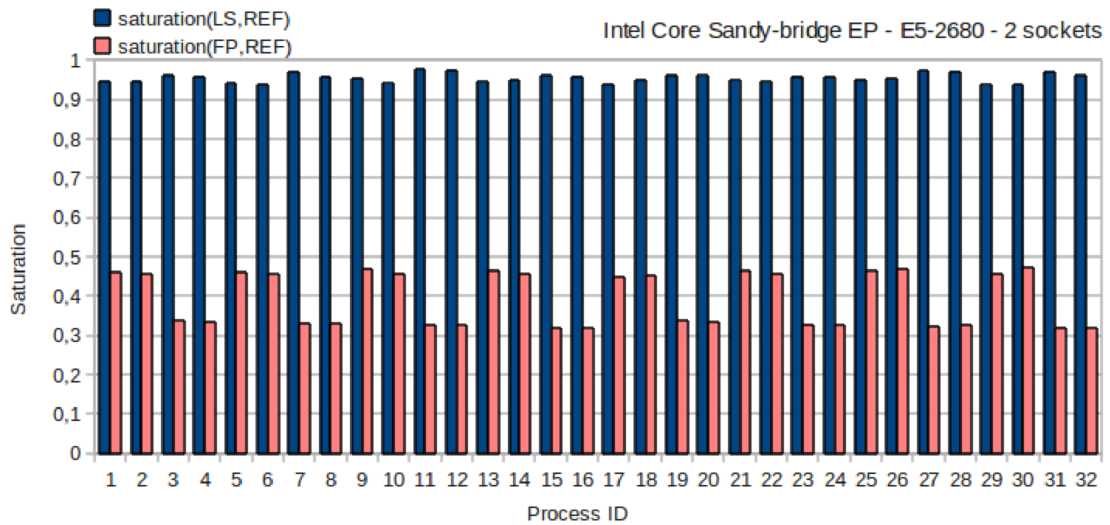


Figure 6.7: `Stream` variants on 32 processes of the PN application

6.5.1 Code Layout Sensitivity

By suppressing, modifying or replacing instructions, DECAN is likely to change the original layout (size and form) of the loop. Whether the change has an effect on performance or not, depends on the behavior of the transformed loop. Indeed, the total execution time of a loop is the aggregation between the time it spends in the processor front-end and the time it spends in the processor back-end. When the back-end is a bottleneck, the front-end time becomes negligible, but as the back-end time shrinks the front-end becomes more important.

The problem raised is that instruction transformations, such as suppressions, do not only affect the back-end but the front-end as well. The goal of instruction suppression is to assess the latency of the instruction, therefore, it becomes difficult to know how much gain is due to each.

We thus need to cancel the front-end effect due to instruction suppression. Fortunately, the front-end is more sensitive to instruction size. We thus preserve the original layout at the front-end level by replacing the suppressed instruction by a neutral instruction called a `nop` . The advantage of a `nop` instruction is that it only has a cost in the processor front-end, since it is evicted before reaching the back-end (on earlier Intel plateforms, the instruction passes through the back-end but has a small latency making it suitable for use also). Therefore, it would be possible to preserve the same code layout in front-end and obtain the desired effect in the back-end.

Globally for a transformation that alters code layout, three possible variants can be be constructed, based on the degree of preservation targeted:

- **Layout Unaware:** Is a non aware version on the code layout. It deletes instructions without paying attention to the loss in the number of `Uops` between the two codes. Consequently, the resulting loop is smaller than the original one in terms of `Uops`.

- **Layout Aware (One-Byte NOP):** Each deleted instruction is replaced by a standard one-byte `NOP` instructions (the `NOP` instruction is coded on one byte). The original number of `Uops` is preserved, but the code size and alignment

are not preserved.

- **Layout Aware (Muli-Byte NOP):** Each deleted instruction is replaced by a multi-byte NOP instruction (the code size of the NOP instruction is the same as the deleted instruction one). The original number of Uops is preserved as well as the code size and alignment.

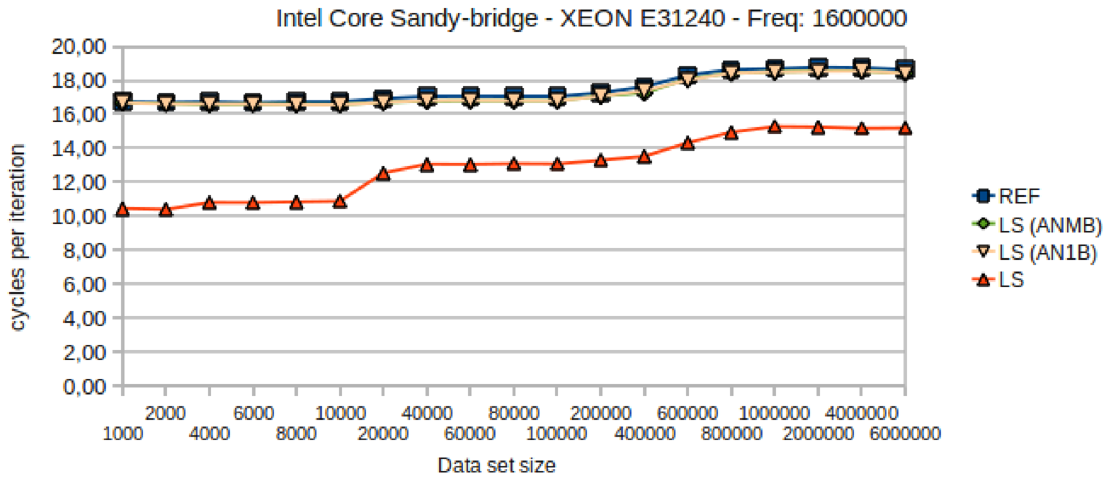


Figure 6.8: Performance of the `REF`, `LS(ANMB)`, `LS(AN1B)` and `LS` variants for the NR codelet `toeplz_4` on a low frequency execution

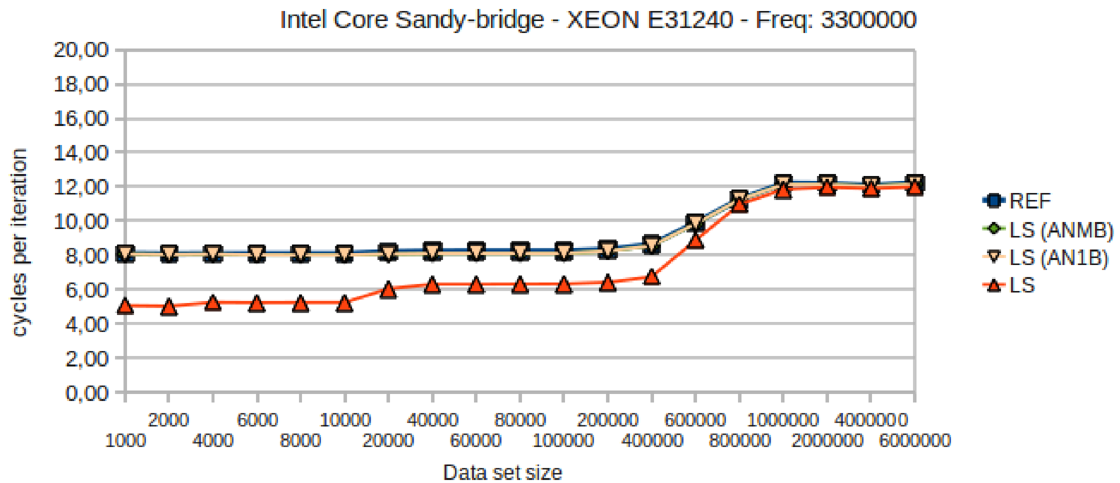


Figure 6.9: Performance of the `REF`, `LS(ANMB)`, `LS(AN1B)` and `LS` variants for the NR codelet `toeplz_4` on a high frequency execution

Figure 6.8 correspond to experiments conducted on `toeplz_4` an NR codelet, and gives an overview of the effect the change in code layout may have. The figure shows the cycles per iteration for the original version of the loop (`REF`) and three `LS` variants corresponding to the three possibilities of code layout described above. The LS variant is unaware of layout, its curve deviates from the REF one. Though, the deviation shrinks when the codelet is executed with big datasets, this matches our former statement on the Back-end becoming more consequent because of longer stalls. The two other curves AN1B and ANMB respectively correspond to the *one-byte* NOP and *multy-byte*NOP versions, these variants have the same performance

as the original REF which means that the deviation between `LS` and `REF` was due to layout effect induced by the deletion of `FP` instructions. We can notice another factor that affects the loop sensitivity to layout via Figure 6.9. It takes exactly the same codelet and variants, the only difference is that the machine frequency has been fixed to its biggest value "3300000" (in the first experiment the frequency was fixed to the smallest value "1600000"). Execution times of all variants have been divided by two, but more importantly for big data set sizes there is no deviation between `LS` and `REF` variants, the increase in frequency makes the instruction flow in the pipeline faster. However, the lower levels of memory (e.g. RAM) run at a different frequencies, lower than the processor. Therefore, the wait for data references located in those level becomes more significant and completely covers the effect of layout.

Other tricky issues happen at the processor back-end level when instructions are suppressed. These mainly include mainly the different buffers (ROB, PRF, *etc*) located within the back-end, and led to pipeline stalls when they are full. Instruction suppression hence releases some of the pression on them and leads to gain in execution time. Unfortunately, the granularity is too small, and we do not know for the moment how to handle such cases.

6.5.2 Data Dependence Alteration

One critical point in the instruction trasformation process of DECAN is the preservation of the original dependencies between instructions. We seek to preserve the original dependencies of the instruction if it is suppressed, modified or replaced. At first, we did not give much attention if our transformations created new dependences, we thought that the later would have a negligible effect. This allowed us to put some automation into our transformations, for example the same transformation we created for two operands SSE instructions were used for three operands AVX instructions. It turned out that some of the automated transformations introduced some depedencies which biased significantly our results.

Original instruction	VMOVHPD 0x2912d3c0(%R15),%XMM6,%XMM2
Transformation	Instruction subset ={LS}, Transformation={DELETE}
Modified instruction (V1)	VXORPS %XMM2,%XMM6,%XMM2
Modified instruction (V2)	VXORPS %XMM6,%XMM6,%XMM2

Table 6.1: Example of transformation which alters data dependency between instructions. The resulting instruction (V1) adds new dependencies whereas (V2) preserves the original ones

The transformation described in Table 6.1 illustrates the issue. The table shows two versions of the DELETE transformation on the `vmovhpd` instruction, the first version (V1) was our initial solution constructed by an automated process and consisted of replacing the memory operand with the destination register. The transformation targeted the LS subset and was supposed to produce an `FP` variant of the loop. A fragment of the original assembly code of the loop is shown in Table 6.2, along with the original and newly created dependencies related to the instructions `vmovhpd`. We notice that the initial dependency between instruction 8 and 1 is a *WAW* on $\%XMM2$, the transformation introduced a new true dependency *RAW* on the same register. The same goes for instructions 9 and 15.

