UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

Université de Versailles Saint-Quentin **École Doctorale "SoFt"**

Analyse et optimisation de performances sur architectures multicoeurs

On code performance analysis and optimisation for multicore architectures

THÈSE

présentée et soutenue publiquement le 22 Octobre 2012

pour l'obtention du

Doctorat de l'université de Versailles Saint-Quentin

(spécialité informatique)

par

Andrés Salim CHARIF RUBIAL

Composition du jury

Président :	Raymond Namyst	- Professeur, Université de Bordeaux 1
Rapporteurs :	Phillipe CLAUSS	- Professeur, Université de Strasbourg
	David LEVINTHAL	- Docteur, Google, Mountain View, Etat-Unis
Examinateurs :	Allen Malony	- Université de l'Oregon, Eugene, Etats-Unis
Directeur de thèse :	William Jalby	- Professeur, Université de Versailles
Co-Directeur de thèse :	Denis Barthou	- Professeur, Université de Bordeaux 1

LRC IT@CA

Exascale computing research

Remerciements

Je souhaite remercier en premier lieu ma mère, à qui je dédie cette thèse, pour avoir toujours été là quand il le fallait.

Je remercie chaleureusement mes encadrants William JALBY et Denis BARTHOU, avec qui j'ai eu grand plaisir à travailler.

Je remercie également Phillipe Clauss et Alain Ketterlin pour leur disponibilité. Je leur en veux un peu d'avoir sorti NLR pendant que je travaillais sur la même problèmatique !!! A part cela, merci de m'avoir fait découvrir Strasbourg !

Un salut à tous ceux qui m'ont aidé directement ou indirectement au Labo PRiSM et par la suite au Lab Exascale. Un salut particulier à Cédric Valensi à qui j'ai souvent donner du fil à retordre avec mes besoins d'instrumentation binaire.

Je n'oublie pas l'équipe INRIA RUNTIME de Bordeaux où j'ai passé un an et demi. J'ai pu profiter de leur machines multicoeurs qui m'ont permis de travailler sur plusieurs des aspects de mes recherches (une pensée pour la grosse Bertha :-)).

À ma mère.

Résumé :
Aux alentours de 2005, la montée en fréquence des processeurs a atteint un pic. Depuis, les processeurs sont passés d'une architecture mono-cœur à une architecture multi-cœurs. De ce fait, les processeurs actuels que l'on peut trouver dans les serveurs ou les stations de travail sont plus complexes. Les tendances actuelles dans l'industrie informatique projettent une montée en nombre de cœurs toujours plus grande, suivant ainsi l'augmentation en nombre de transistor prédite par la loi de Moore. Cette augmentation rapide du nombre de cœurs dans les processeurs actuels ne se traduit pas par une mise à l'échelle en conséquence de la bande passante mémoire. Or les performances dépendent de plus en plus des motifs d'accès à la mémoire. De plus, les transformations permettant d'améliorer aussi bien la localité spatiale que temporelle, deviennent incontournables pour les applications de calcul haute performance.

Il existe un réel besoin d'outil d'amélioration de performance permettant de mieux appréhender les problèmes dont souffrent les applications parallèles, permettant ainsi de mieux tirer parti de l'énorme puissance de calcul disponible et qui ne cesse de croitre.

Dans cette thèse, nous présentons dans un premier temps l'outil d'analyse de performance MAQAO. Nous montrons comment il permet, grâce au couplage d'analyses statiques et dynamiques, de mieux cerner les problèmes d'évaluation de performance, qui possèdent en générale plusieurs facettes. Nous mettons ensuite en lumière la propriété d'extensibilité de l'outil à travers un langage de script reposant sur un cadre de travail riche en fonctionnalités. Enfin nous abordons notre méthodologie d'évaluation de performance allant des problèmes gros grain vers les problèmes à grain fin.

Nous décrivons dans un second temps la première contribution majeure de cette Thèse, à savoir un langage dédié à l'instrumentation permettant de construire efficacement des outils d'évaluation de performance à surcoûts réduits. Afin d'obtenir des surcoûts d'instrumentation réduits, nous combinons des analyses statiques et dynamiques. Nous illustrons la simplicité et l'efficacité de notre langage à travers l'exemple d'intégration de ce dernier dans l'outil d'analyse de performance parallèle TAU. Sur des applications de tests parallèles utilisant OpenMP, nous montrons dans quelle mesure notre approche fournit des résultats plus concis et avec moins de surcoût d'instrumentation en comparaison avec d'autres outils d'instrumentation binaire.

Finalement, nous mettons en avant un outil de caractérisation du comportement mémoire d'applications, qui représente la seconde contribution majeure. Notre travail se focalise aussi bien sur les problèmes mono-thread que multi-thread. Nous utilisons plusieurs analyses permettant de déceler des motifs d'accès à la mémoire inefficaces ainsi que de rechercher les problèmes liés à l'interaction entre plusieurs threads et affectant la hiérarchie mémoire (en particulier les caches).

Mots clés : analyse de performance, optimisation de code, analyse binaire, re-écriture binaire, langage d'instrumentation, caractérisation mémoire, réorganisation de données, analyse statique, analyse dynamique, OpenMP, applications multi-threads, simulation de cache

Abstract:

Around 2005, the increase in frequency of unicore processors reached a ceiling. Since then, processor architectures started a shift away from uni-core processors and towards multi-core processors. As a consequence, today's general purpose processors, found in most server and desktop machines, are very complex. Recent industry trends show that the number of cores in chips will still continue to grow, following the increase in the number of transistors predicted by Moore's law. This fast increase in the number of cores of modern Chip Multiprocessors (CMP) is not followed by a similar increase of memory bandwidth. Performance depends more and more on access patterns and transformations enhancing spatial or temporal locality are essential for high performance applications.

There is a need for performance tuning tools in order to gain a better understanding of issues suffered by parallel applications and thus to harness the ever-increasing available horsepower.

In this thesis, we first present the MAQAO performance analysis tool. We show how it combines static and dynamic analyses in order to better understand performance evaluation issues which are generally multifaceted. Besides insisting on the extensibility of the tool through a scripting plugin framework, we describe our top-bottom methodology, from coarse grain down to fine grain performance evaluation.

We then describe the first major contribution of this thesis, a domain specific instrumentation language (DSL) to easily build low-overhead performance evaluation tools. To achieve low overhead instrumentation, we combine both static and dynamic analyses. We illustrate the simplicity and efficiency of the DSL with the example of the integration in the TAU parallel performance tool. On parallel benchmark codes using OpenMP, we show how our approach provides lower overhead and more accurate results compared to other binary instrumentation tools.

Finally, we propose a tool to characterize the memory behavior of applications, which is the second major contribution. Our work focuses both on single threaded and multi-threaded issues. We use several analyses to detect inefficient access patterns and lookup for issues related to interactions between threads, and their impact on the memory hierarchy (caches).

Keywords: performance analysis, code optimization, binary analysis, binary patching, instrumentation language, memory characterization, data reshaping, static analysis, dynamic analysis, OpenMP, multithreaded applications, cache simulation

Contents

1 Introduction **1**

1.1 Context . . . 1
1.2 Objectives . . . 3
1.3 Outline . . . 3

2 Performance analysis in the multicore Era **5**

2.1 Introduction . . . 5
2.2 Memory bandwidth and latency wall . . . 6
2.2.1 Memory hierarchy . . . 6
2.2.2 Caches . . . 7
2.2.3 Scaling issues . . . 11
2.3 Complex architectures . . . 12
2.3.1 Flynn taxonomy . . . 12
2.3.2 Pipelined execution . . . 14
2.3.3 Multiple issue pipeline . . . 16
2.3.4 Vector extensions . . . 16
2.3.5 SMP: Symmetric MultiProcessing . . . 17
2.3.6 SMT: Simultaneous MultiThreading . . . 17
2.3.7 A modern example: Intel Nehalem microachitecture . . . 19
2.4 Leveraging parallelism . . . 22
2.4.1 Compiler strengths and limitations . . . 23
2.4.2 Importance of data layouts . . . 24
2.4.3 Parallel programming paradigms . . . 25
2.4.4 Dynamic execution side effects . . . 26
2.5 Performance analysis approaches . . . 27
2.5.1 Modeling . . . 27
2.5.2 Simulation . . . 27
2.5.3 Measurement . . . 27
2.6 Performance analysis tools . . . 29
2.6.1 Binary instrumentation . . . 30
2.6.2 Compiler . . . 31
2.6.3 Architecture simulation . . . 31
2.6.4 Introduction to Hardware Performance counters . . . 31
2.6.5 Tools . . . 34
2.7 Summary . . . 48
2.8 Conclusion . . . 50

3 MAQAO : Coupling static and dynamic analysis approaches **51**

3.1 Introduction . . . 51
3.2 Static analysis : code quality assessment . . . 52
3.2.1 Speedup levers on x86 Microarchitectures . . . 52
3.2.2 Improving code quality . . . 53
3.3 Dynamic analysis : capturing the dynamic behavior . . . 55
3.3.1 Architecture characterization . . . 56
3.3.2 Coarse grained performance evaluation . . . 56

3.3.3 Code characterization . . . 57
3.3.4 Fine grain performance evaluation . . . 59
3.4 MAQAO tool and Framework . . . 59
3.4.1 MAQAO Framework . . . 59
3.4.2 MAQAO Tool . . . 61
3.4.3 Contributions related to MAQAO . . . 66
3.5 Methodology . . . 67
3.5.1 Defining a goal . . . 67
3.5.2 Locate hotspots . . . 68
3.5.3 Characterize target hotspots . . . 68
3.5.4 Use relevant tools . . . 68
3.5.5 Iterating through the process . . . 68
3.6 Conclusion . . . 68

4 Instrumentation language 71
4.1 Introduction . . . 71
4.2 Related work . . . 72
4.3 Instrumentation Language . . . 73
4.3.1 Abstract code structure and Filters . . . 73
4.3.2 Complex instrumentation queries . . . 75
4.3.3 Instrumentation Probes . . . 75
4.3.4 Using MIL to reduce instrumentation overhead . . . 77
4.3.5 Configuration and environment . . . 77
4.4 Instrumented Code Generation . . . 78
4.4.1 Static binary instrumentation . . . 78
4.4.2 Advanced static analysis . . . 79
4.5 Building Performance Tools . . . 84
4.5.1 Integration in the TAU Performance System . . . 84
4.5.2 Loop centric profiling . . . 85
4.5.3 Memory tracing of OpenMP codes . . . 87
4.5.4 Dynamic extension of static prediction . . . 87
4.6 Experiments . . . 88
4.6.1 Instrumentation Overhead on Parallel Applications . . . 89
4.6.2 Real case example : QMC=Chem . . . 90
4.7 Conclusion . . . 93

5 Memory behavior characterization 95
5.1 Introduction . . . 95
5.2 Related work . . . 97
5.3 Compact multi-threaded trace collection . . . 97
5.3.1 Trace compaction techniques . . . 98
5.3.2 Our choice : Nested Loop Recognition . . . 99
5.3.3 Memory Trace . . . 100
5.3.4 Instrumentation time . . . 101
5.3.5 Reconstructing address streams . . . 102
5.4 Simplified cache simulator . . . 104
5.5 Single thread memory behaviour analysis . . . 104
5.5.1 Analyzing Strides . . . 106
5.5.2 Address based analysis . . . 106

5.5.3 Experiments 109
5.6 Understanding interactions between threads 110
5.6.1 Data sharing 113
5.6.2 Workload balancing 116
5.6.3 Affinity 116
5.6.4 Experiments 117
5.7 Data Reshaping 119
5.7.1 Related work 123
5.7.2 Overview 123
5.7.3 Reconstructing shared data structures 124
5.7.4 Analysis and hints generation 127
5.8 Conclusion 128

6 Conclusion 131
6.1 Contributions 131
6.2 Performance evaluation tools and methodology 132
6.3 Future work and research leads 133
6.3.1 Extending MIL 133
6.3.2 Extending MTL 133
6.3.3 Research aspects 134

Appendix 134

A Appendix A: MAQAO scripting examples 135
A.1 Example 1 : objdump-like binary desassembler 135
A.2 Example 2 : Printing function names of a binary 135
A.3 Example 3 : Maximum number of instructions in innermost loops . . 136

B Appendix B: STAN dynamic extension 137

Bibliography 145

List of Figures

2.1 Simplified overview of performance over time. 5
2.2 Starting with 1980, Gap in performance between processor and memory over the time. Source: Hennessy, Patterson: Computer Architecture, page 73, 5th edition, Morgan Kaufmann 7
2.3 Elements of a memory hierarchy 7
2.4 Intel Core i7 (Nehalem) memory hierarchy 8
2.5 Address decomposition and cache structure 9
2.6 Example of NUMA nodes: here 8 interconnected nodes. Source: Intel 11
2.7 Example of a NUMA machine: here a macro-node of 4 interconnected (QPI) sockets that can also be connected to other macro-nodes. Source: IBM 12
2.8 Example of a last level cache (Intel Sandy Bridge EP) with non uniform cache access (NUCA). Source: Intel 13
2.9 Flynn taxonomy including associated memory models and the some subdivisions of the MIMD class 14
2.10 Example of simple and pipelined (instruction) execution paths. Source: http://www.renesas.eu/ - search for "FAQ 1008742" 15
2.11 Example of SMP processor : Intel core i7 (Nehalem cores). Source: Intel. 17
2.12 Schedule obtained thanks to SMT compared to single and multi superscalar processors. Source: http://ixbtlabs.com/articles/pentium4xeonhyperthreading/index.html 18
2.13 2-Way SMT double execution pipeline. Source: Molester Waterball . 19
2.14 Overview of Intel Nehalem microarchitecture. Source: The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms by Dr. Thomadakis (Texas A&M) 20
2.15 Percentage of peak performance for four programs on four multiprocessors scaled to 64 processors. Source: Hennessy, Patterson: Computer Architecture, page 58, 5th edition, Morgan Kaufmann 22
2.16 Current and Projected performance development of TOP 500 clusters. Source TOP500.org 23
2.17 Overview of GCC Architecture: Source: GNU 24
2.18 An overview of the existing hardware performance counters ecosystems 33
2.19 Example of gprof flat profile output 35
2.20 Example of gprof callgraph output 36
2.21 Example of cachegrind output: function-by-function statistics 36
2.22 Coverage lists and call tree graph visualisation. Source: http://kcachegrind.sourceforge.net/html/Screenshots.html 37
2.23 On the right the call graph view. Source: http://kcachegrind.sourceforge.net/html/Screenshots.html 38
2.24 Multi-Programmed CMP$im Implementation Overview. Source [?] . 38
2.25 Cache Performance and Sharing Characteristics of a Multi-Threaded Workload using CMP$im. Source [?] 39
2.26 TAU parallel performance system overview. Source: [109] 41
2.27 Example of display with the ParaProf tool. Source: [109] 42

2.28 An overview of HPCToolkit's primary components and their relationships: Source: http://hpctoolkit.org/ . . . 43
2.29 Using hpcviewer to assess the hotspots. Source: http://hpctoolkit.org/ 43
2.30 Workflow. Source: http://www.openspeedshop.org/wp/category/presentations/ 44
2.31 PC Sampling experiment results displayed in the Open|SpeedShop GUI. Source: http://www.openspeedshop.org/wp/category/presentations/ . . . 45
2.32 Example of runtime summary analysis report in CUBE visualizer. Source: http://www.scalasca.org/. . . . 46
2.33 Example of performance view. Source: http://www.renci.org/focus-areas/project-archive/pablo . . . 46
2.34 Example of analysis. Source: http://www.tacc.utexas.edu . . . 47
2.35 Example of analysis with multiple passes. Source: http://www.tacc.utexas.edu . . . 48
2.36 Example of suggestion. Source: http://www.tacc.utexas.edu . . . 48
2.37 Classification of studied performance evaluation tools . . . 49

3.1 Simplified overview of the three latest Intel micro-architectures. Rough description of the front and back ends and the major execution pipeline enhancements . . . 53
3.2 MAQAO Profiler output for NPB3.3 bt benchmark (class.S). Each box exposes the name of a function along with its exclusive and inclusive percentage time (related to the walltime) . . . 58
3.3 MAQAO Framework . . . 60
3.4 Hierarchy of the structural abstractions defined in MAQAO . . . 61
3.5 Displaying load memory instructions . . . 62
3.6 MAQAO Tool overview . . . 63
3.7 MAQAO GUI : Web front-end . . . 64
3.8 Example of output produced by STAN . . . 66

4.1 Hierarchy of the structural abstractions defined in MAQAO . . . 74
4.2 Definition of an event with its action. . . . 75
4.3 MIL : MAQAO Instrumentation Language and its integration in the MAQAO framework. . . . 78
4.4 Part of the CFG of the main function (MAIN__) from the bt benchmark (using Intel fortran compiler with -O3) . . . 82
4.5 More efficient solution than int 3 trap handler on x86-64 architectures 84
4.6 TAU instrumentation file using MIL. . . . 85
4.7 Simple profiler all in MIL . . . 86
4.8 Distribution of the number of iteration per instance of a loop and per thread. Y axis reports the number of instances . . . 87
4.9 MIL file for a loop value profiling example . . . 88
4.10 Comparing overhead time on NAS benchmarks for MIL with probes in external library, MIL with probes in LUA (MILRT), Dyninst and Opari using TAU. X axis reports the overhead ratio compared to the original run. Lower is better. Overhead ratios greater than 10 are cut. A zero ratio means that either the concerned benchmark failed to compile or crashed at runtime . . . 90

4.11 Comparing the pprof profiling tool output of MAQAO and Dyninst for two threads . . . 91
4.12 Profiling results of the QMC==Chem application before and after transformation . . . 93

5.1 Percentage of memory accesses performed on shared blocks of data in hotspot loops of SPEC OpenMP benchmarks. Min and max correspond to the minimum and maximum ratios considering all the threads. The architecture used is a 2 socket quad-core 2.26Ghz Nehalem, with 8MB L3 and 256KB L2. . . . 96
5.2 DGEMM 64x64 Source loop and the NLR representation of its store statement . . . 100
5.3 Instrumentation overhead using the naive method . . . 101
5.4 Instrumentation overhead using the enhanced method. . . . 102
5.5 Assembly code of a memory instruction on x86_64 architectures. In general : MNEMONIC OFFSET(BASE,INDEX,SCALE),REGISTER 102
5.6 312.swim SPEC OpenMP 2001 benchmark : types and values of Z-polytopes of the most time consuming loop extracted from the memory trace . . . 103
5.7 Overview . . . 104
5.8 Components of an address. . . . 104
5.9 Content of a cache line of the cache simulator. . . . 105
5.10 Wrong access pattern . . . 106
5.11 Multiple strided access for one instruction . . . 107
5.12 Memory stream performance of a Store/Store/Store/Load pattern on A SandyBridge machine (Xeon E5). Left Y-Axis carries cycles per iteration. Right Y-Axis accounts for alignment of the third stream and finally X-Axis is the alignment of the fourth stream. First and second stream are fixed to 0. . . . 108
5.13 SPEC OpenMP 2001 benchmark : trace of the first most time consuming loop . . . 110
5.14 SPEC OpenMP 2001 benchmark : 324.apsi_m WCONT in apsi.f . . 111
5.15 Merged trace of threads for the store instruction, before and after transformation . . . 111
5.16 NAS Parallel benchmarks FT . . . 112
5.17 Trace of each thread for the store instruction after transformation . . 112
5.18 Source loop of the Dassault code . . . 113
5.19 Dassault plot . . . 113
5.20 Original code of the main hot loop of RECOM application. Note that the increment `INC` results from an indirection. . . . 114
5.21 On the left, we can see the 2D access pattern iterating only black tiles of a checkerboard. On the left, the splitting transformation solution.30% of gain has been obtained on this loop. . . . 114
5.22 LU decomposition application (OpenMP) on a 96 cores machine (4 nodes,16 sockets). Evaluates data sharing between Nodes,Sockets : Working set (shared,not shared) and Coherence based on shared cache lines (worst case). . . . 115

5.23 Evaluating work share strategies on a simple triangle traversal example on a 96 threads machine. Threads are sharing a 15MB array. X-Axis caries Affinity,Schedule,Chunk size triplets and Y-Axis accounts for executed cycles 117
5.24 Examples of workload balancing report between threads. X-axis refers to memory accesses (% of total accesses) and Y-axis to the work of each thread (from 1 to 96). The blue line represents the amount of accesses that each thread should do in order to have a uniform access over all the threads. 118
5.25 SPEC OMP 324.apsi run on 56 threads 118
5.26 Example of a sharing data plot 119
5.27 Shared data plots for galgel benchmark, syshtn.F loop at line 98. The lower diagonal pattern indicates read-only shared data among all threads. In X and Y axis, the ids of the threads. (a) For the 8-core Nehalem, each thread has 6% of its accesses on shared data with any other thread (b) For the 96-core Dunnington (only 24 threads are used). Each thread has 2% of its accesses that are shared with any given thread. 120
5.28 Shared data plots for galgel benchmark and loop at line 68 showing false-sharing among threads, according to different scheduling strategies and architectures: (a) STATIC, on 8-core Nehalem (b) GUIDED, on 8-core Nehalem (c) DYNAMIC, on 8-core Nehalem (d) GUIDED, on 96-core Dunnington, with only 24 cores used. 121
5.29 Shared data plots for apsi benchmark. (a) This lower diagonal pattern corresponds to a STATIC distribution of loop iterations, in large chunks. A cache line is shared only at the border of these chunks (b) An example of unbalanced shared data, here between scalars. 122
5.30 Typical access patterns of arrays shared by multiple threads. Each thread T1 to Tn reads a subset of the same array. P1 to P4 represent common access patterns : interleaved, interleaved with overlap, consecutive and separate (structure fields) 125
5.31 Trace of each thread for one instruction (a). Resulting polytope describing the shared region (b). Representation of the shared region (c). 126
5.32 Candidate group of instructions for merging (a). Merged region (b) . 127

B.1 Shell script to execute the differents steps 143
B.2 MIL script to get instances and cycles information 143

List of Algorithms

1 InsertProbe . . . 83

2 MAIN . . . 123
3 Function MergeThreads . . . 125

List of Tables

4.1 Structural abstractions and associated events. 73
4.2 Structural abstractions and associated filter mechanisms. 74
4.3 Comparison between MAQAO and IFORT on the number of cycles measured for the three most time consuming loops 92
4.4 Comparison of execution times (in seconds) between MAQAO and IFORT 92

5.1 Table showing for each benchmark (same as before) the mean size of the working set of each thread. 95

Chapter 1

Introduction

1.1 Context

Around 2005, the increase in frequency of uni-core processors reached a ceiling.

Since then, processor architectures started a shift away from uni-core processors and towards multi-core processors. As a consequence, today's general purpose processors, found in most server and desktop machines, are very complex. The most recent and famous ones, in the High Performance Computing (HPC) field, are the Intel Xeon, AMD Opteron, SPARC (developed by Fujitsu, Sun Microsystems and Texas Instruments), IBM POWER7 series.

Simultaneously, processor architectures have shifted away from uni-core processors to- wards multi-core processors. Consequently, the general purpose processors found in most server and desktop machines today are more complex. The most recent and famous ones in the High Performance Computing (HPC) field are the Intel Xeon, AMD Opteron, SPARC (developed by Fujitsu, Sun Microsystems and Texas Instruments) and IBM POWER7 series. These CPUs are superscalar (issuing more than one instruction per cycle), out-of-order, multi-core (symmetric multiprocessing or SMP) and even multi-threaded (simultaneous multi-threading or SMT) for some. Each new processor generation brings greater optimizations by enhancing the execution path to improve the overall performance compared with the previous generations. Enhancing the execution pipeline involves ameliorating existing mechanisms and leveraging resources. Hardware data prefetching, branch prediction, cache size, memory accesses (better handling of unaligned accesses and memory disambiguation) are common examples. Moreover, many processors introduce enhancements targeting the HPC field. Fujistu SPARC64 introduces HPC-ACE extensions (High Performance Computing - Arithmetic Computational Extensions) and Intel, followed by AMD, periodically improve its vector extensions. Precisely, one of the most bankable technique, when scaling with the rest of the architecture, is increasing the length of vectors. Theoretically, the computation throughput increases with the vector length (when vectorization is possible at the compiler level).

The fast increase in the number of cores of modern Chip Multiprocessors (CMP) is not followed by a similar increase of memory bandwidth. Performance depends more and more on access patterns and transformations enhancing spatial or temporal locality are essential for high performance applications. On shared-memory multi-core architecture, the benefits of spatial locality depends on the type of memory access. This locality improves cache usage by making use of the hardware prefetcher, by reducing memory pressure (enabling vectorized loads for instance), or by bringing in a shared cache memory blocks used by other cores. However, if one core writes a memory block from a shared cache while the others read data from the same block, the delays added by the shared-memory coherency protocol may be detrimental to performance. Similarly, false-sharing situations can significantly

degrade performance depending on the level of the memory hierarchy involved and the temporal locality of the multi thread access.

Recent industry trends show that the number of cores in chips will still continue to grow (see SCC [50], MIC [105, 37, 95, 91] or Tera-scale project [96] for instance), following the increase in the number of transistors predicted by Moore's law [12]. The cache hierarchy bridging the increasing speed disparity between processors and memory plays a critical role for achieving the best performance on multi-core machines. As the number of cores increases, the number of caches, their complexity (NUCA [32]) and levels of the hierarchy has also increased and has led to ccNUMA behaviors. Cores on the same chip can share some cache space and performance highly depends on how they use this space [20]. The structure of caches, their limited capacity and different hardware mechanisms, such as prefetch or cache coherency mechanisms are essential factors for the overall performance of multi-threaded codes.

The bottom line is that we will have tremendous horsepower for the taking, with the advent of the Exascale Era, but we will not be able to easily take advantage of it. Applications developers need to be involved at a higher degree compared to the past. Practically, applications must expose some of the algorithms and data organization more explicitly in order to take full advantage of these architectural features, allowing compilers to more effectively optimize the code generations and data layout. For instance, at the moment, using hybrid shared and distributed memory programming models is one solution to take advantage of current architectures [93]. The shared model inside a nide (threads) and distributed memory between nodes.

In "A Conversation with John Hennessy and David Patterson" [89], John Hennessy and David Patterson, summarized their thoughts according to the context described above. Quoting David Patterson:

> "I think today this shift toward parallelism is being forced not by somebody with a great idea, but because we don't know how to build hardware the conventional way anymore. This parallelism challenge involves a much broader community, and we have to get into applications and language design, and maybe even numerical analysis, not just compilers and operating systems. God knows who should be sitting around the table - but it's a big table. [Computer] Architects can't do it by themselves, but I also think you can't do it without the [computer] architects."

Quoting John Hennessy:

> "The fundamental problem is that we don't have a really great solution. [...] [H]ow we're going to change our programming languages; what we can do in the architecture to mitigate the cost of various things, communication in particular, but synchronization as well.Those are all open questions in my mind. We're really in the early stages of how we think about this. If it's the case that the amount of parallelism that programmers will have to deal with in the future will not be just two or four processors but tens or hundreds and thousands for some applications, then that's a very different world than where we are today."

Even though some auto-tuning methods or frameworks exist for very specific problems [57, 122, 31], in general, performance tuning is the common way to find out

how an application behaves. There are many performance evaluation tools dealing with different issues. Some try to address specific problems by providing feedback though hints related to the source code, while others provide general information, statistics and even value profiling. Tools can also be classified depending upon their granularity, for instance, function level, loop level or instruction level.
Usually, application developers have one or a set of tools that are systematically employed. However, there is no real methodology or glue between the different tools, which most of the time are presented as competitors. A form of methodology appears with the knowledge acquired throughout performance tuning experiences.

1.2 Objectives

In the introduction, we illustrated the critical role of memory and its related hierarchy in current and future architectures and consequently why we focus on memory related issues in these studies. To concentrate our effort, we selected the shared memory model, because of the future trends of industry, multiplying the number of cores per node (building block for distributed architectures). In particular, we have chosen the OpenMP programming model and Intel architectures (processors) as main targets.

Another major concern, is the ability to easily build performance evaluation tools when no other tools suit our needs. Besides the previous objective of a memory behavior characterization tool, we wanted to have a simple tool to perform the profiling of OpenMP applications at function and, especially, loop level. Thus, the need for a robust and flexible instrumentation framework.

With all these objectives, we wish to provide the HPC community with new approaches and tools and enrich the MAQAO Tool [69].

1.3 Outline

The first part of this dissertation, Chapter 2, introduces principles and aspects of multi-core architectures, focussing on the need for parallelism and present the performance analysis methods and tools. Chapter 3 presents our new analysis approach which couples static and dynamic analyses. We also introduce the MAQAO tool and the underlying Framework, along with the definition of a performance analysis and tuning methodology. In Chapter 4 we detail our domain specific instrumentation language. In Chapter 5 we describe our memory tracing infrastructure and how it provides a mean to characterize the memory behavior of multi-threaded applications. Finally, we conclude with the contributions of this thesis, mid-term future work and research leads.

Chapter 2

Performance analysis in the multicore Era

2.1 Introduction

The advent of multi-core processors brought its share of issues. Since multiple processors share the same set of resources, less resources are available for each processor and conflicts appear frequently. Due to the complexity of modern architectures, including the memory bandwidth and latency limitations, more work is expected from the programmer.

Figure 2.1 illustrates the transition between the frequency (uni-core) increase Era and the multi-core Era.

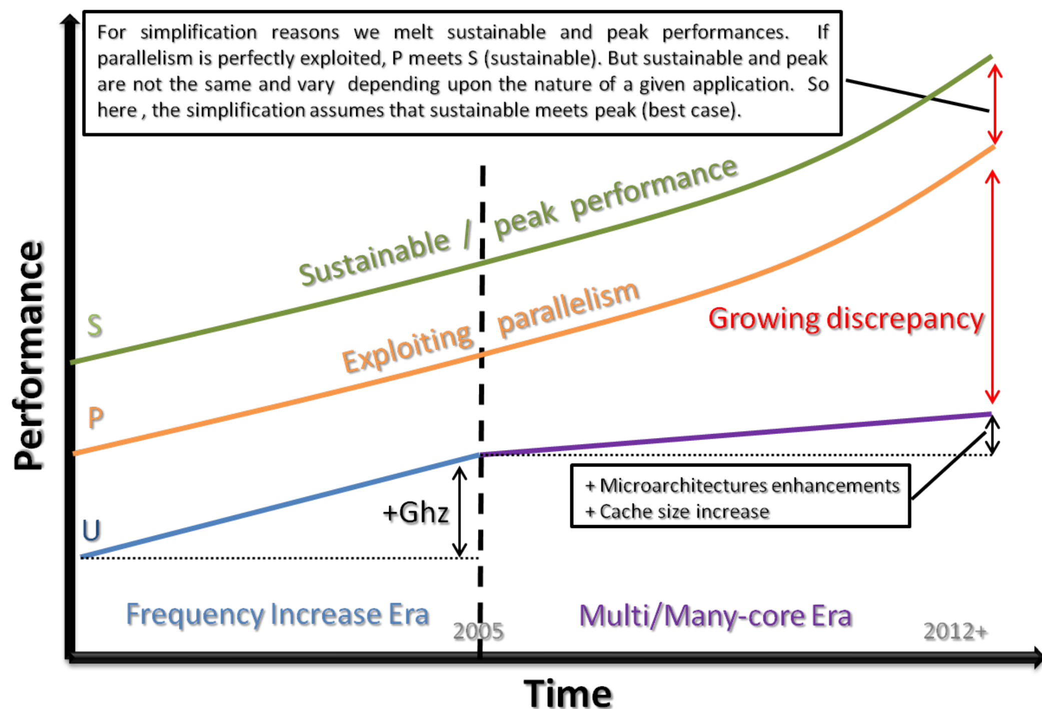


Figure 2.1: Simplified overview of performance over time.