The FP variant has predictable performance. We are able, with our static analysis tool, CQA to predict its performance with a less than 10% error. Therefore, we compared the real and estimated execution times of the variant. The results are shown in Table 6.3. We noticed a big gap between the real measure (134 cpi) and the static estimation (87 cpi) for the FP(V1) which was surprising. We also verified the recMII(*minimum initiation interval due to recurrence constraints*), a known metric used within the modulo scheduling algorithm [48] and provided by CQA, and found a significant difference between the REF (3 cpi) and FP(V1) variant (55 cpi). These observations led to a review of the generated code and a modification of the transformation. The modification consisted in replacing the first source operand by the second source operand instead of the destination operand (modified instruction (V2) in Table 6.1). The modification had a positive effect. We noticed that the execution time of the corresponding variant FP(V2) (90 cpi) was close to the estimated execution time (87 cpi) which is the usual behaviour, more over, we recovered the initial recMII of 3 cycles, which means that our critical dependency chain was preserved.

This issue made us aware of the risks that automated transformations can bring. We concluded that a validation suite made of small micro-benchmarks would help us to detect the cases where a transformation tested on an instruction set and used on a newer modifed one, fails to bring the desired behaviour.

Assembly code
1. **VMOVSD 0xa0(%R15),%XMM2**
2. **VMOVSD 0xd0(%R15),%XMM1**
3. VMOVHPD 0xb8(%R15),%XMM2,%XMM3
4. VMOVHPD 0xe8(%R15),%XMM1,%XMM15
5. VMOVUPS -0x70(%RBP),%YMM14
6. VMOVSD 0xa8(%R15),%XMM6
7. **VMOVSD 0xd8(%R15),%XMM5**
8. **VMOVHPD 0xc0(%R15),%XMM6,%XMM2**
9. **VMOVHPD 0xf0(%R15),%XMM5,%XMM1**
10. VMOVSD 0xe0(%R15),%XMM6
11. VINSERTF128 $0x1,%XMM15,%YMM3,%YMM7
12. VMOVUPS -0x50(%RBP),%YMM15
13. VSUBPD %YMM7,%YMM14,%YMM3
14. VMOVSD 0xb0(%R15),%XMM7
15. **VMOVHPD 0xc8(%R15),%XMM7,%XMM5**

Initial dependencies	**Added dependencies**
8. WAW(8,1,XMM2)	**RAW(8,1,XMM2)**
9. WAW(9,2,XMM1)	**RAW(8,1,XMM2)**
15. WAW(15,7,XMM5)	**RAW(15,7,XMM5)**

Table 6.2: Code fragment of a loop extracted from POLARIS application. With an emphasize on initial and added dependencies on some instructions after transformation application

Code version	Real exec time (cycles/iteration)	Estimated exec time (cycles/iteration)	recMII (cycles/iteration)	Saturation
REF	156	87	3	1.00
FP (V1)	134	87	55	0.86
FP (V2)	90	87	3	0.54
Application:POLARIS - Function:grade2c - Loop:2126				
Xeon E31240 Sandy Bridge 1600000-**3300000** Mono socket - L1:32 - L2:256 - L3:8192				

Table 6.3: Comparison between the performance of REF variant and two versions of FP variant

6.5.3 Instructions with variable Latencies

One of the main relaxation points enabled by the use of Differential Analysis is the destruction of code semantic. That makes the values in the registers no longer correct. Still, this relaxation is only, possible if instruction latencies are fixed for all operands ranges (the restriction concerns flaoting-point instructions only as memory instructions latencies vary depending on data location), otherwise, the propagation of random values within the code will lead to random latencies and break the comparison process. Fortunately, in recent architectures, and particularly in the case of the base of our DECAN tool, the Intel x86, most instruction latencies are standardized with fixed latencies. The small fraction of instructions with variable latencies are not used within scientific codes, except for two instructions ***division*** and ***square-root***. For these two, if operands values change, the latency of the instruction changes too. Thus, we had to provide a special transformation which would ensure a stable behaviour.

Original code	VMOVUPD 0x20(%RAX,%RCX,8),%YMM3 VDIVPD %YMM0,%YMM3,%YMM7
Transformation	Instruction subset ={LS}, Transformation={DELETE}
Generated code (V1)	VXORPS %YMM3,%YMM3,%YMM3 VDIVPD %YMM0,%YMM3,%YMM7

Code version	Execution time (cycles per iteration)	Saturation
REF	191.12	1.00
FP (V1)	59.60	0.31
Application:POLARIS - Function:mind2 - Loop:2943		
Xeon E31240 Sandy Bridge 1600000-**3300000** Dual socket capable - L1:32 - L2:256 - L3:8192		

Table 6.4: Creation of an `FP` variant on a code which contains a division operation

We can see the difference between the forward and controlled handling of divisions and square-roots in Tables 6.4 and 6.5. In the forward transformation (Table 6.4), no special processing is reserved for the division instruction. Deleting the `vmovupd` causes the second source operand of the division `XMM3` to be set to zero. This is already problematic since it can generate FP exceptions (division by zero

Original code	VMOVUPD 0x20(%RAX,%RCX,8),%YMM3 VDIVPD %YMM0,%YMM3,%YMM7	
Transformation	Instruction subset ={LS}, Transformation={DELETE}	
Generated code (V2)	VXORPS %YMM3,%YMM3,%YMM3 VMOVUPD -0xa65(%RIP),%YMM3 VDIVPD -0xa4d(%RIP),%YMM3,%YMM7	
Code version	**Execution time** (cycles per iteration)	**Saturation**
REF(V2)	181.87	1
FP (V2)	89.61	0.49
Application:POLARIS - Function:mind2 - Loop:2943		
Xeon E31240 Sandy Bridge 1600000-**3300000** Dual socket capable - L1:32 - L2:256 - L3:8192		

Table 6.5: Application of FP stream transformation on a code which contains a division operation

and NANs) which can raise interrupts and trigger additional special code to handle the exceptions. In this case we notice that the resulting `FP` variant has near 30% saturation. Our solution consists in controlling the outcome of the division operation through its source operands. We, therefore, load the two source operands from a memory location as illustrated in the generated code of Table 6.5. The loaded values always are the same and the memory location does not change, this ensures a small additional latency. Still, the modification of the source operands deviates the latency of the division from its original output, which makes the comparison with the original code impossible. We handle this by applying the same controlling process (injection of load instructions) to the `REF` variant in order to obtain the same behaviour in the two variants. We notice, in this case, that the execution time of the `REF` variant changes relatively to the uncontrolled version (181 vs 191).

6.5.4 Instrumentation Side Effects

A number of side effects might be introduced by the instrumentation mechanism itself. We encountered such a case with the `DL1` variant. In the `DL1` variant, the static patcher creates a new binary section for the global variables which should replace the load and store operands of memory instructions. It also places the section at the end of the binary file. During runtime, we noticed that the execution time of the `DL1` variant was far bigger than the `REF` variant. After an extensive investigation, we found that the global variables injected by the patcher were located on an unloaded memory page, and caused a costly `TLB` miss. We later solved the case by adding a first touch on the new variables before entering the transformed loop.

6.5.5 Floating-point Exceptions

The semantic alteration caused by instruction transformations are likely to change the numerical values calculated within the loop. In a number of cases, this already led to generating floating-point exceptions (e.g. denormal numbers, underflow, overflow). The problem with FP exceptions is that they trigger special software handling

mechanisms through interrupts. Once an FP exception occurs, an interrupt is immediately triggered and a system function is called to handle the exception.

FP exceptions harm the comparison process of DECAN. It is not possible to know how many exceptions happen in each variant. Each time they are triggered, several functions inside the OS are called, and the execution time of the handler is not negligible, especially in the case of small loops.

Our solution to this problem was simple. It consists in turning off the interrupt triggers before entering the loop and activating them again after its execution. On Intel x86 architectures, the interrupts can be controlled through the `MXCSR` register. The register holds a flag for each interrupt type which enables to either activate or deactivate it.

6.5.6 Wrap-up: Side Effects Sources

In the previous section, we highlighted some of the unwanted effects that might arise in the variants after code transformations. We therefore set up the root cause categories from which the side effects might happen. It is worth noting that the categories are likely to be incomplete as they are only derived fromprevious experiences.

6.5.6.1 Code Level

Refers to specific properties of code that result in an unwanted runtime behavior in the variant. The most critical being instructions involved in the control flow of the applications. This is obviously inherent to all kinds of codes, and should be handled appropriately.

6.5.6.2 Hardware Level

Technically, almost every component in the micro-processor can be a source of unwanted side effects. In a utopian transformation process, the effects to be highlighted only are the reactions of the hardware towards the properties of instructions themselves (latency et cetera). But in practice, the reactions of the hardware to the change in instructions, such as the layout change case discussed in Section 6.5.1, add up some effect in the equation. These are also inherent in general, few are the cases where we can completely mask or undo their effect, therefore, the best strategy would be to try and reduce their effects as much as possible.

6.5.6.3 Instrumentation level

Effects on the patching process range, they range from simple implementation details such as the jump instructions introduced to enter the patched areas, which harm performance if they are misplaced, to more complex ones such as probe nature and placement (discussed in Chapter 7). In a nutshell, in some cases, these are just the limitations of instrumentation itself, we just cannot do better. In other cases, they rather are mistakes or bad choices of instrumentation that can be avoided (see cases discribed in sections 6.5.4 and 6.5.2).

6.6 Summary

Within this chapter, we have introduced the overall architecture of DECAN and MAQAO, the framework on top of which DECAN is built. We detailed two important features introduced in our version of DECAN: first how we handled major issues related to control flow alteration, and how we switched from an *in-vitro* mode to an *in-vivo* mode, and second how we handled the transformation process in parallel programs. We also exhibited the unwanted side effects that DECAN might introduce, and showed their effects on the comparison process. In all encountered cases, we were able to provide workarounds which reduced their effects.

CHAPTER 7

Tackling Measurement Precision, Stability and Probe Intrusiveness

7.1 Introduction

Measurement is a widely used technique in application performance analysis. Mainly, it consists of monitoring hardware generated events during the execution of a particular portion of code. Analogously to the observations performed on physical entities where measurement instruments and captors are used, and in order to observe the behavior of a program, we use a special piece of code that we call a *probe*. Furthermore, event measures can be obtained with two different measurement methods:1) *exhaustive measurement*, where a code area is delimited with markers in order to measure a number of events within the delimited area. 2)*sampling*, which consists of taking event values only at particular points, the points can be either defined as time intervals, particular points in the code, event count (threshold) or a combination of the three. Within this chapter, we only focus on *tracing*, since it was the method we used in all our measurements for *Differential Analysis*.