It is a rough overview of global performance of architectures over time. The transition occurs around 2005 when the increase of frequency of processors is no more a solution to bring more performance. This was due to physical constraints, heat reached a ceiling. Also because it was the simplest manner for processor architects. Besides optimizing an application, the mechanic increase of frequency provided developers with significant, and "free", additional performance each new generation. Since the advent of multicore processors, additional performance is for the taking

only if programmers exploit it. That is why we can observe a growing gap between uni-core and parallel performance. In this figure, sustainable performance is the maximum level of parallelism that could be achieved if the application was perfectly parallelized. For simplification reasons we melt sustainable and peak performances. If parallelism is perfectly exploited, P meets S (sustainable). But sustainable and peak are rarely the same and vary depending upon the nature of a given application. So here, the simplification assumes that sustainable meets peak (best case). Peak performance can be thought of as the higher bound performance that will never be exceeded, on a given architecture. The bottom line is that we are getting more and more horsepower but without being able to exploit it as easily as before, because the compiler and the increase in frequency alone cannot cover the gap anymore. Adequate programming models must be chosen and adapted to this new environment in order to harness such resources.

Besides the selection of appropriate programming models, performance tuning remains the keystone for performance enhancements. Many performance tuning tools exists and try to address different problems. Application developers tend to select one tool for performance tuning which is a difficult task because there is no magic tool that fix it all. Depending upon a given programming model, the priorities and the constraints an application developer may have, the valid selection of tools may not be the same.

The chapter is organized as follows. Section 2.2 provides an introduction to the problem of memory bandwidth and latency wall, including the description of the memory hierarchy, its caches and the scaling problem of multi-core chips. Then in section 2.3, the complexity of modern architectures is discussed. After that, section 2.4 presents the important factors to take into account in order to leverage parallelism. In section 2.5 we will see the different approaches to performance analysis. Before concluding, section 2.6 provides a categorization of existing performance analysis tools, underlying frameworks and a summary of addressed issues

2.2 Memory bandwidth and latency wall

Since 1980, the gap between processor and memory performance has increased. Figure 2.2 illustrates the slower growth of memory performance over processor performance. In this section, we will first present the structure of the memory hierarchy between the processor and the main memory. Then, we will introduce caches, which are the building block of the memory hierarchy. Finally, issues related to resources scaling in modern processors will be discussed.

2.2.1 Memory hierarchy

To alleviate the discrepancy between the processor and memory performances, a hierarchy of memories have been introduced. The two main characteristic factors when considering memory are the size (storage) and access speed. From the processor's point of vue, the main memory (RAM) has a vast addressing space but at the cost of a high latency. Conversely, accessing onboard memory, .i.e. the cache, is quite immediate but with less storage space. Hence a global tradeoff between size and access speed.

Figure 2.3 presents the elements of a memory hierarchy.

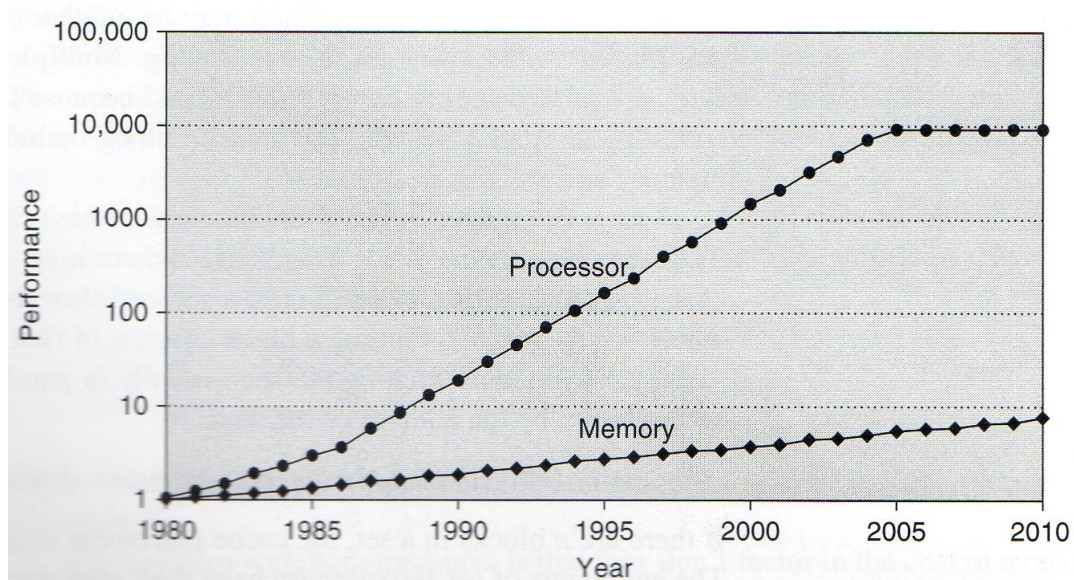


Figure 2.2: Starting with 1980, Gap in performance between processor and memory over the time. Source: Hennessy, Patterson: Computer Architecture, page 73, 5th edition, Morgan Kaufmann

Figure 2.3: Elements of a memory hierarchy

Registers of the CPU are the fastest and smallest available memory. Between registers and memory, we can find levels of caches, each one being slower and with more storage space. It is also possible to have further elements after the main memory to temporarily store pieces of memory, when the latter is full, e.g. a hard drive. Each element can be on the processor's die itself or on its hosting board. Most recent processors have on-die three-level caches. Figure 2.4 shows the memory hierarchy of the Intel Core i7 (Nehalem) processors. There are three levels of cache which are connected to the main memory and can also have access to a remote memory, i.e. of another processor.

2.2.2 Caches

As we mentioned above, caches are the main component of a memory hierarchy. A cache is a small piece of fast memory which is split into lines of memory blocks. Cache management and utilization involves multiple techniques and mechanisms that will be detailed below.

2.2.2.1 Structure

Figure 2.5 describes the structure of a cache and how an address is decomposed in order to verify if it is already present in it. The address is split into tag, set number

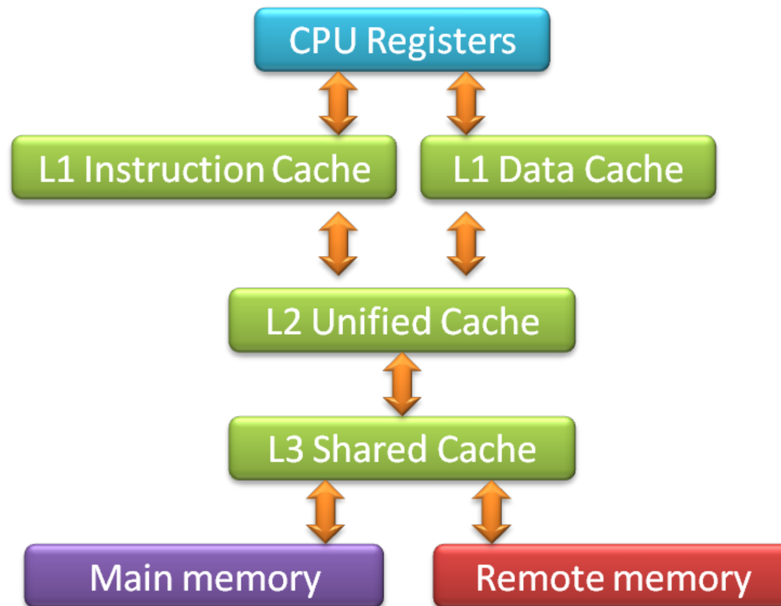


Figure 2.4: Intel Core i7 (Nehalem) memory hierarchy

and byte offset. At a given point, a cache contains blocks of memory words stored in lines. Each line may contain multiple words. The byte offset gives the position inside a line. Lines are grouped into sets. The set part of an address is actually a hash index to find out a set. Lines of a set are distinguished thanks to the tag. Hence finding a tag is an associative search within a given set. For instance, when accessing an element of an array, the caches are questioned, from the lower to the highest level, about the presence of that element based on its address. The address is decomposed and the first step is to locate the correct set. Then, the target line is found thanks to the tag. Finally the data element can be extracted thanks to the offset in the cache line, and placed into a register.

2.2.2.2 Addressing

In Figure 2.5 we introduced the concept of n cache lines arranged into groups, i.e. *sets*. The number of lines in a set are called ways ($n - ways$) and defines the associativity of the cache. There are actually three types of cache addressing:

- Direct mapped: n is equal to one. It means that a block of memory can only go in a unique line.
- Fully associative: n is infinite. Actually it corresponds to the number of lines in the cache. Each block of memory can be stored in any cache line.
- n-way associative: n is small, usually 2 to 64.

Direct mapped caches have the advantage of being very simple to implement. This main advantage turns to be a huge drawback when considering multiple collisions scenarios. In other words, addresses of the same region can only fit in one cache line. Fully associative cache would be the best choice if only they could be implemented easily. It is actually impossible to maintain low latency accesses when using complex hashing logic. Most of the time, n-ways associative caches offers a nice tradeoff between both worlds.

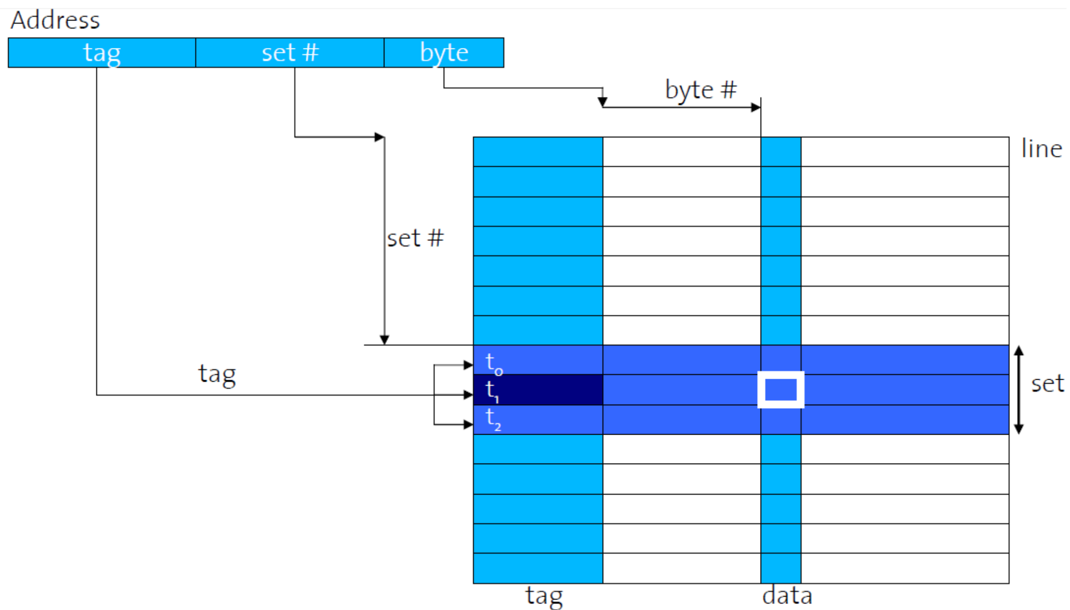


Figure 2.5: Address decomposition and cache structure

When data corresponding to an address is in the cache, it is considered a *hit*. Conversely, when data is not present in the cache, it provokes a *miss*. There are four types of misses:

- Compulsory: the first time some data is read, it is not in the cache, and will automatically provoke a series of misses.
- Conflict: happens in set-associative caches when all the ways are occupied.
- Capacity: occurs when all cache entries are occupied by other data.
- Coherence: when the hardware coherence protocol detects an incoherent state between two memory accesses. This scenario occurs in a multi-core environment sharing the same memory space.

2.2.2.3 Write policy

When reading a cache line, we saw that we could either perform a *hit* or a *miss*. The alternate case to take into account is writing a cache line. There are two main approaches to consider:

- Write back: a written cache line may stay in the cache until it really needs to be written to main memory.
- Write through: a store operation systematically updates the cache and memory.

When a store operation to a line not currently in the cache is detected, two strategies may be taken, namely, write allocate and no-allocate. The former allocates a new cache line and writes it, whereas the latter directly writes into memory, bypassing the cache. Pratically, processors use typical combinations such as write-back along with write-allocate and write-through along with no-allocate.

On the one hand, write through policy ensures a coherent state between the cache and memory, but generates more memory traffic leading to quicker contention. One the other hand, write back policy maintains an inconsistent state between cache and memory which has the advantage of alleviating the pressure on memory traffic. The drawback in multiprocessor systems is that is implies additional resources to maintain a coherent state between the processors and memory. Indeed, multiple copies of the same data may exist at the same time in distinct processors.

2.2.2.4 Locality

Usefulness of caches is based the concept of locality. There are two types of locality:

- Spatial: when an element is accessed, the next ones may also be accessed. Accessing contiguously the elements of an array is an example of spacial locality.

- Temporal: the same elements may be accessed in a near future. Regularly accessing different data is an example of temporal locality.

If an application exhibits this kind of properties, then it will take advantage of the memory hierarchy. Practically, many optimizations consists in coping with a cache size. For instance, a blocking optimization can be performed on a level of cache depending on the minimum size of data to manipulate. If data fit in a cache, then there is no need to read information from memory each time and a huge speedup in performance will be achieved.

2.2.2.5 Types of caches

Let us consider again the example of Figure 2.4. We observed three levels of caches. A hierarchy of cache is not just a pile of growing size caches. Each cache level has specific characteristics besides its size. We will present the specificities of each level of cache of our example.

Level 1 - Separate instructions and data Current architectures are a mixture of Harvard and Von Neumann architectures. While using one unified memory addressing scheme, data and instructions are actually separated in order to be able to processes at the same time instructions and data. Level 1 cache is split into an instruction cache and a data cache. These caches are small enough to be quickly accessed (1-3 cycles).

Level 2 - Unified and victim cache The second level cache is a unified cache, meaning that it can contain either data or instructions. It is usually used as a victim cache, receiving cache lines dropped from first levels caches. Its size is usually bigger by a factor of 4, which ensures a tradeoff between size and access time (4-15 cycles).

Level 3 - Last level shared cache In multi-core chips (CMP), level 1 and 2 and core-specific whereas level 3 is shared among all the cores. The latter can be inclusive or non-inclusive, meaning that it can, or not, replicate the contents of the lower level caches.

2.2.3 Scaling issues

Given the current multi-core architectures, there is no choice but to integrate more cores in a single package (die) in order to get more power. Doing so is not an easy task because multiplying cores within the same processor introduces problems. The main problem is resource contention. The mosr critical resource is memory accesses. More cores are getting added while the memory hierarchy does not scale. The evolution of Intel chips is a good example. Until the Nehalem-EP micro-architectures, the memory controller (IMC) was not integrated on chip, but was are non-uniform memory accesses (NUMA). Actually it is cache-coherent NUMA (ccNUMA) since coherence must be maintained not only at (multi-core) processor level but also between multiple of these. Figure 2.6 presents a NUMA, shared memory, architecture containing up to 8 sockets or NUMA nodes, interconnected through dedicated channels (Quick Path Interconnect in Intel chips). Such example is used in real products like Fujitsu PRIMERGY servers [36].

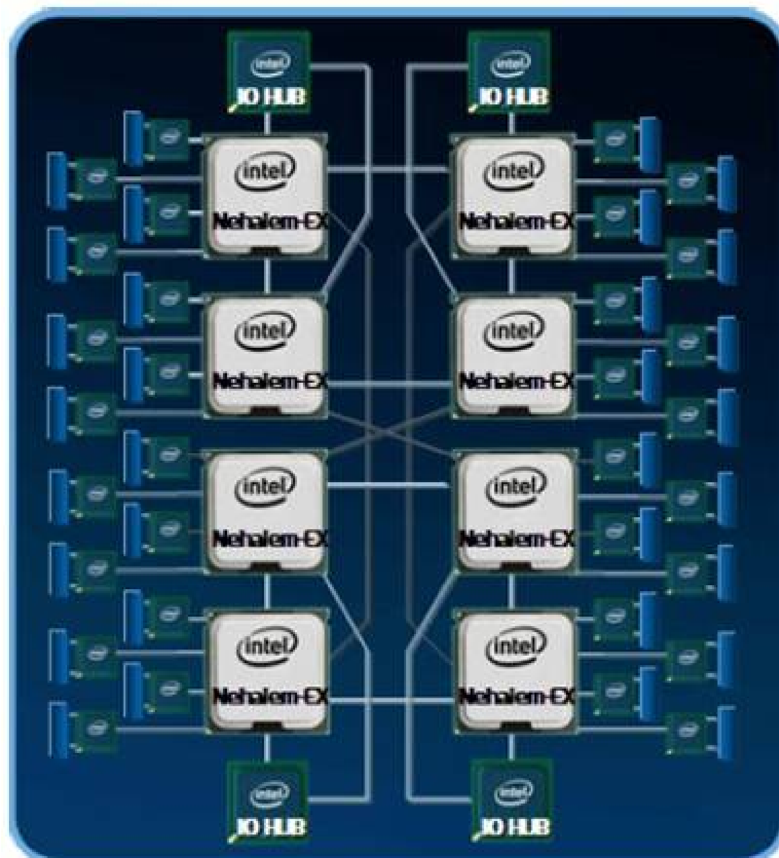


Figure 2.6: Example of NUMA nodes: here 8 interconnected nodes. Source: Intel

Practically, it is possible to go one step further with distributed shared memory (DSM) architectures. The architecture presented in Figure 2.6 is considered as a building block for wider architectures. For instance, DSM architectures proposed by IBM can link up to four building blocks to obtain a single system image. IBM introduced a fourth level of cache to enhance performance between each building block. Another example of huge DSM architecture is the Altix UV 1000 [108] systems manufactured by SGI and which can contain up to 2560 cores. Connections between each building block is guaranteed by NUMAlink[107] connections. It also

possible to transform multiple independent machines into a one system image thanks to software solutions like ScaleMP [110] (Versatile SMP architecture).

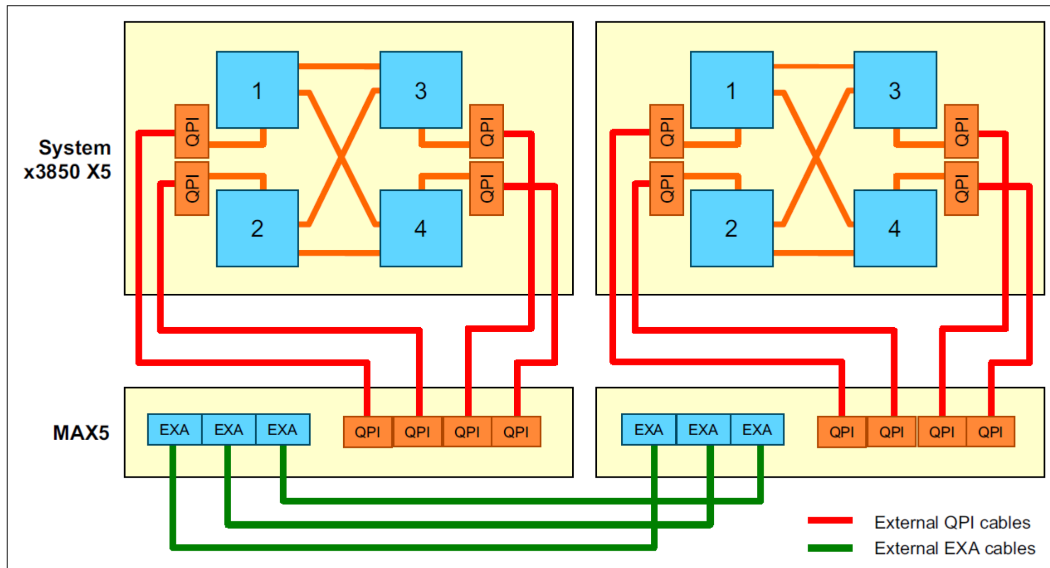


Figure 2.7: Example of a NUMA machine: here a macro-node of 4 interconnected (QPI) sockets that can also be connected to other macro-nodes. Source: IBM

The easiest way to update such systems in order to significantly increase the performance is to add more cores per processor. But as mentioned before, there are scaling issues to such a mechanic approach. For instance, progressing from six cores to eight cores lead Intel to think to a new way of sharing the last level cache between all the cores, because the performance was not scaling. Starting with the Nehalem-EX micro-architecture, the last level cache uses a split cache linked by a bidirectional ring network. Figure 2.8 illustrates such a cache structure. Each core has its dedicated part of the last level cache but can share data thanks to a communication protocol. Splitting the cache into smaller pieces introduces non-uniform cache accesses (NUCA).

2.3 Complex architectures

Modern processor architectures combines several architectural techniques to achieve the highest performance. In order to further understand the issues that can arise due to the switch from uni-core to multi-core processors, we must first introduce the involved techniques and models. We will first introduce an extended version of Flynn's taxonomy in order to categorize the different classes of existing architectures. Then we will study the main concepts used in today's architectures. Finally, we will present an example of modern processor and describe how all the mentioned concepts are implemented in its micro-architecture.

2.3.1 Flynn taxonomy

Applications are a succession of instructions that manipulate data in a given memory model. Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn back in 1966. It is considered as a simple but efficient way

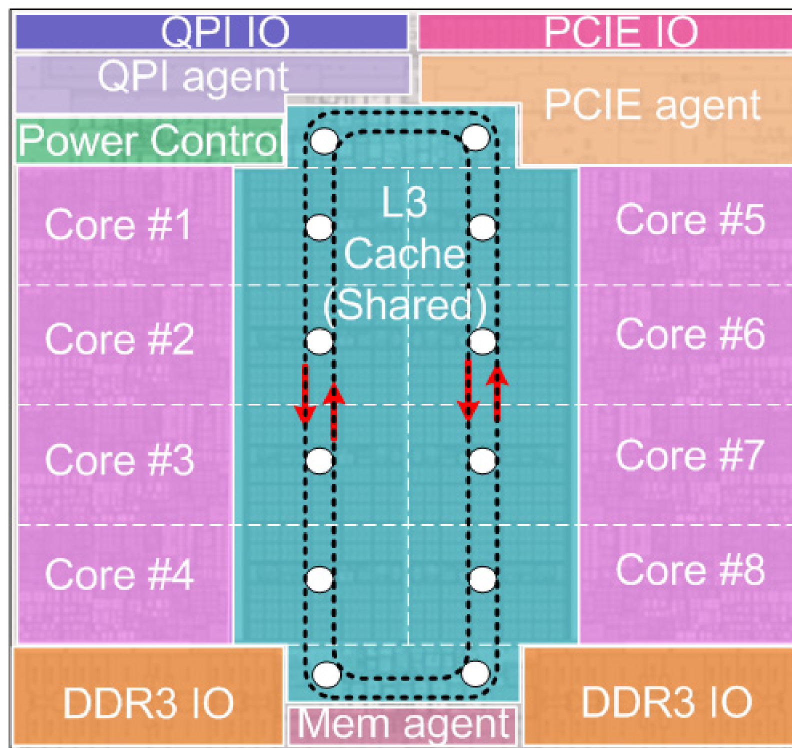


Figure 2.8: Example of a last level cache (Intel Sandy Bridge EP) with non uniform cache access (NUCA). Source: Intel

to classify high-performance architectures. Four main classes of architecture are defined:

SISD: Single Instruction Single Data This kind of architecture can only execute one instruction with one associated data. The execution is then accomplished in a serial fashion. Older uniprocessor desktop machines are a good example. The CPUs were actually not using pipelined execution flow as modern CPUs do.

SIMD: Single Instruction Multiple Data In this class, multiple data can be manipulated by the same instruction to perform, in a lock-step, operations which may be parallelized. This is usually the case when considering operations on arrays. In recent processors, we can find vector extensions, which are actually under the form of vector functional units, that can process multiple elements of a given array (block of memory).

MISD: Multiple Instruction Single Data Theoretically, in this kind of architectures, multiple instructions can manipulate a single flow of data. This class of architectures are not common but very useful when considering redundant execution contexts. If we consider an airplane decision system, the same data is actually processed by different execution units. This approach enables the global system to be fault tolerant and based on result agreement (majority).

MIMD: Multiple Instruction Multiple Data This type of architecture is able to execute at the same time multiple instructions that can act in parallel on distinct

data flows. This class shows the limit of Flynn's taxonomy because different kind of architectures can fall in this same category.

In order to have a better insight on this category, we will further subdivide it into SPMD and MPMD subclasses

SPMD: Single Program Multiple Data As evoked previously, a program is a set of instructions. Depending on the control flow of a program, a multiprocessors system may execute the same instance of an application but at independent points or phases on different data. Nowadays, SPMD is the most common style of parallel programming. The main (most used) programming models are message passing, with MPI, and data parallelism with OpenMP. The latter can be considered as SPMD because when reaching a parallel region, distinct threads will execute the same program that may execute different instructions on different data (phases).

MPMD: Multiple Program Multiple Data SPMD can be generalized to MPMD when more that one program is involved. We can think of producer/consumer scheme. The producer applications are responsible for generating data that is sent and consumed by the consumer applications. Results can then go back to the producer in order to take the decision of stopping or not the production process. Programming on a Cell **??** architecture involves using a primary processing unit and secondary processing units.

Depending upon the considered memory model, extended abbreviations may be used, namely, Shared Memory abbreviated SM-[S|M]PMD and Distributed Memory or DM-[S|M]PMD.

Figure 2.9 depicts the four classes of architectures defined by Flynn's taxonomy. We added the memory models that can be associated to it and the main subdivisions of the MIMD class.

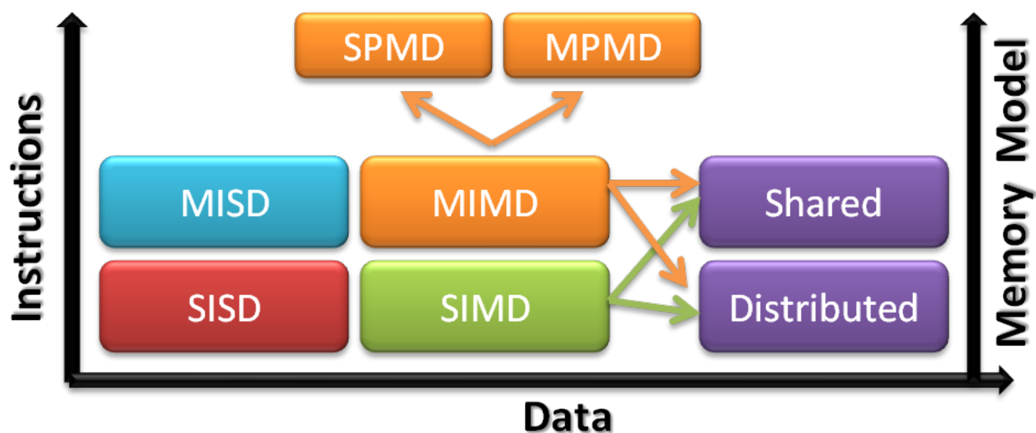


Figure 2.9: Flynn taxonomy including associated memory models and the some subdivisions of the MIMD class

2.3.2 Pipelined execution

Superscalar processors are able to process more than one instruction at a time. This fact is actually made possible thanks to the principle of pipelining. Fig-

ure 2.10 depicts an example of pipelined execution path opposed to a simple one (non pipelined).

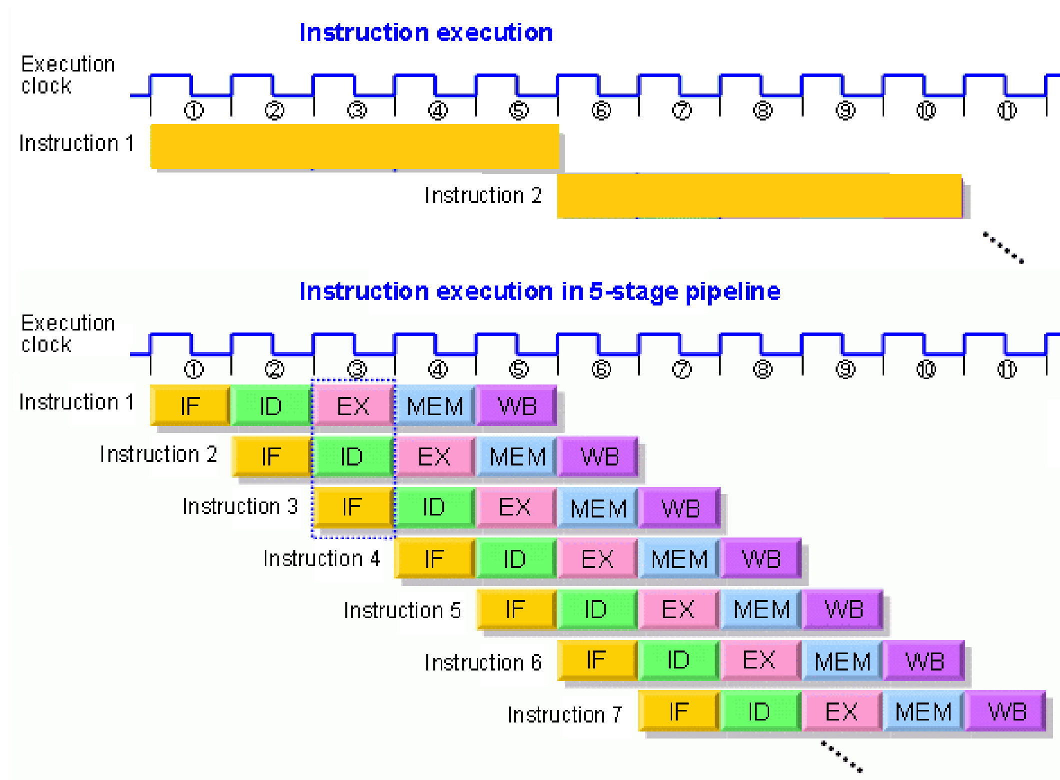


Figure 2.10: Example of simple and pipelined (instruction) execution paths. Source: http://www.renesas.eu/ - search for “FAQ 1008742”

Each step is briefly described blow:

- IF: Instruction is fetched from cache (or memory)
- ID: Instruction is decoded, i.e. recognized
- EX: Execution involves operations such as computations (ALU)
- MEM: Perform memory accesses
- WB: Write back of execution result to destination register

In the case of a simple execution path, only one instruction can occupy the available hardware resources throughout the execution path. Thus, the processor can accept only one instruction after the other in a serial fashion. Since the execution path is actually composed of multiple stages, it would be possible to split it. This is when pipelining comes into play. When pipelined, an execution path can process as much instructions as stages it contain. Instead of executing one instruction each x cycles, where x is the minimum length of the execution path, it can actually execute one instruction per cycle when the pipeline is full (considering that each stage requires only one cycle). Practically, it will depend upon the bottlenecks that can appear within the pipeline, the worst scenario being a pipeline flush. Indeed,

some factors such as dependencies between instructions, conflicts of hardware resources, and execution of branch instructions, will provoke a stall (stop) or require a re-execution. As a consequence, the execution pipeline is slowed down. These issues are know as "pipeline hazard". In a nutshell, there are three types of pipeline hazards, namely, conflict, control and data hazards. Each hazard can stall one or more stages of the execution pipeline until resolved. Resources hazards can only be addressed by duplicating hardware resources. Control hazard related to branching can be prevented by executing the branch destination instruction in advance with the branch prediction function. Compilers can handle problems between instructions by changing the sequence they form thanks to dependency analyses. There are four types of dependencies between registers:

- Data dependency: it avoids the parallel execution of two instructions when the second needs to read to result of the first. It is also called RAW (Read After Write), true or flow dependency.
- Anti-dependency: it is the opposit of a data dependency also called Write After Read.
- Output dependency: exists between two instructions writing the same destination, one after the other. It is also know as WAW (Write After Write) dependency.
- Input dependency: exists between two instructions reading the same destination, one after the other. It is also know as RAR (Read After Read) dependency.