Differential analysis is a comparative approach that works at binary loop level. This makes it very sensitive to measurements. Indeed, being able to say that a particular event differently behave in two DECAN variants is conditioned by the correctness of measures. The approach still relies on execution time as the principal event for the comparison process, because it is a known accurate and stable event. However, the integration of other events, which might be more sensitive (*eg*: memory related events) needs a primary step of investigation. This chapter globally describes how we dealt with measurements in DECAN, it also summarizes the investigations that we conducted on a variety of events in order to see which opportunity they offer within the comparison process of Differential Analysis.

In exhaustive measurement, three factors have a big impact on the quality and accuracy of measures: *stability*, *precision* and *intrusiveness*:

1. **Stability** (we may also refer to it as *measurement bias*) is related to the reproducibility of measures; it is not specific to performance analysis, but common to all measurement purposes (*e.g.* the true gain of an optimization would hardly be assessed if the measurements were unstable). Measurement stability is present in the majority of experimental setups used in a variety of tasks: program performance analysis, optimization validation, *etc.* If ignored, these setups may lead to serious mistakes such as the validation of an optimizationthe results of which in reality is a highly biased measure. For *Differential Analysis*, a biased measure may lead to the assessment that two variants are different when they are not(or the opposit) thus invalidating the comparison process itself.

2. **Precision** is related to probe placement and lightness. Precision here means

the ability to only measure the events of the targeted code area. The more probes are close to the area to measure, the better is the precision. Some trade-offs are however necessary. A probe in itself is a source of noise if the area to monitor does not generate enough events to make this noise negligible.

3. **Intrusiveness** is related to probe quality. As stated above the probe in itself can be a source of events, which cannot be differentiated from those of the monitored area, and may even fight with them on hardware resources (*e.g.* cache lines). Intrusiveness is just another perspective to deal with measurement precision, and aims to quantify and subtract the noise instead of minimizing it.

7.2 Events of Interest

The main event we rely on in the comparison process of *Differential Analysis* is *execution time*. The event is obtained through the Time Stamp Counter TSC which can be accessed quickly with the `rdtsc` instruction (more detail on this in Section 7.5.2) and the accurately measured. Nonetheless, we are interested to know if other events provided by hardware counters deliver reliable values that can be used for comparisons as well. We are particularly interested in memory related events. Thus, the events we depicted in our study correspond to a number of hardware counters we found in the Intel x86 Sandy-Bridge micro-architecture. Table 7.1 provides the list of counters we depicted; they fall into the following categories: *elapsed cycles*, *instructions retired*, *memory traffic* counters and *cache hits*. We use the terms *event* and *counter* interchangeably since the term event reflects a more abstract view of what the counter monitors.

Metric	Category	Alias
CPU_CLK_UNHALTED_CORE	elapsed cycles	CCUC
INSTR_RETIRED_ANY	instruction retired	IRA
L1D_REPLACEMENT	memory traffic	LR
L2_LINES_IN_ALL	memory traffic	L2LIA
L2_TRANS_L2_WB	memory traffic	L2TLW
L3_LAT_CACHE_MISS	memory traffic	LLCM
MEMLOAD_UOPS_RETIRED_HIT_LFB	cache hits	MURHL
MEMLOAD_UOPS_RETIRED_L1_HIT	cache hits	MURL1H
MEMLOAD_UOPS_RETIRED_L2_HIT	cache hits	MURL2H
MEMLOAD_UOPS_RETIRED_LC_HIT	cache hits	MURLLH

Table 7.1: List of hardware counters of interest, available in the Intel Sandy-Bridge micro-architecture

.

7.3 Experimental Methodology

It is difficult to validate hardware counter measurements against true values. Validation is sometimes possible for counters. Their values can be more or less statically determined from the code (*e.g.* instructions retired). In other cases, only a simulation reproducing the exact behavior of the micro-architecture would be able to

produce the correct values of the counters. Unfortunately, we do not know any simulator able to accurately replicate the micro-architecture. To validate our measurements, we had to set up reference values of the counters by using a software measurement workaround. The idea is to use small computing kernels that we call codelets. The advantage of using codelets is that we have a total control over their execution; it is possible to change the problem size and easily repeat the execution several times.

Our test suite is extracted from the Numerical Recipes (NRs) [82]. We chose 22 NR codelets as the base set for the experiments. These are part of a broader set introduced in Section 4.5.6.3. The chosen codelets are selected between well behaving codelets which operate on one and two dimensional arrays. For each codelet, the problem size (size of matrices) can be controlled, 21 points enable us to cover the entire memory hierarchy. This gives us a total of 462 points for the study.

Reference measures are obtained by putting the monitoring probes around a repetition loop for the codelet (Algorithm 4). The number of repetitions is increased until the measures reach stable values (normalized by the number of repetitions). They correspond to the most accurate and close to real values we can have with a measurement technique, and qualify as the *reference measure*. In contrast, *real measures*, those we seek to evaluate, have their monitoring probes inside the repetition loop (Algorithm 5).

Within our study, real measures are compared to reference measures in order to determine their accuracy. We consider a real measure to match the reference if its deviation is under 5%.

Algorithm 4 Reference measures

```
Data: codelet data
Result: codelet results
begin
    Monitoring_Start()
    for rep = 1 to NREP do
        Codelet()
    Monitoring_Stop()
end
```

Algorithm 5 Real measures

```
Data: codelet data
Result: codelet results
begin
    for rep = 1 to NREP do
        Monitoring_Start()
        Codelet()
        Monitoring_Stop()
end
```

All experiments were conducted on an Intel Sandy-Bridge E5-2680 dual socket processor with three cache levels having respectively: 32 kB, 256 kB and 20 MB.

7.4 Measurement Stability

Measurement stability has been addressed in a number of works, Mytkowicz and *al* [76] showed that in that *measurement bias* is unpredictable, by using program layout change through UNIX variables resizing. In other studies [97], V.M.Weaver and *al* showed that hardware interrupts increment most hardware counters leading to a non-deterministic over count, and described a correction method but did not test it. Curtsinger and *al* in [36] proposed a tool that would randomize several parts of the code and of the environment in order to make the program independent from environment bias. Execution times following a gaussian distribution can then be obtained. The method only considers as an event, the execution time; however the range of potential events is much larger, one may consider a wider use of hardware

counter events as well. Moreover, several external sources introduce variations which prevent from having a stable measure. these sources include the operating system [70], the program layout [76], the measurement overhead [103] and the hardware implementation details [96].

because of the various sources of variation previously cited, a certain instability is inherent to the measurement process. We adopted good practices in order to stabilize and minimize it as much as possible. On all our experiments, we adopted a rigorous experimental methodology by applying the following rules:

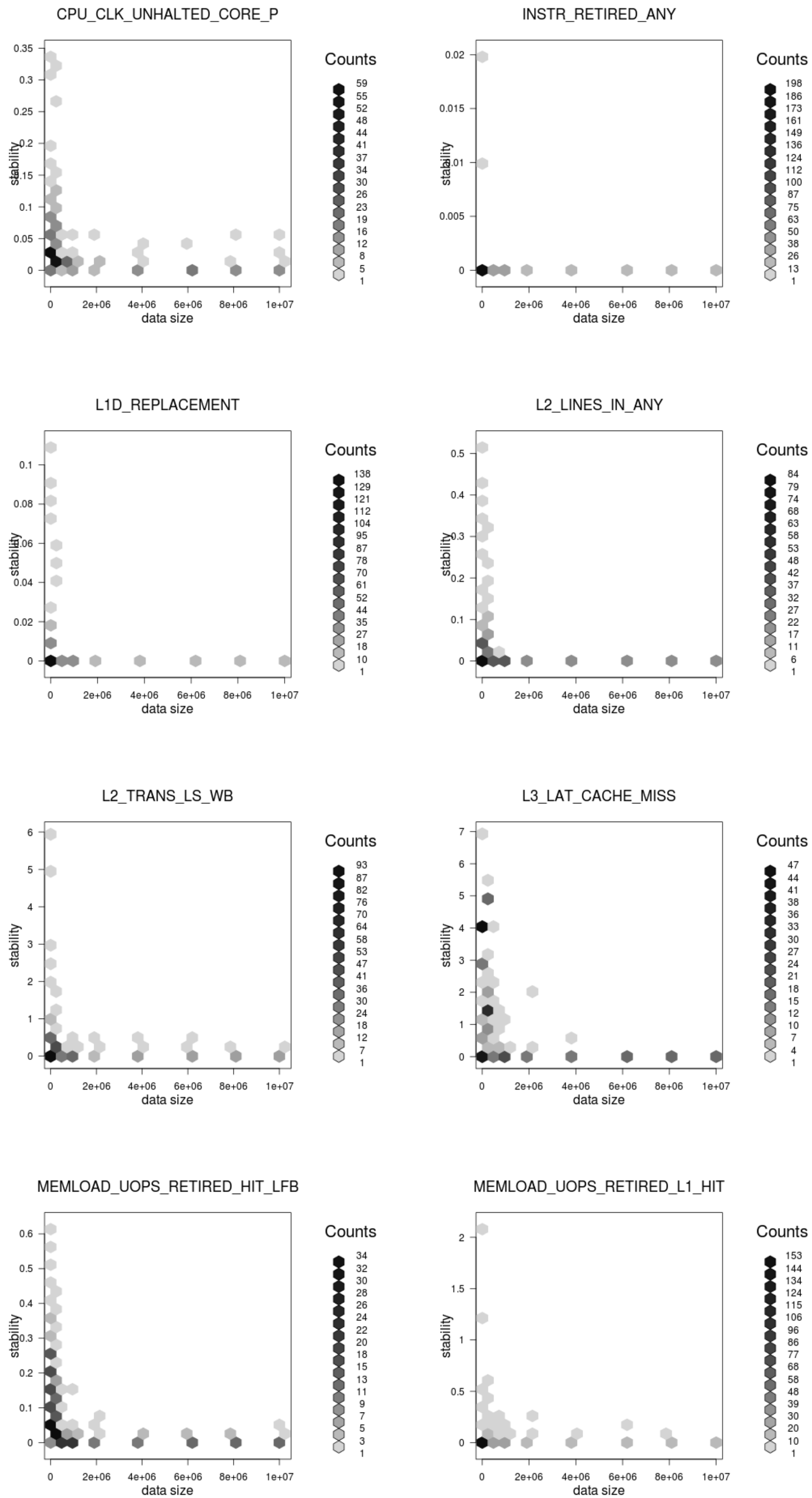
- Turn off all hardware features that may generate indeterminism such as Intel Hyper-Threading technology, turbo mode and power optimizations such as frequency scaling.
- Repeat the experiment several times in order to have a wide enough sample to be able to verify measurements stability. We call these repetitions *meta-repetitions*. The problematic part, however, is how to fix the number of meta-repetitions since a small number is likely to be insufficient to represent the distribution (if any). In our experiments, we fixed it to 31, the value has been given in [52] which is a good reference in application performance analysis.
- Use the same machine and same OS for a single experiment in order to ensure closer initial conditions.
- Our targets being loops, it is better to perform an individual measure on each loop call, as the loop behavior may significantly vary throughout loop calls.