Data dependencies (data) conditions the degree of the instruction-level parallelism because they introduce a latency stalling the MEM stage. WAW and WAR dependencies are solved through register renaming since their existence is only due to a lack of registers (at the allocation stage). Processors have more registers than the ones that are accessible by the compiler. Internally, processors have several additional registers and a subset of them are used for register renaming. RAR dependencies have actually no impact and are only defined for completeness' sake.

2.3.3 Multiple issue pipeline

As mentioned before, superscalar processors can achieve, thanks to pipelining, the execution of up to one instruction per cycle when discarding pipeline hazards. A simple approach to improve instruction level parallelism is to increase the number of instructions that can be fetched. We can think of it like having multiple pipelines in parallel but within the same resource field. However, this approach also introduces the possibility of more bottlenecks. Since the basic idea is to take advantage of all the available resources, the counterpart is the higher probability of resource conflicts.

2.3.4 Vector extensions

Vector processors are based on an SIMD model. They can performance a single instruction on multiple data. In general, it consists in processing consecutive elements of a one-dimensional array, i.e. vectors. Thus, when nothing prevents processing multiple data elements in parallel, the instruction-level parallelism can be increased by the factor of the number of elements contained in the vector. For instance, the

Intel Sandy Bridge micro-architecture features 256 bit wide vectors (AVX) which can contain four double precision or eight single precision floating-point numbers. Hence a factor eight (or four) speedup.

2.3.5 SMP: Symmetric MultiProcessing

After the frequency scaling Era, i.e. impossibility to increase frequency anymore, manufacturers started duplicating cores in order to leverage processing power. Symmetric MultiProcessing (SMP) is an architecture that involves identical processor cores attached to the same memory sub-system. Cores are connected through internal buses. They usually share a level of cache (last level). At the Operating System level, an SMP system can easily migrate tasks between processors to balance the workload efficiently. Moreover one task can only be executed by one of the available cores. Modern processors implements the SMP architecture. Figure 2.11 presents the example of an Intel core i7 processors containing four Nehalem (micro-architecture) cores.

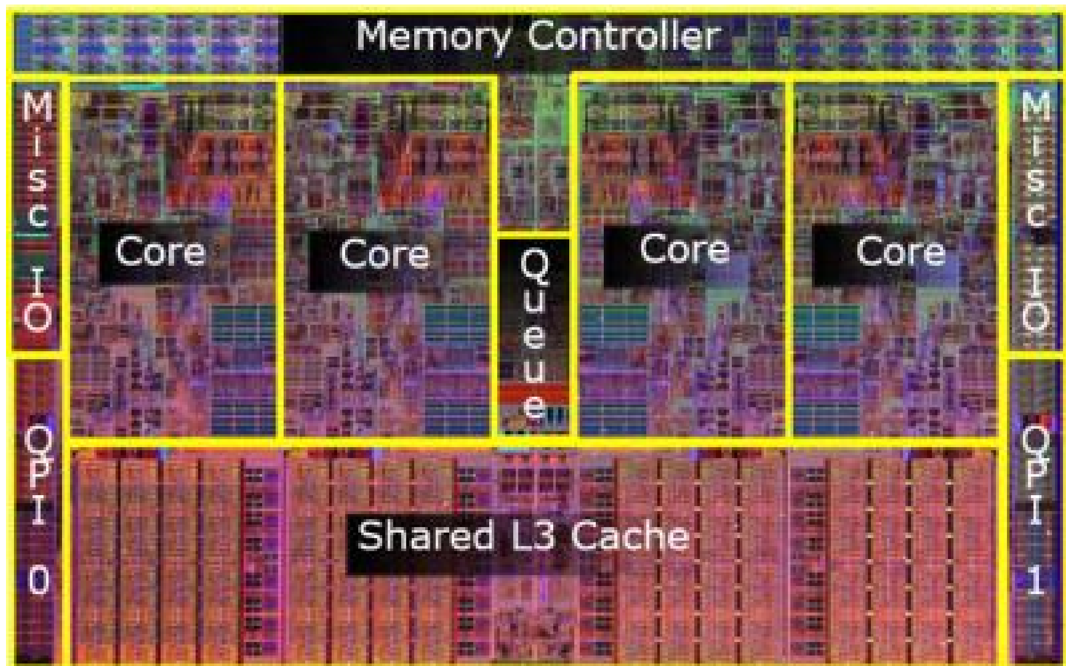


Figure 2.11: Example of SMP processor : Intel core i7 (Nehalem cores). Source: Intel.

2.3.6 SMT: Simultaneous MultiThreading

In general, a superscalar core is rarely saturated, that is to say using all its execution units at the same time. Simultaneous multi-threading is a technique to increase instruction-level parallelism in such cores. The basic idea is to have multiple physical threads that can overlap each other in order to exploit, in the best manner, available hardware resources. Moreover, since each thread has its own context, no costly context switching is needed. This is extremely important when considering architectures using out-of-order execution engines. The instruction-level parallelism

can then be improved by making one thread using functional units that are not being occupied by the other thread. Applications can be divided into two main classes, either dominated by memory accesses or by computations. That is why, practically, some modern implementations supports the simultaneous execution of two threads in one processor core. But in reality, this scheme is only efficient if the code has specific properties. A favourable scenario is to have one thread performing computations while another is waiting for data from the memory subsystem. In this case, we can consider this approach as thread-level parallelism since two distinct threads are actually alive at the same time. Figure 2.12 presents an example of scheduling (functional units allocation) when using simultaneous multi-threading compared to single and multi superscalar configurations. We can notice that SMT maximizes resource allocation. In the current industry, we can find Intel's hyperthreading technology.

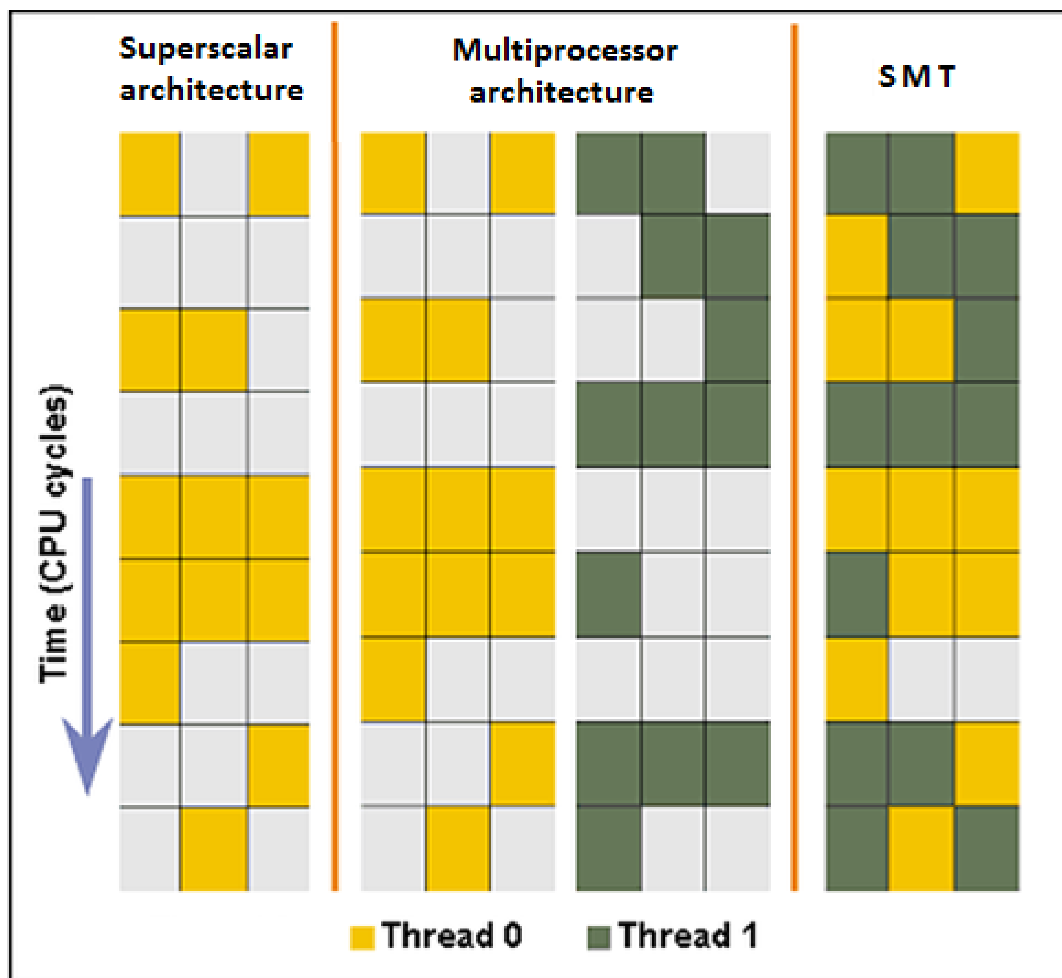


Figure 2.12: Schedule obtained thanks to SMT compared to single and multi superscalar processors. Source: http://ixbtlabs.com/articles/pentium4xeonhyperthreading/index.html

Figure 2.13 shows an example of 2-Way SMT superscalar execution pipeline.

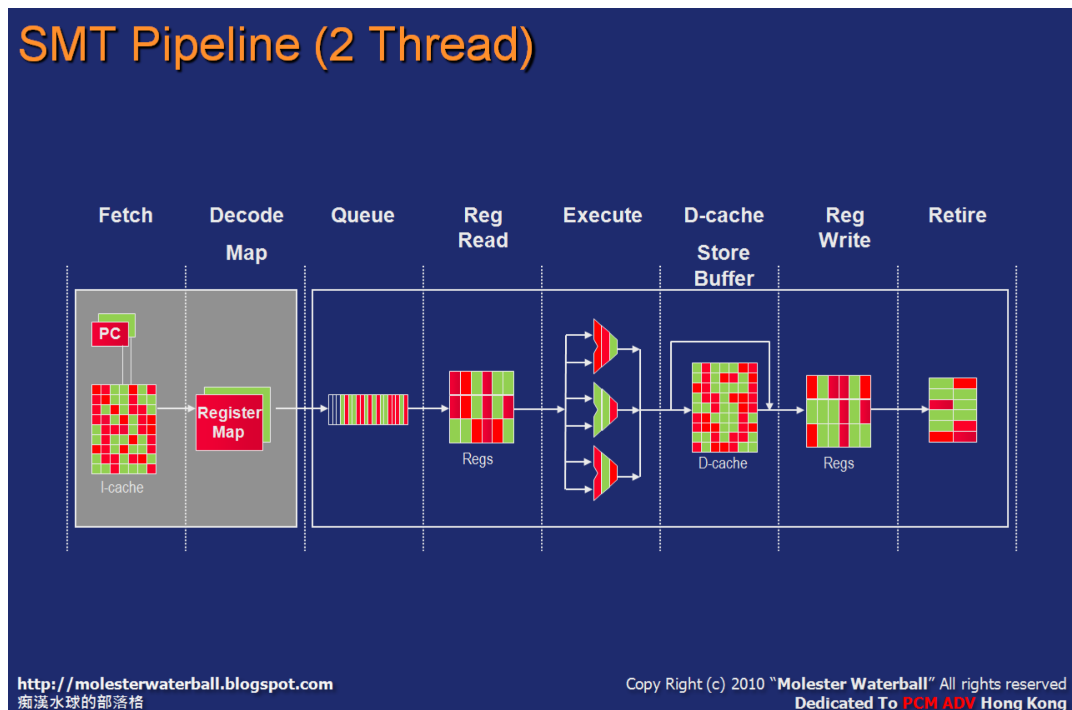


Figure 2.13: 2-Way SMT double execution pipeline. Source: Molester Waterball

2.3.7 A modern example: Intel Nehalem microachitecture

In the rest of this section, we will study a modern microprocessor, the Intel core i7, that implements all the techniques evoked in this section. Since all the cores are identical (SMP) we will concentrate on the details of the micro-achitecture of one core. Studying how the cores are actually interconnected is not of high importance. Figure 2.14 shows the overview of the Nehalem micro-architecture, which uses the X86-64 instruction set architecture (ISA).

From a macro-level point of view there are two main parts, namely the front-end and the back-end of the execution pipeline.

2.3.7.1 Front-end

The front-end part of the pipeline is responsible for fetching macro-instructions and breaking them into micro-operation(s) in order to feed the back-end. It actually converts a block of CISC macro-instructions into a set of RISC micro-operations. Basically, It contains three main stages, namely, instruction block fetch, instruction length decoding, instruction decode.

Instruction fetch unit (IFU) The front-end pipeline starts by the fetching of a block of instructions from the instruction cache (level 1). The bus feeding the instruction fetch unit is 128 Bit wide, which means it can fetch a word of 16 Bytes from the instruction cache. Coupled to the instruction fetch unit, the branch prediction unit (BPU) is responsible for starting the execution of an instruction flow based on predictions. It is the most effective way to avoid control pipeline hazard. The more the pipeline is long, the more its role is decisive. If a misprediction occurs, all the operations originating from the branch prediction unit are canceled.

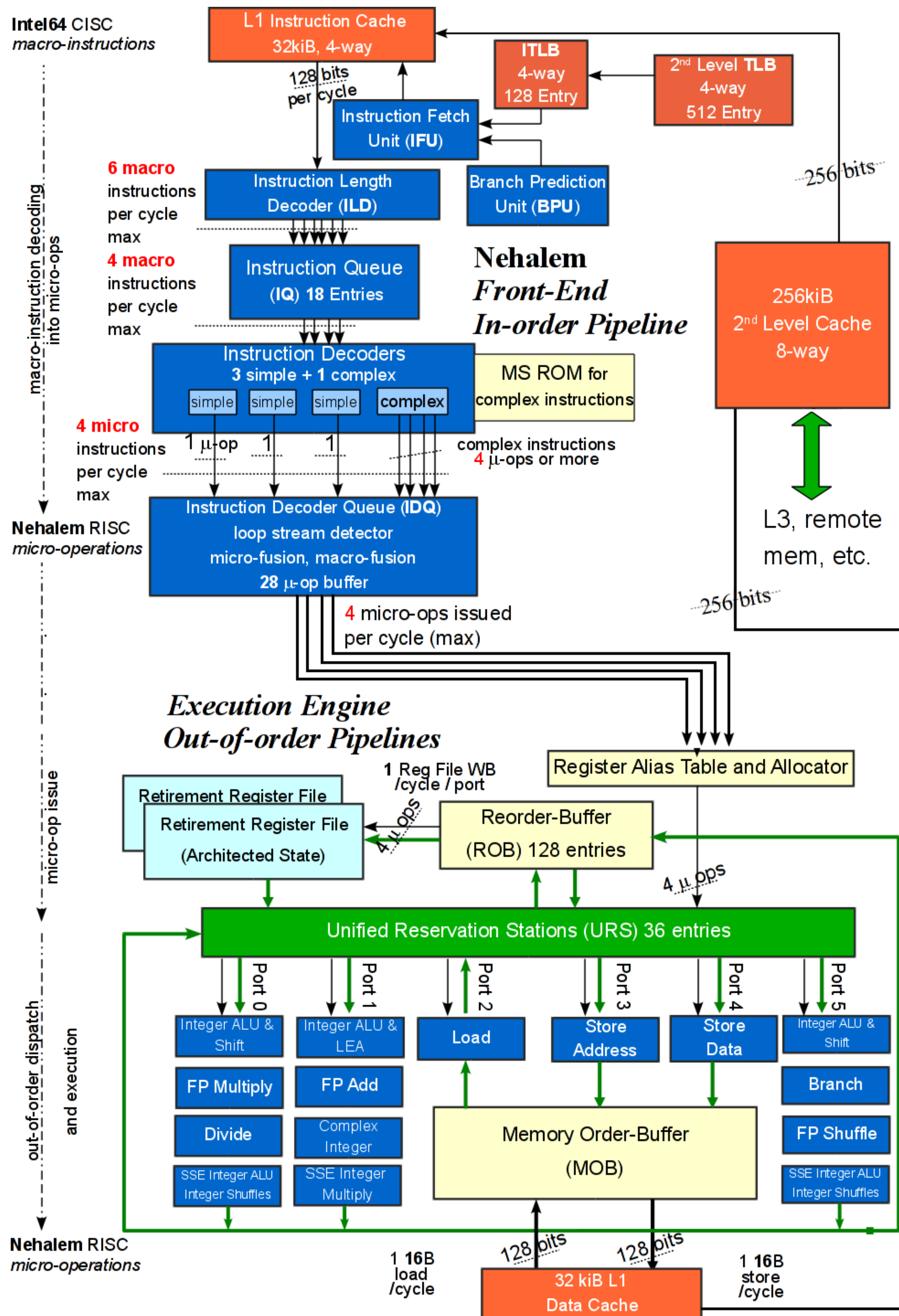


Figure 2.14: Overview of Intel Nehalem microarchitecture. Source: The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms by Dr. Thomadakis (Texas A&M)

Instruction length decoding Since the x86-64 ISA is of variable length, the 16 Bytes block fetched from the instruction cache must first be decomposed into macro

instructions. Up to six CISC macro-instructions can be recognized at once. Since the instruction length decoder can produce up to six macro-instructions which can only be processed by four decoders, an instruction queue is placed between both stages.

Instruction decode unit (IDU) Each macro-instruction is then converted into micro-operations thanks to the four available instructions decoders. Three of them are simple decoders whereas the fourth can process complex instructions. A micro-operation sequencer is used to convert those complex instructions into multiple micro-operations. Note that only four micro-operations can be produced each cycle tick. Apart of just mechanically translating CISC macro-instructions into RISC micro-operations, the IDU also applies transformations in order to achieve more efficient micro-operations. These transformations alre listed below:

- Loop stream detector: starting with the Nehalem micro-architecture, a loops stream detector (LSD) is present after the decoders stage. When a loop, which micro-operations feet in the LSD, is detected, the front-end can be switched off until a misprediction.
- Stack Pointer Tracking (SPT): In older micro-architectures stack manipulation instructions required multiple micro-operations. Thanks to STP, those instructions can be converted into a single micro-operation. It actually hides the hard work behind the scene by working directly on the concerned instructions.
- Micro-fusion: The IDU is able to fuse multiple micro-operations originating from the same instructions into a larger but single one, in order to reduce the number of micro-operations sent to the back-end.
- Macro-fusion: Consecutive instructions involving a test or compare instruction followed by a branch (jump) one can actually be fused into a single macro-instruction.

2.3.7.2 Back-end

The back-end part of the execution pipeline is responsible for the execution of the micro-operations produces by the front-end.

The execution engine includes the following major components:

Register Rename and Allocation Unit (RRAU) . Allocates resources in the execution engine for every micro-operation originating from the IDQ and passes them to the execution engine.

Reorder Buffer (ROB) . Tracks all micro-operations in-flight. It is an important stage because it allows newly allocation micro-operations to be executed even if older micro-operations are waiting for data.

Unified Reservation Station (URS) . It can queue up to 36 micro-operations until all source operands are ready. Then it schedules and dispatches up to six ready micro-operations to the available execution units.

Execution units . There are six execution units which cluster multiple functional units. Each execution unit is fully pipelined in order to maximize the utilization of the available functional units. Execution units can produce a result for most micro-ops with a latency of 1 cycle. Note that there is an associated operand forwarding network that facilitates the routing of results across the execution engine stages.

Memory Order Buffer (MOB) . It supports speculative and out of order loads and stores. Its main function is to ensure that writes to the memory subsystem take place in the right order, i.e. the macro-instructions order, and with the right data.

Retirement . When all the micro-operations of a macro-instruction are completed, retirement and write-back of results to "genuine" registers are performed. If a misprediction was detected, then nothing is written-back.

2.4 Leveraging parallelism

In the previous sections we described the complexity of modern architectures. Figure 2.15 presents the percentage of peak performance for four programs on four multiprocessors scaled to 64 processors. The NEC Earth simulator and Cray X1 are vector processors where as Power 4 and Itanium are superscalar. The former two delivered between 6% and 58% where as the latter two delivered only between 5% and 10% of the peak performance. As mentioned before, depending upon the nature (exposed parallelism) of a given application and the underlying architecture, the peak performance may be close or far. Indeed, an application optimized for vector processors will not obtain the same results on superscalar processors.

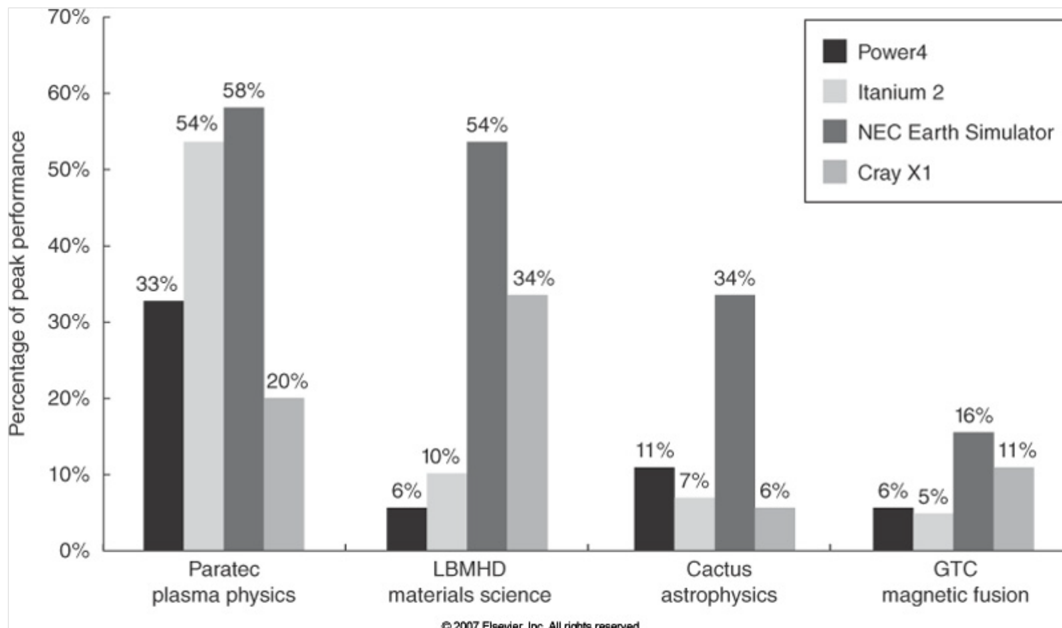


Figure 2.15: Percentage of peak performance for four programs on four multiprocessors scaled to 64 processors. Source: Hennessy, Patterson: Computer Architecture, page 58, 5th edition, Morgan Kaufmann

The current and future trends in the high performance computing field (Top 500) are illustrated by Figure 2.16. A tremendous horsepower is for the taking with the advent of the Exascacle Era. But we will need to be able the exploit the maximum level of parallelism, along with optimizations that can be performed at the core level, in order to harness such available performance.

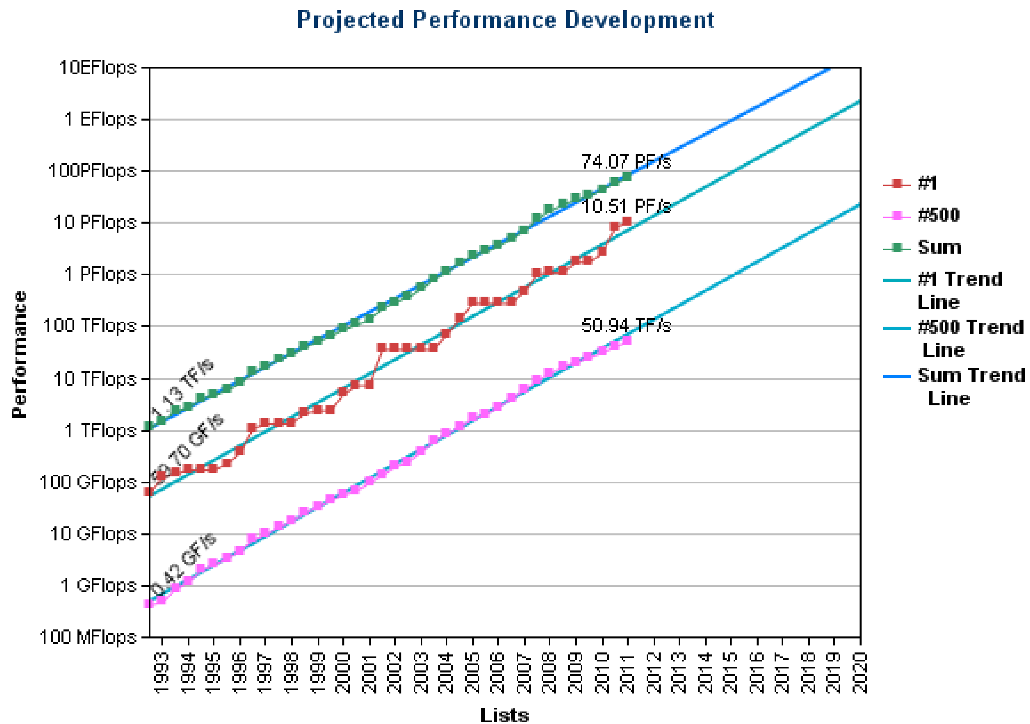


Figure 2.16: Current and Projected performance development of TOP 500 clusters. Source TOP500.org

In this section, we will first analyze the strengths and limitations of compilers, since it is the central tool to write applications. Then, we will discuss complementary aspects such as the importance of data layouts and the use of parallel programming paradigms to achieve a high level of parallelism. After that, a brief description of dynamic execution side effects that must be taken into account, will be presented.

2.4.1 Compiler strengths and limitations

Besides hand written codes in assembly language, compilers remain the main code generators. Given a set of source files, it produces a target application which is generally a binary file that may depend on companion or third party libraries.

Figure 2.17 gives an overview of the GCC compiler Architecture. Besides transforming the source code into an intermediate language and producing machine language code, the compiler applies optimizations in order to achieve a faster execution of an application. A compiler usually provides multiple flags to help or force some decisions. For instance, to further optimize performance on a specific CPU, it is possible to specify a precise target architecture.

However, using the highest level of optimization of a compiler does not guarantee

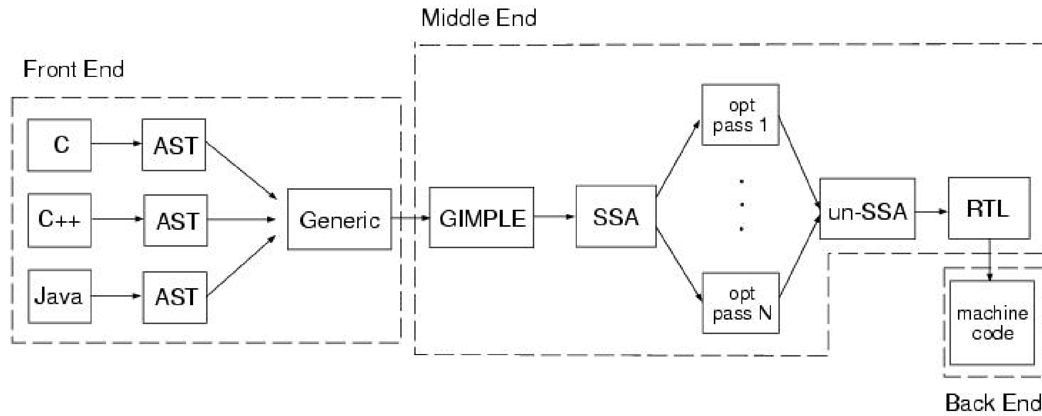


Figure 2.17: Overview of GCC Architecture: Source: GNU

the best performing code. Actually, it may even lead to slower code compared to lower level optimizations. This is due to the fact that the compiler applies a series of optimizations in a certain order that is not proven to be optimal. It have been proven that the decidability of the phase ordering problem in optimizing compilation is undecidable [114]. Even more localized optimizations like register allocation actually use heuristics. Register allocation can be reduced to the problem of K-coloring the associated interference graph, where K is the number of registers available on the target architecture. As graph coloring in general is NP-complete, so is register allocation. Hence using a heuristic. Practically, it means that some registers, which is a rare resource in most architectures, may remain unused because of a suboptimal solution at the regiter allocation stage.

Even if the compiler can perform multiple optimizations, many limitations appears because it is actually constrained by the way a programmer structures his code. Many optimizations will fail because of the presence of dependencies, unknown values at compile time, the way data structures are accessed. Furthermore, exploiting the recent complex architecture implies exposing more parallelism. Once again, the programmer is responsible for this part. As a consequence compilers tends to integrate parallel primitives, that are processed at compile time, to better exploit the huge amount of available performance power.

2.4.2 Importance of data layouts

Inefficient data layout structures have an impact on memory. Since we saw the growing gap between memory accesses and computations, one of the huge challenges to face is detecting and fixing such issues. There are two main methods to reduce or even get rid of this class of problem. The first one is by letting the compiler take care of it in a parallel environment, hence carving up data blocks among multiple parallel units (processes or threads). However it is not sufficient in a parallel environment. Especially in a shared memory system, an extra care have to be adopted because of thread interaction provoking cache issues. The second method is to resort to techniques like the cache-oblivious model in which data structures and algorithms are aware of the existence of a multi-level memory hierarchy but do not assume any knowledge about the parameter values of the hierarchy, such as the number of levels in the hierarchy, the capacity and the block size of each level. However, it is not

easy to find the correct granularity because deep recursions may hide the benefit of processing smaller problems.

2.4.3 Parallel programming paradigms

In order to exploit performance brought by the combination of multiple processors or cores, parallel programming paradigms are required. Writing sequential applications that may communicate with each other in order to process smaller part of a bigger problem is no more sufficient. Parallel programming paradigms are actually tightly related to the underlaying parallel architecture models. There are two main models, namely shared and distributed architectures. Some programming models may use a combination of both. For instance, Partitionned Golbal Address Space (PGAS) model is a mixture between shared and distributed models. Quoting the authors, "UPC combines the programmability advantages of the shared memory programming paradigm and the control over data layout and performance of the message passing programming paradigm".