7.4.1 Estimation

In order to assess the stability of our results, we established a simple metric which gives an idea of the distribution of measures in our samples. The metric, given in equation (1), calculates the gap between the median and minimum values of a sample of measure S, a small ratio means that half the points of the sample are close to each other. We give less importance to the second half since we consider that the biggest values are likely to be outliers.

$$\textbf{Stability(S)} = \frac{Median(S)-Min(S)}{Min(S)} \qquad (1)$$

For each event in Table 7.1, Figure 7.1 shows the stability of all the data size points within each codelet of our test suite. Thus, we observe for all events that, small data size points generally are concerned with instability; big data do not suffer from it because of the high number of generated events, except `MEMLOAD_UOPS_RETIRED_L2_HIT` and `MEMLOAD_UOPS_RETIRED_LLC_HIT` for which we recorded some instability. We also notice that the stability within small data size points vary between events, we record an excellent stability for the event `INTR_RETIRED_ANY`, a good stability for cache traffic events, except `L3_LAT_CACHE_MISS` for which we observe a high number of instable points. Still, we noticed two major cases of instability: first in the majority of cases, instable data size points are located between two stable data size points in memory related events, which probably means that this instable point triggered its own abnormal memory behavior, and second the number of events is too small to be considered, thus the notion of stability is not really relevant in this case.


CPU_CLK_UNHALTED_CORE_P
INSTR_RETIRED_ANY
L1D_REPLACEMENT
L2_LINES_IN_ANY
L2_TRANS_LS_WB
L3_LAT_CACHE_MISS
MEMLOAD_UOPS_RETIRED_HIT_LFB
MEMLOAD_UOPS_RETIRED_L1_HIT
stability
data size
Counts

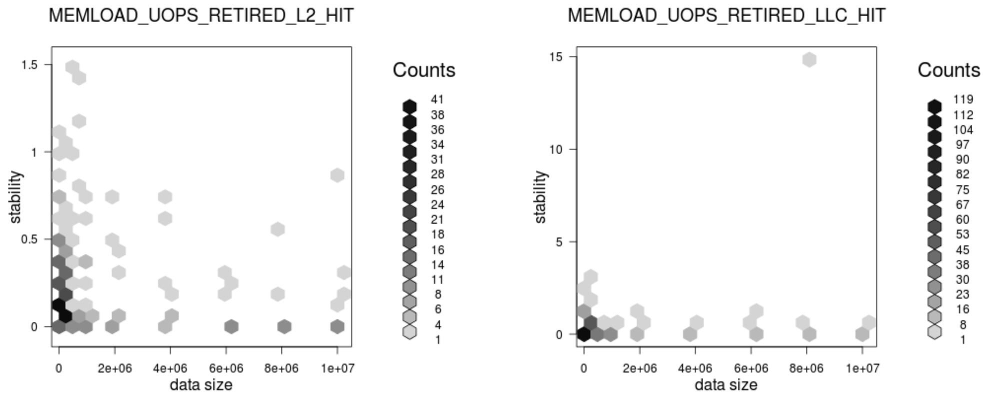


Figure 7.1: For each event, the stability of all data size points for all the NR codelets of the test suite

7.5 Measurement Precision

As mentionned earlier, *Differential Analysis* is a comparative approach, it operates on loops at binary level. Therefore, the precision of measure should be better if the probes are placed closer to the loop.

Figure 7.2 (a) illustrates the source code of *hqr_ 12* codelet. Suppose we target the inner loop (highlighted in light blue) with Decan. The corresponding binary code (highlighted in light blue) in (b) contains three loops, a peel, main and a tail loop. Decan only targets the main loop (highlighted in dark blue), since the peel and tail loops are negligible compared to it. If the probes were to be placed around the loop at source level, they would monitor the events of the three loops, which may result in a loss in precision. We, therefore, use Madras to inject the probes at the entry and exits of the main loop, which ensures the highest possible precision.

Still, the closest to the loop probes are, the more bias is likely to be introduced to the measure. This is possible because probes also generate events. Probe generated events are considered as noise, and need to be negligible for the measure to be accurate. Two parameters are essential in the significance of such noise:

- The ratio probe event on loop events. The more events are generated within the loop, the less probe events are significant.
- The quality of the probe itself: if carefully written, a probe generates less events.

In the two following sections we analyze the impact of each of these two parameters on our measurements.

7.5.1 Small Regions

Following the experimental setup detailed in Section 7.3, we made the data size within each of our codelet vary in order to get loops with a variable number of events. We then observed the data size for each event, starting from the value where the total event count converges in real measures till the total count found in reference measures. We have been able to distinguish two types of events: 1) stable events in which the real measure gets closer to the reference one as more events are being

```
do i=1,n
  imaxarg1 = i-1
  imaxarg2 = 1
  do j= max(imaxarg1,imaxarg2)+1,r
   anorm = anorm + abs(a(j,i))
  end do
end do
```

(a) hqr_12 source code

```
402d79: test   %rcx,%rcx
402d7c: jbe    402d98 <astex_codelet__12_+0x108>
402d7e: movss  -0x4(%r15,%r14,4),%xmm1
402d85: inc    %r14
402d88: andps  0x804b1(%rip),%xmm1        # 483240
402d8f: addss  %xmm1,%xmm0
402d93: cmp    %rcx,%r14
402d96: jb     402d7e <astex_codelet__12_+0xee>
402d98: xorps  %xmm1,%xmm1
402d9b: movss  %xmm0,%xmm1
402d9f: xorps  %xmm0,%xmm0
402da2: movaps 0x80487(%rip),%xmm2        # 483230
402da9: movaps 0x80480(%rip),%xmm3        # 483230
402db0: andps  -0x4(%r15,%rcx,4),%xmm2
402db6: andps  0xc(%r15,%rcx,4),%xmm3
402dbc: add    $0x8,%rcx
402dc0: addps  %xmm2,%xmm1
402dc3: addps  %xmm3,%xmm0
402dc6: cmp    %r8,%rcx
402dc9: jb     402da2 <astex_codelet__12_+0x112>
402dcb: addps  %xmm0,%xmm1
402dce: movaps %xmm1,%xmm0
402dd1: movhlps %xmm1,%xmm0
402dd4: addps  %xmm0,%xmm1
402dd7: movaps %xmm1,%xmm2
402dda: shufps $0xf5,%xmm1,%xmm2
402dde: addss  %xmm2,%xmm1
402de2: movaps %xmm1,%xmm0
402de5: cmp    %rbp,%r8
402de8: jae    402e0c <astex_codelet__12_+0x17c>
402dea: lea    0x4(%rdi,%rbx,4),%rcx
402def: add    %rsi,%rcx
402df2: movss  -0x4(%rcx,%r8,4),%xmm1
402df9: inc    %r8
402dfc: andps  0x8043d(%rip),%xmm1        # 483240
402e03: addss  %xmm1,%xmm0
402e07: cmp    %rbp,%r8
402e0a: jb     402df2 <astex_codelet__12_+0x162>
```

(b) hqr_12 binary code

Figure 7.2: hqr_12 codelet source and binary codes.

generated within the loop, until the two meet at a specific point and do not separate for bigger data sizes. An example of this is shown in Figure 7.3 where real measures of `L1D_REPLCEMENT` event slowly converge to reference values. 2) unstable events which converge to reference values in more than one point. An example of this is shown in Figure 7.4 where real measures values of `MEMLOAD_UOPS_RETIRED_L1_HIT` converge to reference measures when the two reach the size 4k, after what the curves disjoin to meet again in point 20k.

The problem that unstable events raise is that it is hard to define a threshold value for the minimum event count that can ensure an accurate measure. Moreover, we observed that event families: *elapsed cycles*, *instructions retired* and *cache traffic* generally were stable, whereas *cache hits* events generally were unstable.

In order to get more insight on the threshold event counts starting where real and reference measures match, we gathered for each event threshold values of all codelets and studied their distribution. The first problem we encountered was a difference in scale; some codelets generated a lot more events between two data size points than others. Thus, we had to normalize each codelet event count at the total number of executed instructions of the loop. Figure 7.5 shows distributions of the normalized events for each event family. We, thus, noticed that

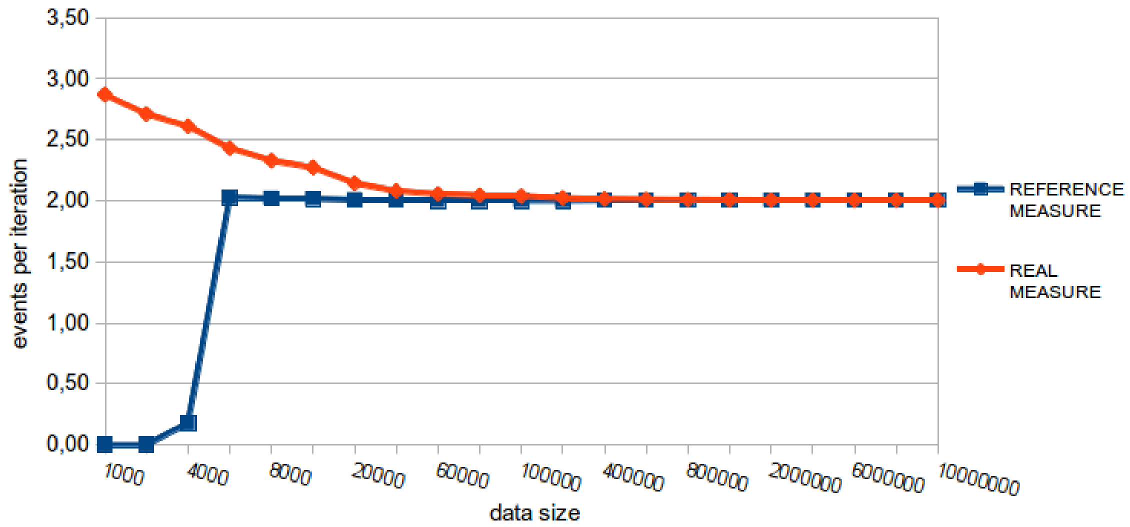


Figure 7.3: Evolution of the number of measured `L1D_REPLACEMENT` events per iteration following data size for `balanc` codelet.

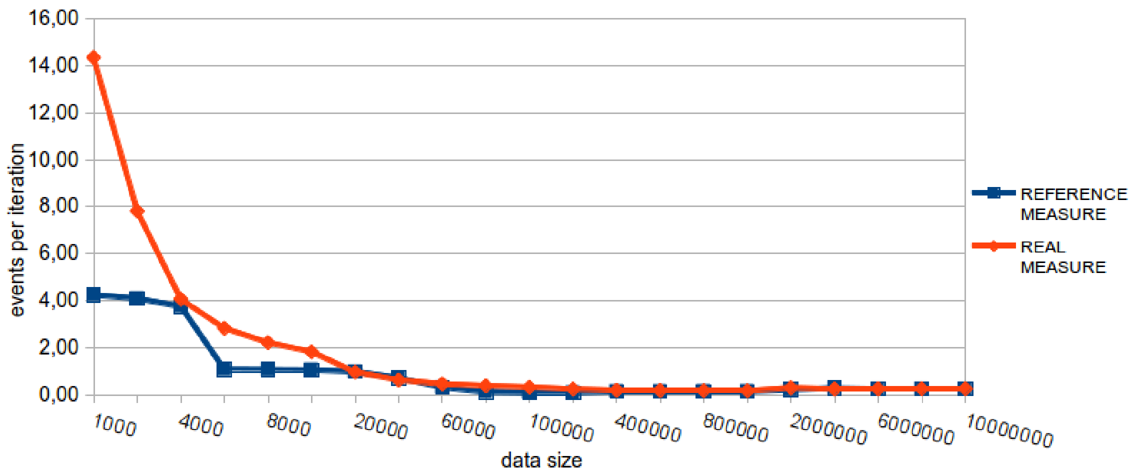


Figure 7.4: Evolution of the number of measured `MEMLOAD_UOPS_RETIRED_L1_HIT` events per iteration following data size for `balanc` codelet.

all cache traffic events except `L2_TRANS_L2_WB` seemed well distributed arround their median, `L1D_REPLACEMENT` and `L2_LINES_IN_ALL` being arround a median of 0.1 events per instruction and `L3_LAT_CACHE_MISS` at a much lower median. `CPU_CLK_UNHALTED_CORE` and `INSTR_RETIRED_ANY` also seemed also well distributed, with a median close to 2 events per instruction for each. Our suspicions about the instability of cache hits events were confirmed, only `MEMLOAD_UOPS_RETIRED_L1_HIT` and `MEMLOAD_UOPS_RETIRED_HIT_LFB` seemed relatively well distributed. We only collected a few points for `MEMLOAD_UOPS_RETIRED_L2_HIT`, and could not collect any threshold points for `MEMLOAD_UOPS_RETIRED_LLC_HIT` due to its high instability.

7.5.2 Probes Accuracy

The lightness of the probe plays a significant role in the accuracy of a measure. Using hardware counters requires to directly access to special hardware registers (on Intel x86 they are called Model Specific Registers MSRs). These usually are divided into

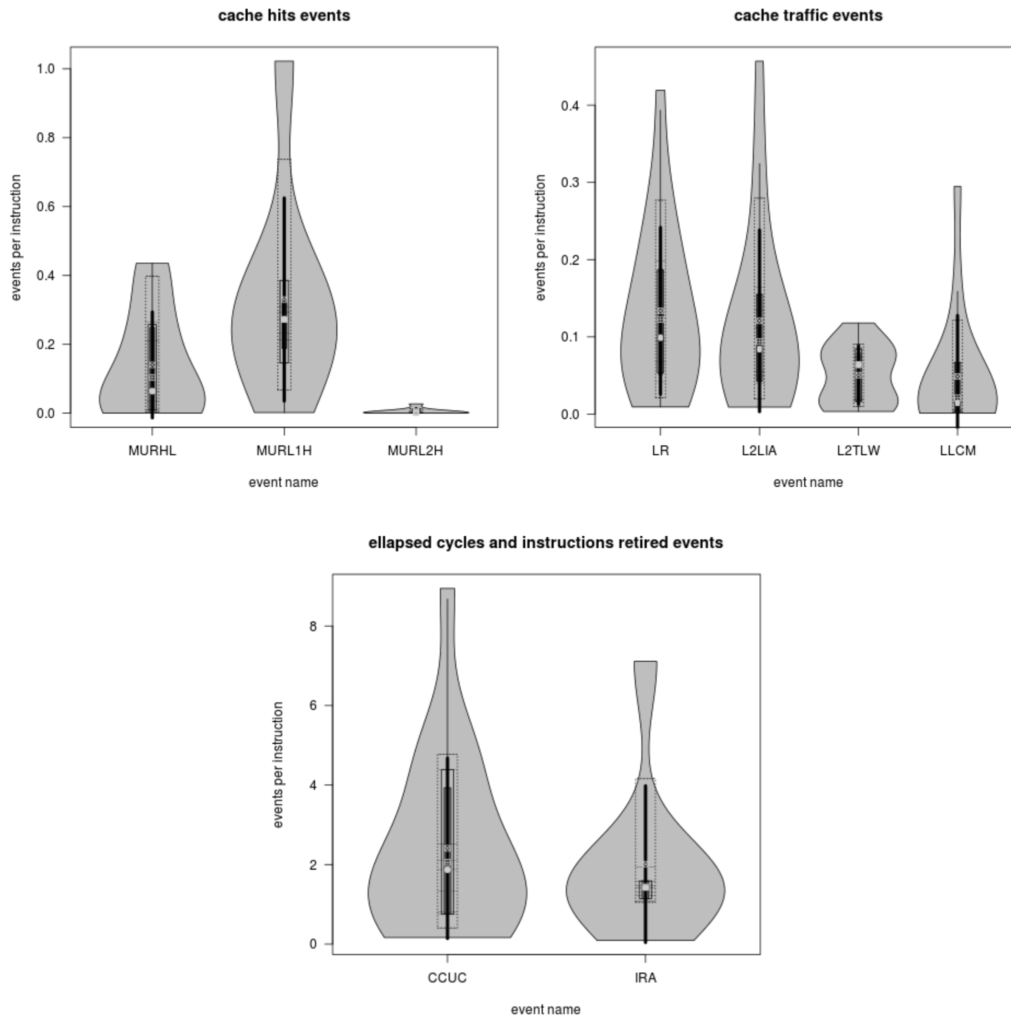


Figure 7.5: For each event, the distribution of the smallest event count starting from where the real and reference measures match

configuration registers and *counting registers*. Because the number of counters is too big, the right elements have to be chosen among the configuration counters before being read through the counting registers. The configuration usually needs ring 0 privileges (kernel), whereas counter reading is less critical but still, in the majority of cases, need kernel assistance. Thus, the Linux kernel provides the perf_event interface [15] that translates users requests into the proper low-level CPU calls. In general, access is done through high level libraries which may add more overhead such as *PAPI* [75]. In [95] it has been shown that the perf_event, as well as older interfaces, used to access hardware counters on Linux, such as perfmon2 [39] and perfctr [74], and introduced an overhead because of the system context switches and measurement, in addition to unindentified overheads that differs between Linux kernel versions. They also observed that the current interface perf_event introduced more overhead than perfmon2 and perfctr because more work is done inside the kernel. They concluded that if CPU vendors provided faster access to the counters, this might help reduce the overhead, and they observed efforts on AMD with the `spflt` instruction and on the Intel MIC chip [11].

On the Intel x86 processors on which we conducted our measurement, only

the *time stamp counter* (TSC) is made accessible in ring 3 (userspace) through the assembly instruction `rdtsc`. The instruction enables to directly read the value of the counter. It allowed us to build a light probe based on a method to access the values of the counter described in [81].

In order to study the gain in precision obtained through the use of the light method, we compared the results of the TSC (obtained with a light `rdtsc` based probe) against the results obtained with the counter CPU_CLK_UNHALTED_CORE (obtained with a heavy probe). The comparison is made possible because the counters have the same accuracy in cycle counting, the only difference is that texttttCPU_CLK_UNHALTED_CORE is sensitive to the core frequency change, whereas, TSC is converted to the highest frequency possible for the processor. Therefore, by setting the core frequency at the maximum, we ensure the same scale for the two counters.

We also used the same metric as in the previous section. For the two measurement methods, we recorded, for each codelet, the minimum counter value that matches the reference measure. The results presented in Figure 7.6 show that, for the majority of codelets, the use of `rdtsc` offers a much better precision. We notice big disparities for some codelets (e.g. **Lop_13** and **Relax_26**). Only **elmhes_11** records better precision for the heavy method, but we observed that the codelet has unstable and varying performance among data size points.

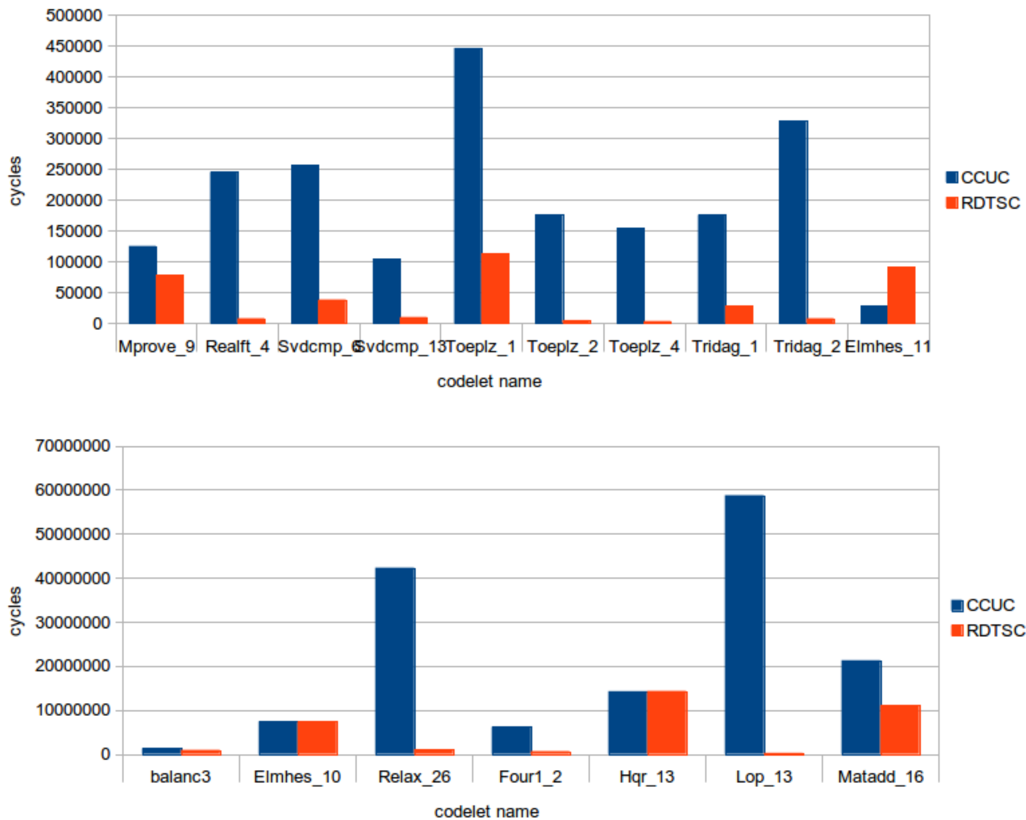


Figure 7.6: Minimum cycles count for which the real measure matches the reference measure for heavy measurement method that accesses the CPU_CLK_UNHALTED_CORE counter (**CCUC**) and light measurement method using **rdtsc**. Results are shown for 17 NR codelets.