2.4.3.1 Shared architectures

Shared architectures are typically composed of SMP processors sharing the same address space. Different levels of parallelism exists with modern SMPs. We already evoked $instruction - level\ parallelism$. The higher level of parallelism is $thread - level\ parallelism$ which can be decomposed into $data - level\ parallelism$ and $task\ or\ request - level\ parallelism$.

data level parallelism $Data - level\ parallelism$ focuses on distributing the data across different parallel computing nodes. It is also know as loop-level parallelism because it is usually used to partition chunks of the iteration space of a loop across multiple threads.

task level parallelism $Task - level\ parallelism$ focuses on distributing execution threads across different parallel computing nodes. It is also know as function parallelism because of its broader action field.

The OpenMP Application Program Interface [11] supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved. Examples of other libraries based on shared memory programming exists, such as TBB [49], CILK Plus [44], PTHREADS [85].

Shared architectures are generally considered as small systems since few multiprocessors may live together on the same physical board. Distributed Shared architectures extends "simple" shared architectures to the next level. Given an interconnect protocol that maintains a single system image, all processors can reach the same address space.

2.4.3.2 Distributed architectures

Distributed architectures are typically composed of multiple shared architectures connected through fast interconnect links, each having its own system image. Thus,

a distributed system may have several autonomous computational entities, each of which has its own local memory and the entities communicate with each other by message passing. MPI [30] is the current most used message passing interface. It is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. Examples of other libraries based on distributed memory programming exists, such as Charm++ [87], Co-array Fortran [86], Unified Parallel C [56], STAPL [116].

In general, since distributed architectures are based on shared architectures, the best way to exploit parallelism is to build hybrid applications harnessing the power of each models.

Some programming models like Intel Concurrent Collection [46] and Array Building Blocks [43] for C++ can be used on both shared and distributed memory, thanks to their level of abstraction. Since the shared memory model can be considered as a special case of the distributed memory model, parallel programming paradigms working on distributed architecture will work shared memory architectures. Note that for instance MPI applications can run on shared memory architectures but most of the time will not be as efficient as shared memory based programming models.

2.4.4 Dynamic execution side effects

The overall execution time of an application is not only attributable to the compiler work. There are two other sources of variability of the overall execution time, namely, operating system and architecture levels.

OS level. Depending upon the type of memory allocator used at OS level, memory fragmentation may reduce the performance of memory requests. Several conventional dynamic storage allocators provide near-zero fragmentation [55], once we account for overheads due to implementation details such as headers, alignment, etc. Page migration [121] can also affect performances.

While having a low impact on sequential applications, thread migration at OS-level can degrade the performance of parallel programs. For instance, not fixing an affinity (pinning) when running OpenMP applications can dramatically lower the overall performance[72]. For example, the Intel OpenMP runtime proposes two predefined affinity heuristics, compact and scatter, which achieve better performances compared to the system default.

Architecture level. Architectures do not always behave as expected. This is mainly due to a vast number of hardware mechanism that are supposed to be optimizations and which can be counterproductive. For instance, on Intel processors, hardware prefetching may consume a lot of memory traffic that is actually not relevant. Other optimizations like store-forwarding or 4k-aliasing may introduce additional latencies when failing[48].

Dynamic execution side effects varies from one architecture, and OS, to the other. As a consequence, a special care should be taken to integrate this dimension into the performance tuning process, when analyzing performance issues.

2.5 Performance analysis approaches

Due to compiler limitations, evoked earlier in this section, performance tuning is necessary in order to pinpoint issues related to the dynamic behavior of a given application. Static analyses are mostly used to perform predictions, verify the code quality generated by the compiler, or are part of a broader process including dynamic analysis, i.e. post-mortem analyses. In this section we will present the three main categories of approaches to analyze the performance of applications, namely, modeling, simulation and measurement. The latter will be further detailed since it is the main approach used in the contributions of this thesis.

2.5.1 Modeling

Modeling is mainly used to predict performance metrics based on a performance model and validate measurements or simulations. The main advantages of modeling is that it can be easily adapted, when considering additional parameters for a system, and has a lower setup cost compared to the other approaches. However it is not very accurate. There is always a tradeoff between accuracy and required time to obtain a solution. For instance, modeling can be used for cache metrics prediction with examples like Statstack [34], a modeling for LRU cache or StackCC [33], a model for estimating the contention for shared cache resources between co-scheduled applications on chip multiprocessor architectures.

2.5.2 Simulation

Simulation is either used to mimic existing architectures or components, or test future ones. Processor manufacturers tend to integrate an increasing number of cores in a single chip (CMPs). Designing and developing these CMP architectures involves studying and testing several options for on-die interconnect, cache and memory system while optimizing for both power and performance. Simulation-based study is widely adopted for the design space exploration for these systems. When considering performance analysis, the goal is to mimic existing architectures or parts of it. One famous example is the CPU emulator Valgrind [83]. We can also mention cachegrind [82] which targets the caches structures. Another example of simulation is Intel Architecture Code Analyzer [41] which simulates de execution pipeline in order to provide static estimations of throughput and latency, under ideal front-end, out-of-order engine and memory hierarchy conditions.

Simulation remains a flexible model that can integrate new parameters and have better accuracy than mathematical modeling. The counterpart is a higher investment cost in development for an accuracy that still needs to be refined.

2.5.3 Measurement

Measuring metrics or more generally the behavior of an application is the most precise way to pinpoint issues. However it is difficult and time consuming compared to the previous approaches. The difficulty comes from the fact that there are different levels and methods of measurement. Each combination will only be relevant for specific cases. It will also highlights the tradeoff between accuracy and overhead.

2.5.3.1 Instrumentation

Instrumentation consists in inserting probes that will execute routines which goal is to capture information. It is usually applied at multiple levels and associated to semantic structures of applications like functions, loops, etc. There are two types of instrumentations, namely, static and dynamic.

Static instrumentation Static instrumentation takes place before the execution of the application. The main advantage of static instrumentation is that it lowers the overhead induced by instrumentation since it is done offline. Obviously, the associated drawback is the lack of flexibility at runtime.

Dynamic instrumentation Dynamic instrumentation is performed at runtime. The main advantage of dynamic instrumentation is the ability to dynamically insert or remove probes. However, it increases the overhead cost.

The volume of data gathered and the overhead of instrumentation will depend upon the considered measurement method and granularity. Besides using hardware performance counters, we also must recall that instrumentation is an intrusive process since it executes additional code that may change the behavior of the target application depending upon the level of instrumentation.

2.5.3.2 Levels

There are multiple levels where measurements can take place. The lowest level is the binary level. Measurement is performed on the real code that will be executed by the target architecture. Using tools that reconstructs abstract structures is needed in order to identify what we want to measure and where. In fact, we will also need to establish a link between the binary level and the source level. Just after the binary level, measurement can also be done at compiler level. A famous example is gprof [92] which output is activated when using special flags of the compiler (usually -pg). The next level, is the source code level. Calls to external libraries can be explicitly added around targets of interest. Finally, the OS is the highest level if we don't consider virtual machines. For example, the *LDPRELOAD* mechanism, in conjunction with companion libraries, enables to intercepts calls to functions, hence being able to modify or perform actions while the application is running. From the highest to the lower level, is possible to measure more accurate events but with a growing complexity. For instance, OS-level is a cheaper solution to time a predefined list of functions than working at the binary level. However, if more precise measurements needs to be done, source of binary levels would be more suitable.

2.5.3.3 Methods

After having selected a level and an instrumentation type of measurement, a method must be chosen. The latter will depend upon the requirements and constraints raised by the target application. There are three available methods of instrumentation, namely, profiling, tracing and sampling.

Profiling A profile provides an inventory of performance events and timings for the execution as a whole. This ignores the chronology of the events in an absolute

sense. Nothing is timestamped and the resulting report does not say what events happened before other events in a absolute sense. Relative ordering of events may be recorded in a profile. A profile is often sufficient to pinpoint load imbalance due to problem decomposition and/or identify the origin of excessive communication time.

Tracing A trace records the chronology, often with timestamps and is extensive in time. The amount of data in the trace increases with the runtime. As such in order to bound the memory usage by the tracing one must periodically write the data out to disk or network. A trace is useful for detailed examination of timing issues occurring within a code.

Sampling Sampling defines a periodic statistical data gathering which involves less precision. Obviously, the main advantage is the lower overhead compared to the other methods. It is usually used for coarse grain analyses where a high degree of accurary is not required.

All of these methods provide a different tradeoff between accuracy and instrumentation overhead. In the next section, we will discover the existing performance analysis tools based on different instrumentation methods at different levels.

2.6 Performance analysis tools

In order to lookup for performance issues, performance evaluation tools are required. Given the possible number of sources of issues, a unique tool that addresses all the possible issues seems difficult, and even impossible to conceive. Practically, each tool try to address specific issues. Some tools tackle the same issue, but using different methods presented in the previous section. The aim of this section is not list all the existing tools but rather the most significant ones that tries to address the issues evoked at the beginning of the Chapter, related to the complexity of multi-core processors and the associated memory subsystem. Most tools rely on the same building block components (tools, frameworks) but develops different analyses. In order to methodically study these tools, we will first present the building block components before presenting the performance evaluation tools built on top of them. Finally, we will provide a summary containing all the tools and their main features.

We arranged the building block tools based on the following categorization:

- Binary instrumentation (Static or Dynamic)
- Compiler based (Source to source or Embedded)
- Hardware Performance counters

The mixed and toolkits category is actually a way to present tools not falling clearly in any other category because mixing more than one category or grouping several tools.

2.6.1 Binary instrumentation

2.6.1.1 Pin

Pin [118] is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. Some tools built with Pin are Intel Parallel Inspector, Intel Parallel Amplifier and Intel Parallel Advisor. The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications in Linux and Windows. As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code. Pin provides a rich API that abstracts away the underlying instruction-set idiosyncrasies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues to work. Limited access to symbol and debug information is available as well. Pin was originally created as a tool for computer architecture analysis, but its flexible API and an active community (called "Pinheads") have created a diverse set of tools for security, emulation and parallel program analysis.

2.6.1.2 DynInst

Dyninst [14] is program manipulation tool which provides a C++ class library for program instrumentation. It is an Application Program Interface (API) for runtime code patching. Using this library, it is possible to instrument and modify application programs during execution. This library permits machine-independent binary instrumentation programs to be written. DynInst API is partr of the Paradyn [75] project.

2.6.1.3 DPCL

The Dynamic Probe Class Library [27] (DPCL) is a value-added layer built on top of the Dyninst API. DPCL targets the the following goals:

- Provide an infrastructure that allows tools to probe single process applications up to very large MPI applications running across 4096 processors.
- Provide a secure infrastructure.
- Provide thread safe instrumentation to support threaded processes.
- Provide multiple probe types to support a variety of tools.
- Provide the ability for probes to communicate back to the tool.
- Provide support for C, C++, and Fortran applications.

While Dyninst provides a good substrate for tools development, its focus is on modifying a set of running processes on a single machine. The DPCL layer adds additional support for multiple nodes in a parallel machine, a security layer to authenticate inter-node instrumentation requests, and a data transport layer to gather data from nodes in a parallel computer and provide them to a single front-end

process. While Dyninst works on multi-processor machines since it uses the native platform's debugger interface (ptrace or procfs) it does not allow requests to cross between nodes in a parallel computer. Likewise, it lacks support for identifying the processes from separate nodes that are part of a parallel job. DPCL provides features that allow it to interact with the Parallel Operating Environment (POE) on IBM SP systems to identify the processes of a parallel job (the user simply needs to identify the process id the front-end processes associated with the job). Once it has this information, it starts instrumentation processes on each node that is running a process from the parallel job. There are also differences in the security models between Dyninst and DPCL. Dyninst relies on the host operating system's security for ptrace and procfs system calls. These calls restrict debuggers to only be attached to processes owned by the same user as the one running the debugger. However, with DPCL's multi-node feature, it is necessary to add additional security. DPCL extends the security model provided by Dyninst by extending security requirements across SP system or cluster.

2.6.2 Compiler

OPARI OPARI [77] is a source-to-source translation tool based on directive instrumentation transformations. More precisely, it is a source-level instrumentation approach based on OpenMP directive rewriting. The main goal is to provide an interface for OpenMP to performance tools, similar in spirit to the MPI profiling interface in its intent to define a clear and portable API that makes OpenMP execution events visible to runtime performance tools. Rules to instrument each directive and their combination are applied to generate calls to the interface consistent with directive semantics and to pass context information (e.g., source code locations) in a portable and efficient way. The proposed OpenMP performance API further allows user functions and arbitrary code regions to be marked and performance measurement to be controlled using new OpenMP directives.

gprof gprof [92, 73] profiling tool processes results obtained thanks to compiler instrumentation. Application must be compiled with the '-pg' option. The tool itself will be presented later on in this section.

2.6.3 Architecture simulation

Valgrind Valgrind [83] is a CPU emulator which provides an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. The Valgrind distribution includes a set of profiling tools

2.6.4 Introduction to Hardware Performance counters

Modern processors already provide the capability to monitor performance events inside processors. In order to obtain a more precise picture of CPU resource utilization, dynamic data is obtained from a specialized unit inside the processor. For instance, Intel processors provides the so-called performance monitoring units (PMU). When using performance tuning tools based on hardware performance counters, or providing access to them, users must be aware of the strength and particularly the

limitations of this approach or measurement. We will present below the key aspects to be aware of when dealing with hardware performance counters.

Strengths. Since counting is handled by the hardware, this approach introduces only very few overhead and is non-intrusive. Also, some low-level information may only be available through hardware performance counters.

Event count. It is possible to select among more than a thousand of events. Some event names are very obscure, even their textual description helps much. Moreover, documentation is usually quite poor. Besides common events (number of misses in the different levels of cache, elapsed cycles, retired instructions), a deep understanding of the architecture is needed to understand and select other counters.

Types of counting. One the one hand, event count refers to counting the number of times an event has happened. On the other hand, event duration refers to counting cycles as long as an event condition is verified. Finding the right sampling rate is not obvious. Actually, the key requirement in choosing sampling periods is that it is necessary to obtain enough samples to provide statistical significance. Documentation may sometime recommend a sampling rate for specific counters, but it will depend upon the nature of the application. In other words, a sampling rate providing enough samples for a given application may not be adapted for another one.

For instance, to perform timer-based profiling such as gprof, counters are programmed to generate an interrupt on a given event count (e.g. 1000 events) and will attribute the whole event count to the instruction that was running at that time.

Incompatible events. Some events cannot be obtained in the same run and will require multiple runs of a given application. For instance, retrieving the number of misses in level 1 and level 2 caches is not possible at the same time. Actually, some events may map to the same physical count register.

Multiplexing There is a limited number of physical counters. Some hardware performance counters measurement tools provide the ability to measure more events than available counters. They actually resort to multiplexing in order to achieve their gaol. But it is actually an illusion. Thus, the more multiplexing will be used , the less results will be accurate.

System level modules and applications There are several available module which allows access to the performance monitoring units. Figure 2.18 provides an overview of the existing modules, APIs and some applications using them.

The overview is split into three layers, namely, hardware, kernel and user. Only kernel module allows access to the hardware. Then "userland" libraries or applications can access the kernel module's interface.