7.6 Probe Intrusiveness

Probe intrusiveness is related to probe quality. Being a piece of code themselves, probes are likely to generate events as well. Some of these events are counted within the measure and cannot be separated from the events generated by the monitored area. Furthermore, they can fight on hardware resources with the other set of events, thus, changing the original program behavior. Another aspect of intrusiveness is the time overhead that is added by the measurement process to the original execution time of the program. The majority of works on intrusiveness tackle the second aspect and neglect the first one; in [61], transformation methods are proposed to reduce the number of instrumentation points as well as the weight of the probes. In this section, we try to gain more insight on the introduced overhead in order to quantify and reduce it.

Equation (1) provides our view on how a measure t_val on an event e can be decomposed. We use the following terms:

- $t_val(e)$ is the observed number of occurrences of event e
- $l_val(e)$ is the number of occurrences of event e generated within the loop
- $p_val(e)$ is the number of occurrences of event e generated within the probe
- $interac_val(e)$ is the change on the number of occurrences of event e induced by the fight on hardware resources between probe events and loop events

$$t_val(e) = l_val(e) + p_val(e) \pm interac_val(e) \qquad (1)$$

$l_val(e)$ represents the true number of occurrences that we seek for an event e, $p_val(e)$ and $interac_val$ both represent the noise introduced by the probes. We notice that the interaction between the two types of events may lead to either more events, in which case the term $interac_val$ is added, or less events, in which case the term is subtracted.

Within this section, we try to reduce the overhead introduced by a probe $p_val(e)$ in a measure $t_val(e)$, our goal being to get a close approximation of $l_val(e)$. For this, we use our tool DECAN to determine $p_val(e)$ as described in Section 7.6.2.

7.6.1 Relationship Between Event Type and Probe Intrusiveness

The interaction between loop events and probe events greatly depends on the category of the event. We recognize four main cases: no interaction, positive, negative and random interaction:

- **No interaction**: no interaction between probe events and loop events ($interac_val = 0$). Meaning that the events generated within the probe are totally independent from those generated within the loop. Examples of events that fall in this category include all counters that record the passage of micro-instructions through execution ports: numbers of multiplication micro-ops and division micro-ops, number of instructions retired, *etc.* We called event associated with random interaction as *throughput* based events.

- **Positive/negative interaction**: either more events are generated through the interaction of probe events with loop events (*interac_val* is added) or less events (*interac_val* is subtracted), depending on the nature of the event. On the other hand, cache traffic events (cache line replacement) are an example of positive interaction, because of the fight on cache slots between probe events and loop events, more cache lines are likely to be replaced. Cache hit events, on the other hand, are an example of negative interaction where less cache hits are likely to be generated. We called event associated with random interaction as *capacity* based events.

- **Random interaction**: the interaction between probe events and loop events is random and differs between different executions of the same code; The outcome of interaction can generate either more or less events or no extra events at all. An example of events that fall in such category include the branch history mechanism and the events associated with it, where the component that generates the event depends on its own history. We called event associated with random interaction as *hysteresis* based events.

Another important event kind that we tend to classify in the first category are Elapsed cycles events (used to determine execution time). At a first glance, no interaction seems possible between probe and loop events, however, the event indirectly reflects the outcome of other events that admit interaction. For example, if more cache traffic is generated through interaction, it is the noticeable in the form of additional cycles within the execution time.

7.6.2 Reducing Probe Intrusiveness With Decan

The observed value of an event e through measurement $t_val(e)$ is the aggregation of the occurrences generated by the loop $l_val(e)$ and those generate by the probe $p_val(e)$ with the addition or subtraction of the outcome of interaction between the two $inter_val(e)$ if present. It is impossible to know the true value of $l_val(e)$ since it requires the absence of probes, however, it is possible to measure $p_val(e)$ if we manage to obtain a version of the code in which only the probes are present. By fixing $p_val(e)$, it would be possible to reduce equation (1) into equation (2). This simplification offers a better approximation to $l_val(e)$ where $inter_val(e)$ remains the only unknown term.

$$l_val \pm inter_val = t_val - p_val \qquad (2)$$

Decan fortunately offers this possibility. With Decan we are able to create binary variants in which the loop to be measured is suppressed but probes are kept. The `CTRL` variant introduced in Section 4.4.3 can be used for such a task. The variant creates a minimal version of the loop in which only instructions involved in the looping process are kept. It allows, thus, to measure $p_val(e)$ for the majority of events of interest introduced in Section 7.2 especially memory counters.

Regarding the efficacy of the corrections enabled by the technique, we expect different results for each category of interaction. Thus, in the case of no interaction, we expect a precise approximation of l_val as showed in Table 7.2. In a positive interaction case, on the other hand, we expect the subtraction of p_val to systematically reduce the overhead, since extra events are generated by the interaction. In

the case of negative interaction on the other hand, we are more skeptical regarding overhead reduction, since the interaction itself acts as a reduction mechanism, the subtraction of p_val can cause the correction to be worse than the measured value t_val.

Interaction type	Expected approximation of l_val
No interaction	$l_val = t_val - p_val$
Positive interaction	$l_val + inter_val = t_val - p_val$
Negative interaction	$l_val - inter_val = t_val - p_val$

Table 7.2: Expected values for the approximation of l_val following the type of interaction between probe and loop events.

7.6.3 Experimental Results

In order to evaluate the coverage of the overhead reduction mechanism we introduced in the previous section, we applied the technique on all the data size points of our codelets test suite, and for all the events of interest. We then, filtered the points which are already close to the reference measures, and enumerated for the remaining points all the cases where the reduction actually improved the measurement error (reduced the relative error to the reference values). Figure 7.7 shows, for each event, the average error reduction or error increase as well as the cases where the correction did not have any effect. On average we were able to reduce the overhead on 67% of the data size points. We noticed differences in scores between events families. Cache traffic events register the best reduction score with an average of 76%, cache hits events register an average of 57%, whereas `INSTR_RETIRED_ANY` and `CPU_CLK_UNHALTED_CORE` respectively register 67% and 73%. Moreover, in the majority of cases, failuring to reduce the overhead in fact introduces an increase; we indeed recorded an average increase on 28% of the points.

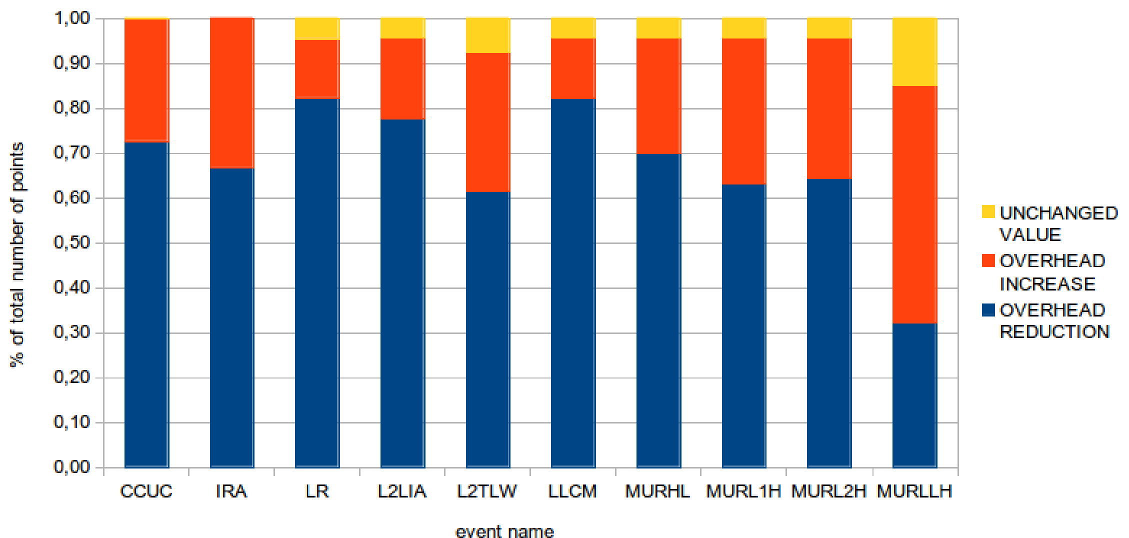


Figure 7.7: Average error reduction and error increase after the subtraction of events generated by the probes

We also evaluated the efficacy of the technique by counting the number of successful corrections, i.e. when the reduction led to a match with the reference value. The results are given in Figure 7.8. On the total number of overhead reductions an average of 33% of cases had their overhead successfully subtracted. Furthermore, we observed more successful cases for elapsed cycles, instructions retired and memory traffic events than for cache hits events. This matches our statements on the efficacy of the technique regarding the later category.

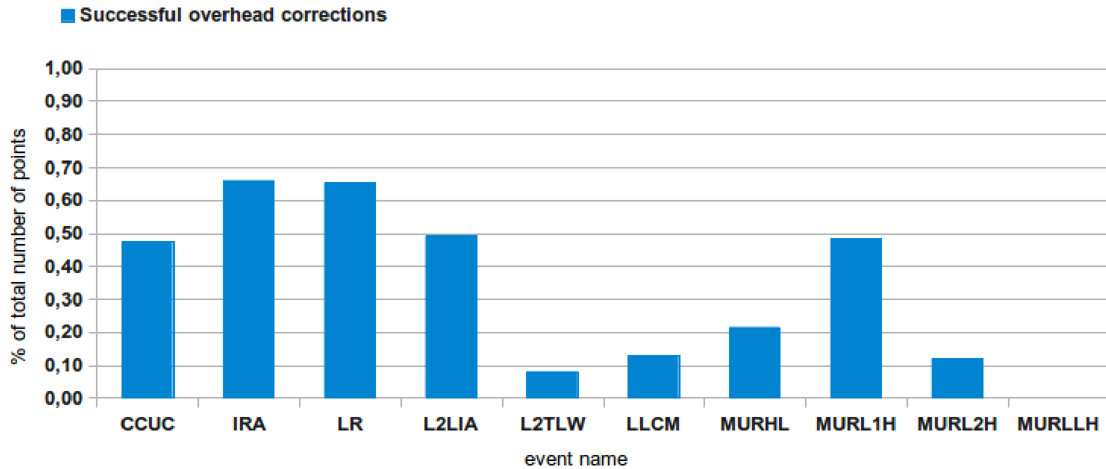


Figure 7.8: Ratio of successful overhead corrections (relative error <0,05) among error reductions (error increase cases are not taken into account)

Finally, the major gap that could limit the application of our technique, despite the high coverage of the reduction, is the detection of the appropriate cases to be examined. It is because we knew that some points already matched the reference values, that we filtered them out of the study, which is not possible in real applications. We think that the small regions issue that we investigated in Section 7.5.1 can be a good start for an intelligent filtering of the points that already match reference. Moreover, we do not exactly know why the overhead reduction attempt led to an increase in the overhead for a certain number of points. We suspect that it is related to the inner properties of the codelets, which means that a targeted characterization might be necessary.

7.7 Summary

Within this chapter we investigated measurement related issues in an exhaustive measurements through code instrumentation. The first issue common to all kinds of measurements is *measurement stability*. Previous works on the matter identified several factors which contribute to the instability of measurements, a number of which cannot be nullified, thus we adopted some good practices in order to minimize their effect. Also, despite finding some disparities in the stabilities of the events of interest, we found that our samples generally had a good stability. Second, we investigated *measurement precision* related issues. The probes being a source of events as well, we studied the ratio of probe generated to loop generated events which do not affect the accuracy of measure. Third, we investigated the issue of *probe intrusiveness*, which is similar to precision issues except that, from this perspective, we explored the possibility to quantifying and subtracting the added overhead. In order to achieve this, we used DECAN to isolate and subtract part of the introduced overhead. Experimental results showed that we were able to reduce the overhead in 67% of cases, among which 33% resulted in a total elimination.

Our investigation on the possibility to use hardware counters in Differential Analysis seems promising. We were able to obtain hints on the minimum amount of events a loop should have in order to obtain a good accuracy, and were able to test an effective method to reduce the overhead of measure. However, the work remains unfinished. On real cases, we still do not know when we should apply the overhead reduction technique. We find that a joint study between these two aspects may lead to better success rates, moreover, we believe that, codelet characterization, an aspect that we neglected within our study also impact the matter and should be included in future studies.

Chapter 8
Conclusions

We proposed extensions to and contributed to the design of a novel technique for application performance analysis called *Decremental Analysis*. The technique adds a new dimension to the observation process involved in a measurement based analysis by introducing small modifications to the code and observing their effects. The goal of the analysis is to detect fine grain bottlenecks. It targeted hot inner-most loops, by deleting instructions in a search for the most expensive ones. The entire process of code modification works at the binary level which has the double advantage of avoiding any collateral process that may trigger other changes in the code, and of being agnostic to the multiple programming paradigms.

Our main contribution consists of the extension of the concept of *Decremental Analysis*, instead of only tying the cost assessment to the suppression of instruction, we extended it to wider notion of Differential Analysis. With Differential analysis, events are not necessarily idealized; they can be modified in order to have their impact reduced (e.g. `DL1` variant).

This was made possible as we established a better transformation process of the instructions in which all the loop variants and analysis methods are motivated by practical needs for both code characterization and bottleneck detection. This technique proved to be effective in getting proper insight on loop behavior, and quick and precise assessment of a number of fine grain performance bottlenecks on industrial codes.

We believe that a single analysis approach cannot cover the entire spectrum of performance evaluation and bottleneck detection. The diversity of the available tools makes it possible to use a time consuming analysis for a performance issue that could be diagnosed with a lighter one. Consequently, we integrated Decan with another set of tools into an analysis methodology called Pamda. The methodology tries to quickly find the location and nature of the most important pathologies, and suggests the right tool to diagnose them. The characterization capabilities of Differential Analysis make it as a central part of this methodology. The process proved to be efficient in reducing analysis time by quickly determining the important performance issues some industrial codes faced.

Differential Analysis relies on a technically challenging tool (Decan). During our work, we focused on three technical aspects:1) the alteration of the inner control flow of the modified loops and the global program control flow, which we managed to either preserve or at least control. We thus have been able to use Decan on loops in their real context (in-vivo) instead of extracted kernels (in-vitro), 2) the feasibility, i.e. the possibility to find the right code transformations to highlight a particular pathology. On this, we show that when facing some limitations, we can either be able to establish some workarounds or, on the opposite, introduce some unwanted noise in the variants that we have to try and reduce, and 3) the extension of the tool to handle common `FOR` loops based on parallel threads (`parallel for` of OpenMP) as well as process based parallelism (MPI context).

Measurement correctness is important for tools that rely on measurement techniques such as Decan. The tool currently relies on elapsed cycles as a mean of comparison between variants. The advantage is that there is a particular hardware mechanism which enables precise and stable measures of time. Still, we intend to make use of other events (e.g. traffic events) as a means of comparison with elapsed cycles. However, a prior study of their accuracy is needed. We conducted this study on three aspects: measurement stability, precision and probe instrusiveness. We showed that our measurement mechanism generally provided good stability and precision, with some disparities between event families, whereas, on probe intrusiveness, we used Decan to remove the overhead introduced by the probes, and observed a reduction of intrusiveness in nearly 70% of the cases. Finally, we found the study was an important enough step to be pushed a little further, the objective being to introduce new counters within Differential Analysis.

8.1 Perspectives

In this section, we describe future work. We discuss the main development lines we think are suitable for the development of Decan as a tool for a better analysis. We also highlight what we consider as the most interesting research directions.

8.1.1 Tool Development

The Decan tool can still benefit from several improvements: tool usability, more features for improved analysis, better coverage, *etc.* The most significant are:

8.1.1.1 Analysis Time Reduction

The current version of Decan can be quite heavy in terms of analysis. Each variant needs an execution of the program. Therefore, a minimal analysis where only two variants are run, *e.g.* `REFERENCE` and `LS`, costs *(2 x T)* (T being the execution time of the program). The cost also has to be multiplied by the number of multiple repetitions of execution in order to have a stable sample. We were able to reduce this time with the *instance mode*, in which the execution is stopped just after exiting the selected loop call by Decan. The analysis cost then depends on the location of the selected loop call.

Several improvements can reduce analysis time. A new version Decan which takes our tool as a starting instance is being developed and should support these improvements:

- The use of *recovery mode* to execute several loops within a single run. The present version creates variants for only one loop at a time. The handling of several loop variants within a single execution of the program enables to divide the analysis cost by the number of loops to analyze.

- *recovery mode* also allows to run several variants within a single program execution. When combined with *instance mode* (without the exit), meaning that only one loop call is processed, it would be possible to execute several Decan created versions of the loop and, at the end, execute the original version in order to ensure the semantic correctness.

- Another interesting improvement we can think of, is the replacement of program repetitions by a lighter process. Since our measurements are done at the loop call level, we may think of building our sample with several loop calls´results. Obviously, this solution works for loops which do not behave very differently when facing various loops calls.

8.1.1.2 Handling Multi-Path Loops

The version of DECAN we developed for our research, only handles regular loops (loops with a single execution path), which provided a good coverage, in term of program execution time, for the majority of applications we had to analyze. Another reason for this choice is that innermost loops are usually regular. However, the handling of irregular loops (loop with multiple execution paths) will have the benefit of providing a better coverage as well as the possibility to handle non innermost loops. The main idea behind irregular loops handling is the preservation of the proper control flow that is located inside the loop. Below are a few possible solutions:

- Perform a static analysis of the code in order to locate the instructions involved in the internal control flow. The analysis can be very lightweight but it has all the drawbacks of a static analysis, the biggest of which being the limited success in detecting all the involved instructions (as runtime information are needed).

- Perform a first execution of the unmodified loop in which the branches outcome (taken/not taken) is recorded in a trace. The trace is then used to control branch instructions in the transformed loop. The difficulty lies in the manner with which the trace is fed to the transformed loop at runtime. New instructions would probably need to be injected inside the loop in order to support the trace.

8.1.1.3 Extensions to Other platforms

For the moment, the primary and only platform supporting DECAN is Intel x86. The concepts on which the tool relies (instruction subsets, transformations and variants) provide a sufficient level of abstraction sufficient to enable an easy integration of other platforms. However, the differences in micro-architectural details between platforms suggests the applications of a rigorous validation process. We particularly intend to provide a support for the ARM platforms in the near future.

8.1.2 Research Topics

The exploration of the potential of Differential analysis is still at its debute. In our work, we focused mainly on its use in application performance analysis, an increased mastering of the technical aspects would enable to continue the research in this direction, but also to use the approach in other areas. Following are some of the interesting research topics to address:

8.1.2.1 Energy Related Issues

Energy consumption has become a big concern in current years, with supercomputers being equipped with a growing number of processors, efforts are twofold: on the

hardware side, efforts are put on the need for highly energy efficient processors, on the software side, optimized softwares for a moderate consumption need to be elaborated.

As a result, we may start to thinking of energy as a performance pathology that needs to be investigated, Therefore, *Differential Analysis* can be useful as illustrated in the following scenarios:

- Characterisation of the different streams (LS, FP), instruction subsets (L, S, GR) and instructions in term of energy consumption through the variants we have introduced in Chapter 4.
- In a context where there is a trade-off between performance and energy consumption, the `DL1` variant can be useful to find out how much energy can be saved in a memory bound loop. The result helps to decide whether the loop should be optimized or not.

8.1.2.2 Hardware-Software codesign

Throughout our research, we have been able, to test DECAN capabilities to explore the behavior of micro-architectural components (see section 4.6.3). We also worked in collaboration with Intel to integrate DECAN in their codesign process. Codesign at core level consists in determining the right capabilities hardware components should have (e.g. size of cache, number of functional units, *etc*) following the needs of the software. Generally, the search space is huge and is either explored through simulation or through modeling. The first is too heavy and the second lacks accuracy. DECAN is involved in a tool the goal of which is to obtain the near optimal parameters with a very lightweight simulation compared to classical techniques. The idea of the approach can be found in [60], and a practical example of it which includes DECAN as one of the simulation components is illustrated in [79].

8.1.2.3 Improve the analysis methodology

The provided methodology in chapter 5 is far from being finished. Our constant challenge is to keep improving it as well as working towards full automation. We also aim to enlarge it for other kind of paradigms through the integration of analyses provided by complementary tools such as Scalasca, Vampir and TAU. Additionally, refining optimization investigations is crucial in order to make it more user-friendly.

Bibliography

[1] 3DNow! technology. http://support.amd.com/TechDocs/21928.pdf. 8

[2] Acumem. http://www.roguewave.com/. 21, 24

[3] Amd64 architecture programmer s manual volume 2:system programming. http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf. 13

[4] Amplifier XE. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-documentation/. 23

[5] ARM NEON support in the arm compiler. http://www.arm.com/files/pdf/neon_support_in_the_arm_compiler.pdf. 8

[6] GCC, the gnu compiler collection. https://gcc.gnu.org. 1

[7] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. ix, 14, 15

[8] Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. 22

[9] Intel architecture code analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/. 21

[10] Intel c and c++ compilers. https://software.intel.com/en-us/c-compilers. 1, 21

[11] Intel xeon phi coprocessor (codename: Knights corner) performance monitoring units. . 97

[12] MAQAO project. http://www.maqao.org. 24, 57

[13] Memory part 2: Cpu caches. https://lwn.net/Articles/252125/. ix, 10

[14] Simulators and such... http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/Simulators.pdf. 19

[15] The Unofficial Linux Perf Events Web-Page. http://web.eece.maine.edu/~vweaver/projects/perf_events/. 74, 97

[16] Top 500 the list. http://www.top500.org/. 1

[17] Valgrind: instrumentation framework for building dynamic analysis tools. http://www.valgrind.org. 21, 23

[18] Acumem. Acumem threadspotter. http://www.roguewave.com/products/threadspotter.aspx. 53

[19] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, Apr. 2010. 67

[20] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *IISWC*, pages 225–236, 2006. 58

[21] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar. *Multi-Core Cache Hierarchies*. Morgan and Claypool Publishers, 2011. v, 12

[22] D. Barthou, A. C. Rubial, W. Jalby, S. Koliai, and C. Valensi. Performance tuning of x86 openmp codes with MAQAO. In *Parallel Tools Workshop*, Dresden, Germany, sep 2009. Springer-Verlag. 57

[23] E. Baysal, D. Kosloff, and J. Sherwood. Rerverse time migration:geophysics, 1983. 47, 65

[24] J. C. Beyler, N. Triquenaux, V. Palomares, F. Chabane, T. Fighiera, J.-P. Halimi, and W. Jalby. Microtools: Automating program generation and performance measurement. In *ICPPW, 2012*, pages 424–433. IEEE, 2012. 53

[25] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003. 71

[26] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 47, pages 133–144. ACM, 2012. 71

[27] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997. 19

[28] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *2010 ACM/IEEE*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. 23

[29] M. Burtscher, B.-D. Kim, J. R. Diamond, J. D. McCalpin, L. Koesterke, and J. C. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *SC*, pages 1–11. IEEE, 2010. 53, 67

[30] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 1, 78

[31] A. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby. Mil: A language to build program analysis tools through static binary instrumentation. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 206–215, Dec 2013. 24, 73

[32] A. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. Cqa: A code quality analyzer tool at binary level. In *High Performance Computing (HiPC), 2014 20th International Conference on*, Dec 2014. 53, 73

[33] A. S. Charif-Rubial. *On code performance analysis and optimisation for multicore architectures.* PhD thesis, Oct. 2012. 54

[34] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. S. Shende, A. D. Malony, and i.-p. William Jalby. MIL: a language to build program analysis tools through static binary instrumentation. In *HiPC'13*, Hyderabad, India, Dec. 2013. 59

[35] A. S. Charif-Rubial, E. Oseret, G. Lartigue, and W. Jalby. Cqa: A code quality analyzer tool at binary level. In *In 21th Annual International Conference on High Performance Computing, HiPC'14*, Goa, India, December 2014. 21, 24

[36] C. Curtsinger and E. D. Berger. Stabilizer: statistically sound performance evaluation. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 219–228. ACM, 2013. 91

[37] A. C. de Melo. Performance counters on linux: the new tools. 2009. 22

[38] Q. V. Dinh, A. Naim, and G. Petit. rapport final de synthèse sur l'optimisation des logiciels de simulation numérique de l'aéronautique. *Technical report, Dassault Aviation*, pages xii, 51, 53, 54, 70, 71, 83, 2007. 49

[39] S. Eranian. Perfmon2: a flexible performance monitoring interface for linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288. Citeseer, 2006. 97

[40] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. *SIGARCH Comput. Archit. News*, 30(2):47–58, May 2002. 50

[41] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. *SIGARCH Comput. Archit. News*, 29(2):74–85, May 2001. 18, 50

[42] B. A. Fields, R. Bodík, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 228–, Washington, DC, USA, 2003. IEEE Computer Society. 50

[43] B. A. Fields, Rastislav, M. D. Hill, and C. J. Newburn. Interaction cost: For when event counts just don't add up. *IEEE Micro*, 24(6):57–61, Nov. 2004. 24

[44] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahàm, D. Becker, B. Mohr, and F. JÃ$\frac{1}{4}$lich. The scalasca performance toolset architecture. In *STHEC*, 2008. 23, 53, 67

[45] J. Goodman and H. Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2004). 2004. 13

[46] Gprof. The gnu profiler. http://sourceware.org/binutils/docs-2.18/gprof/index.html, 2013. 67

[47] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced features of the message-passing interface.* MIT press, 1999. 1

[48] M. Hagog and A. Zaks. Swing modulo scheduling for gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 55–64, 2004. 84

[49] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ix, 5, 6, 13

[50] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao–an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–10. IEEE Computer Society, 2011. 20, 21

[51] Intel. Intel vtune amplifier xe. www.intel.com/software/products/vtune, 2013. 53, 67

[52] R. Jain. *the art of computer systems performance analysis: techniques for experimental design, measurment, simulation and modeling.* 1992. 92

[53] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999. 78

[54] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005. 11

[55] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ACM SIGARCH Computer Architecture News*, volume 32, page 338. IEEE Computer Society, 2004. 18

[56] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 287–296. ACM, 2010. 18

[57] S. Koliai. *Static and Dynamic Approach for Performance Evaluation of Scientific Codes.* PhD thesis, Versailles, France, 2011. 3, 27, 44, 75

[58] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby. Quantifying performance bottleneck cost through differential analysis. In *27th ICS*, pages 263–272, New York, NY, USA, 2013. ACM. 24, 44, 54, 63

[59] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby. A balanced approach to application performance tuning. In *LCPC*, pages 111–125, 2009. 57

[60] D. Kuck. Computational capacity-based codesign of computer systems. In M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, editors, *High-Performance Scientific Computing*, pages 45–73. Springer London, 2012. 108

[61] N. Kumar, B. R. Childers, and M. L. Soffa. Low overhead program monitoring and profiling. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 28–34. ACM, 2005. 99

[62] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010. 71

[63] J. Levon and P. Elie. Oprofile: A system profiler for linux. http://oprofile.sourceforge.net, 2013. 67

[64] J. Liu, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K, and P. Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 2004. 58

[65] X. Liu, J. Mellor-Crummey, and M. Fagan. A new approach for performance analysis of openmp programs. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 69–80. ACM, 2013. 21

[66] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005. 71

[67] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. 19

[68] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 2–13. ACM, 2004. 18

[69] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, 1992. 67

[70] N. Mc Guire, P. Okech, and G. Schiesser. Analysis of inherent randomness of the linux kernel. In *Proc. 11th Real-Time Linux Workshop*. Citeseer. 92

[71] E. M. McCreight. The dragon computer system. In *Microarchitecture of VLSI Computers*, pages 83–101. Springer, 1985. 13

[72] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *International Journal of Parallel Programming*, 29(1):35–58, 2001. 18

[73] T. Moseley, N. Vachharajani, and W. Jalby. Hardware performance monitoring for the rest of us: a position and survey. In *Proceedings of the 8th IFIP international conference on Network and parallel computing*, NPC'11, pages 293–312, Berlin, Heidelberg, 2011. Springer-Verlag. 21

[74] M.Pettersson. The perfctr interface. `http://user.it.uu.se/~mikpe/linux/perfctr/2.6/`. 97

[75] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999. 97

[76] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009. 91, 92

[77] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996. 53, 67

[78] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 52–62. ACM, 1994. 18

[79] J. Noudohouenou, V. Palomares, W. Jalby, D. C. Wong, D. J. Kuck, and J. C. Beyler. Simsys: A performance simulation framework. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM. 45, 108

[80] I. Pantazi-Mytarelli. The history and use of pipelining computer architecture: Mips pipelining implementation. In *Systems, Applications and Technology Conference (LISAT), 2013 IEEE Long Island*, pages 1–7, May 2013. 6

[81] G. Paoloni. How to benchmark code execution times on intel IA-32 and IA-64 instruction set architectures. Technical report. 98

[82] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1988. 28, 91

[83] F. Real, M. Trumm, V. Vallet, B. Schimmelpfennig, M. Masella, and J.-P. Flament. Quantum Chemical and Molecular Dynamics Study of the Coordination of Th(IV) in Aqueous Solvent. *J. Phys. Chem. B*, 114(48):15913–15924, 2010. 54

[84] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007. 1

[85] S. Shende, A. Malony, S. Moore, P. Mucci, and J. Dongarra. Integrated tool capabilities for performance instrumentation and measurement. 2007. 23

[86] S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006. 23

[87] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006. 53, 67

[88] O. Sopeju, M. Burtscher, A. Rane, and J. Browne. Autoscope: Automatic suggestions for code optimizations using perfexpert. In *2011 ICPDPTA*, pages 19–25, July 2011. 67

[89] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers, 1st edition, 2011. v, 13

[90] C. Staelin and H. packard Laboratories. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996. 58

[91] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, Jan. 1967. 8

[92] A. Vajda. *Programming Many-Core Chips.* Springer Publishing Company, Incorporated, 1st edition, 2011. v, 11

[93] C. Valensi. *A generic approach to the definition of low-level components for multi-architecture binary analysis*. PhD thesis, 2014. ThÃ¨se de doctorat dirigÃ©e par Jalby, William Informatique Versailles-St Quentin en Yvelines 2014. 71

[94] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 209–220. IEEE Computer Society, 2007. 71

[95] V. M. Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, page 80, 2013. 97

[96] V. M. Weaver and S. A. McKee. Can hardware performance counters be trusted? *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150, 2008. 92

[97] V. M. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 215–224. IEEE, 2013. 91

[98] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 170–183, Berlin, Heidelberg, 2007. Springer-Verlag. 72

[99] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: statistical sampling of computer system simulation. *IEEE MICRO Special Issue on Computer Architecture Simulation and Modeling*, 26(PARSA-ARTICLE-2007-001):19–31, 2006. 19

[100] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. 9

[101] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell. Adp: automated diagnosis of performance pathologies using hardware events. In P. G. Harrison, M. F. Arlitt, and G. Casale, editors, *SIGMETRICS*, pages 283–294. ACM, 2012. 67

[102] W. Yoo, K. Larson, S. Kim, W. Ahn, R. H. Campbell, and L. Baugh. Automated fingerprinting of performance pathologies using performance monitoring units (pmus). In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar '11)*, Berkeley, CA, 05/2011 2011. USENIX, USENIX. 67

[103] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 23–32. IEEE, 2009. 92

List of Publications

1. Souad Koliaï, Zakaria Bendifallah, Mathieu Tribalat, Cédric Valensi, Jean-Thomas Acquaviva, and William Jalby. 2013. Quantifying performance bottleneck cost through differential analysis. In Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS '13). ACM, New York, NY, USA, 263-272. DOI=10.1145/2464996.2465440

2. Bendifallah, Zakaria, William Jalby, José Noudohouenou, Emmanuel Oseret, Vincent Palomares, and Andres Charif Rubial. "PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis." In Tools for High Performance Computing 2013, pp. 107-127. Springer International Publishing, 2014.

3. David C. Wong, David J. Kuck, Vincent Palomares, Zakaria Bendifallah, Mathieu Tribalat, Emmanuel Oseret, William Jalby. "VP3: A Vectorization Potential Performance Prototype" In 2nd Workshop on Programming Models for Vector Processing (WPMVP) 2015, San Francisco Bay Area, USA - United States