2.6.4.1 PAPI

The Performance API[79] (PAPI) project specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count

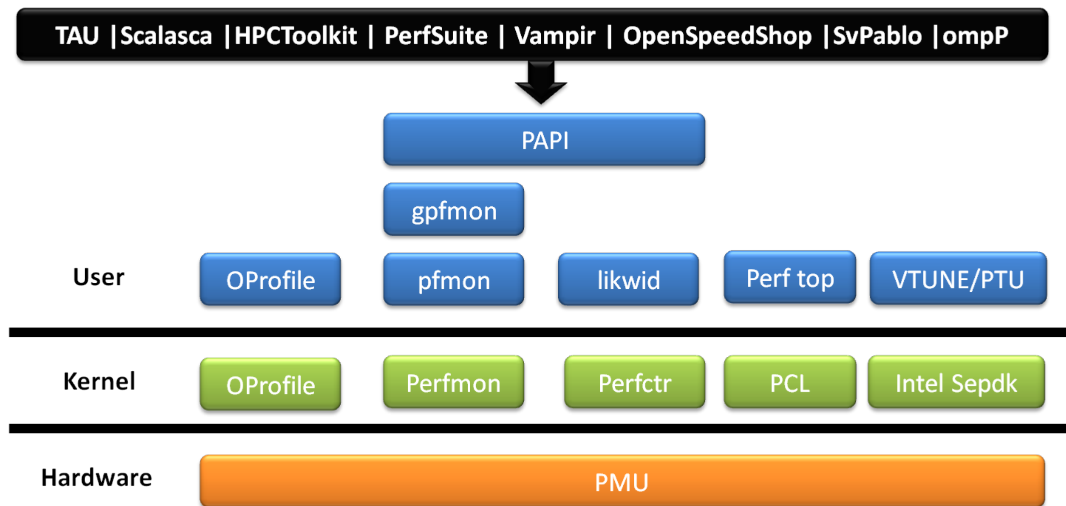


Figure 2.18: An overview of the existing hardware performance counters ecosystems

Events, occurrences of specific signals related to the processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. This correlation has a variety of uses in performance analysis including hand tuning, compiler optimization, debugging, benchmarking, monitoring and performance modeling. In addition, it is hoped that this information will prove useful in the development of new compilation technology as well as in steering architectural development towards alleviating commonly occurring bottlenecks in high performance computing.

PAPI provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level PAPI interface deals with hardware events in groups called EventSets. EventSets reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level 1 cache misses can indicate poor locality and memory management. In addition, EventSets allow a highly efficient implementation which translates to more detailed and accurate measurements. EventSets are fully programmable and have features such as guaranteed thread safety, writing of counter values, multiplexing and notification on threshold crossing, as well as processor specific features. The high level interface simply provides the ability to start, stop and read specific events, one at a time.

2.6.4.2 Likwid

LIKWID [115] is a set of command-line utilities that addresses four key problems: probing the thread and cache topology of a shared-memory node, enforcing thread-core affinity on a program, measuring performance counter metrics, and toggling hardware prefetchers. An API for using the performance counting features from user code is also provided. The provided command lines are the following:

- LikwidTopology: A tool to display the thread and cache topology on multi-

core/multisocket computers

- LikwidPerfCtr: A tool to measure hardware performance counters on recent Intel and AMD processors. It can be used as wrapper application without modifying the profiled code or with a marker API to measure only parts of the code.

- LikwidFeatures: A tool to toggle the prefetchers on Core 2 processors.

- LikwidPin: A tool to pin your threaded application without changing your code. Works for pthreads and OpenMP.

2.6.5 Tools

2.6.5.1 gprof

gprof [92, 73] is a profiling tool that accounts for the running time of called routines in the running time of the routines that call them. Profiling works by changing how every function of a given application is compiled so that when it is called, it will stash away some information about where it was called from. From this, the profiler can figure out what function called it, and can count how many times it was called. This change is made by the compiler when the application is compiled with the '-pg' option, which causes every function to call mcount (or _mcount, or __mcount, depending on the OS and compiler) as one of its first operations. The instrumentation method uses by gprof is sampling, in particular, it is kernel sampling. Basically, a call to the mcount function tells the system kernel to gather information about the execution of the function from which it is called from.

gprof application is practically used after the application execution. It can display either a flat profile report or a call graph.

Figure 2.19 shows an output example of flat profile.

The functions are sorted first by decreasing run-time spent in them, then by decreasing number of calls, then alphabetically by name. Here is what the fields in each line mean:

- % time: this is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

- cumulative seconds: this is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

- self seconds: this is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

- calls: this is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the calls field is blank.

- self ms/call: this represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

Figure 2.19: Example of gprof flat profile output

- total ms/call: this represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function. This is the only field in the flat profile that uses call graph analysis.
- name: this is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds and calls fields are sorted.

Figure 2.20 presents an output example of callgraph.

The lines full of dashes divide this table into entries, one for each function. Each entry has one or more lines. In each entry, the primary line is the one that starts with an index number in square brackets. The end of this line says which function the entry is for. The preceding lines in the entry describe the callers of this function and the following lines describe its subroutines (also called children when we speak of the call graph). The entries are sorted by time spent in the function and its subroutines.

2.6.5.2 Cachegrind/Callgrind

Cachegrind Cachegrind is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code. It identifies the number of cache misses, memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries. It is useful with programs written in any language. Cachegrind runs programs about 20–100x slower than normal. Figure 2.20 presents an output example of callgraph.

Cachegrind gathers the following statistics (abbreviations used for each statistic is given in parentheses):

```
index % time    self  children    called     name
                                              <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05      1/1           main [2]
                0.00    0.00      1/2           on_exit [28]
                0.00    0.00      1/1           exit [59]
-----------------------------------------------
                0.00    0.05      1/1           start [1]
[2]    100.0    0.00    0.05      1             main [2]
                0.00    0.05      1/1           report [3]
-----------------------------------------------
                0.00    0.05      1/1        main [2]
[3]    100.0    0.00    0.05      1             report [3]
                0.00    0.03      8/8           timelocal [6]
                0.00    0.01      1/1           print [9]
                0.00    0.01      9/9           fgets [12]
                0.00    0.00     12/34          strncmp <cycle 1> [40]
                0.00    0.00      8/8           lookup [20]
                0.00    0.00      1/1           fopen [21]
                0.00    0.00      8/8           chewtime [24]
                0.00    0.00      8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01    0.02      8+472         <cycle 2 as a whole> [4]
                0.01    0.02    244+260           offtime <cycle 2> [7]
                0.00    0.00    236+1             tzset <cycle 2> [26]
-----------------------------------------------
```

Figure 2.20: Example of gprof callgraph output

```
--------------------------------------------------------------------------------
Ir        I1mr ILmr Dr        D1mr  DLmr  Dw        D1mw   DLmw    file:function
--------------------------------------------------------------------------------
8,821,482    5    5 2,242,702 1,621    73 1,794,230      0      0  getc.c:_IO_getc
5,222,023    4    4 2,276,334    16    12   875,959      1      1  concord.c:get_word
2,649,248    2    2 1,344,810 7,326 1,385         .      .      .  vg_main.c:strcmp
2,521,927    2    2   591,215     0     0   179,398      0      0  concord.c:hash
2,242,740    2    2 1,046,612   568    22   448,548      0      0  ctype.c:tolower
1,496,937    4    4   630,874 9,000 1,400   279,388      0      0  concord.c:insert
  897,991   51   51   897,831    95    30        62      1      1  ???:???
  598,068    1    1   299,034     0     0   149,517      0      0  ../sysdeps/generic/...
  598,068    0    0   299,034     0     0   149,517      0      0  ../sysdeps/generic/...
  598,024    4    4   213,580    35    16   149,506      0      0  vg_clientmalloc.c:malloc
  446,587    1    1   215,973 2,167   430   129,948 14,057 13,957  concord.c:add_existing
  341,760    2    2   128,160     0     0   128,160      0      0  vg_clientmalloc.c
  320,782    4    4   150,711   276     0    56,027     53     53  concord.c:init_hash_table
  298,998    1    1   106,785     0     0    64,071      1      1  concord.c:create
  149,518    0    0   149,516     0     0         1      0      0  ???:tolower@@GLIBC_2.0
  149,518    0    0   149,516     0     0         1      0      0  ???:fgetc@@GLIBC_2.0
   95,983    4    4    38,031     0     0    34,409  3,152  3,150  concord.c:new_word_node
   85,440    0    0    42,720     0     0    21,360      0      0  vg_clientmalloc.c
```

Figure 2.21: Example of cachegrind output: function-by-function statistics

- I cache reads (Ir, which equals the number of instructions executed), I1 cache read misses (I1mr) and LL cache instruction read misses (ILmr).

- D cache reads (Dr, which equals the number of memory reads), D1 cache read misses (D1mr), and LL cache data read misses (DLmr).

- D cache writes (Dw, which equals the number of memory writes), D1 cache write misses (D1mw), and LL cache data write misses (DLmw).

Each function is identified by a file_name:function_name pair. If a column contains only a dot it means the function never performs that event (e.g. the third row shows that strcmp() contains no instructions that write to memory). The name ??? is used if the file name and/or function name could not be determined from debugging information. If most of the entries have the form ???:??? the program probably wasn't compiled with -g.

Callgrind Callgrind [119, 120] is an extension to Cachegrind. It provides all the information that Cachegrind does, plus extra information about callgraphs. Also available separately is a visualisation tool, KCachegrind, which gives a much better overview of the data that Callgrind collects; it can also be used to visualise Cachegrind's output. Figure 2.22 presents coverage lists and a call tree graph visualisation. Figure 2.23 presents the three available views, namely, the Cost Type View (top left), the Call Graph View (on the right) and the Callee Map View (bottom left).

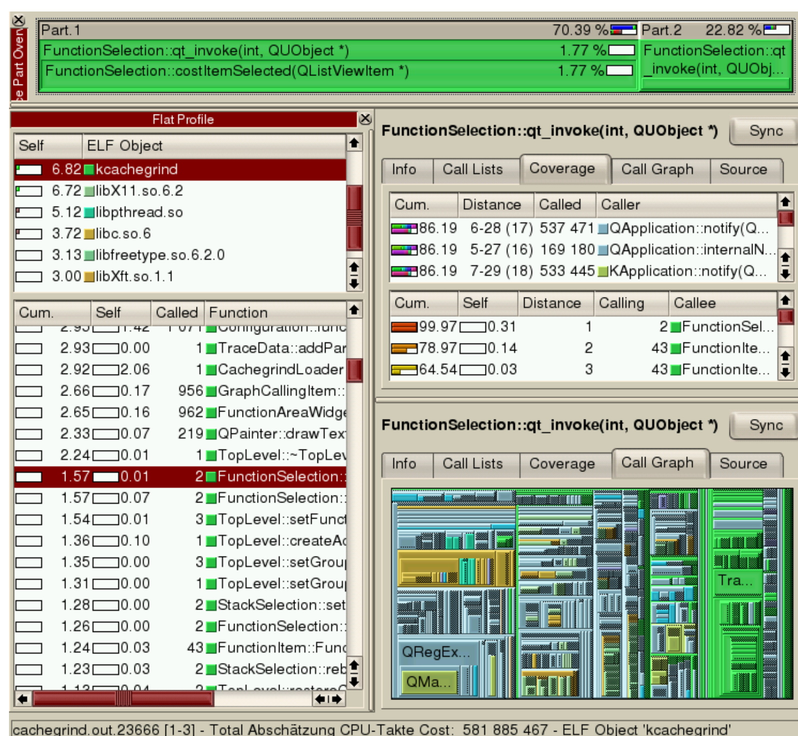


Figure 2.22: Coverage lists and call tree graph visualisation. Source: http://kcachegrind.sourceforge.net/html/Screenshots.html

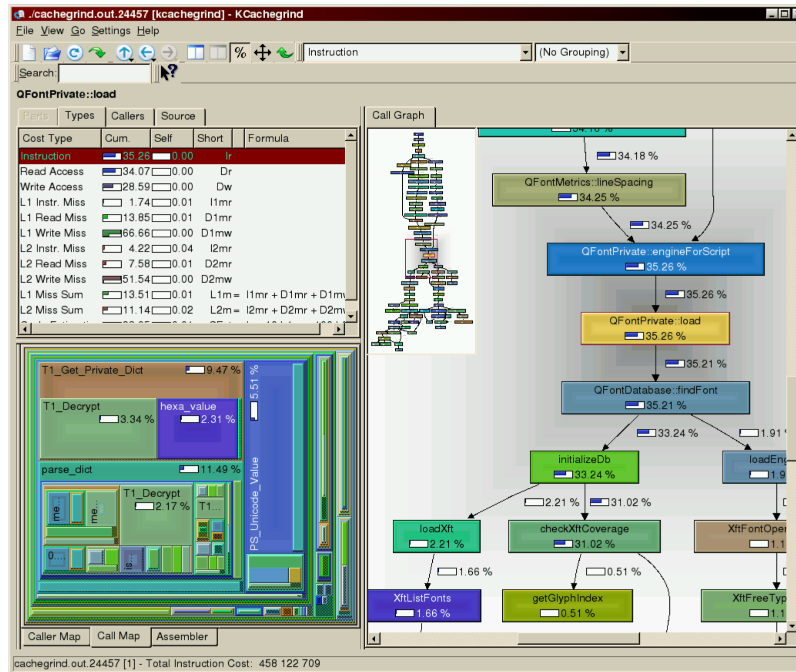


Figure 2.23: On the right the call graph view. Source: http://kcachegrind.sourceforge.net/html/Screenshots.html

2.6.5.3 CMP$IM

CMP$im [51] is a Pin-based memory system simulator. Pin only supports instrumenting a single application. However, CMP$im already supports multi-threaded applications and was extended enable multi-programmed simulation. The cache hierarchy is created in shared memory (using System V or memory mapped I/O) and requires multiple instances of Pin. CMP$im needs to connect to the shared memory (see 2.24). Identical virtual addresses between the different applications are distinguished by comparing the application ID along with the virtual address. Once the required number of applications connects to the shared memory, cache simulation proceeds normally.

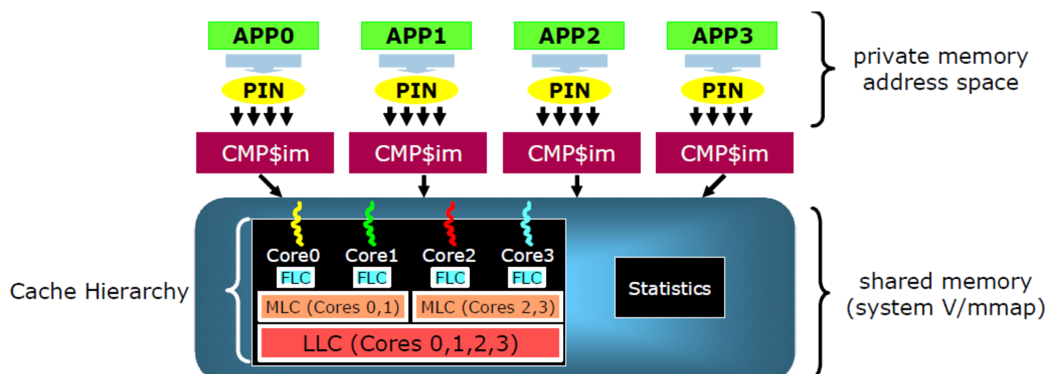


Figure 2.24: Multi-Programmed CMP$im Implementation Overview. Source [?]

CMP$im gathers statistics such as the total number of cache accesses and misses, sharing characteristics of multi-threaded applications, coherence traffic. All statis-

tics are output to a data file when the program finishes execution. Alternatively, to characterize the time varying behavior of the application, statistics can also be logged periodically to the output file. This enables users to visualize the time varying behavior of an application over the course of simulation and helps identify representative regions of execution for detailed simulation.

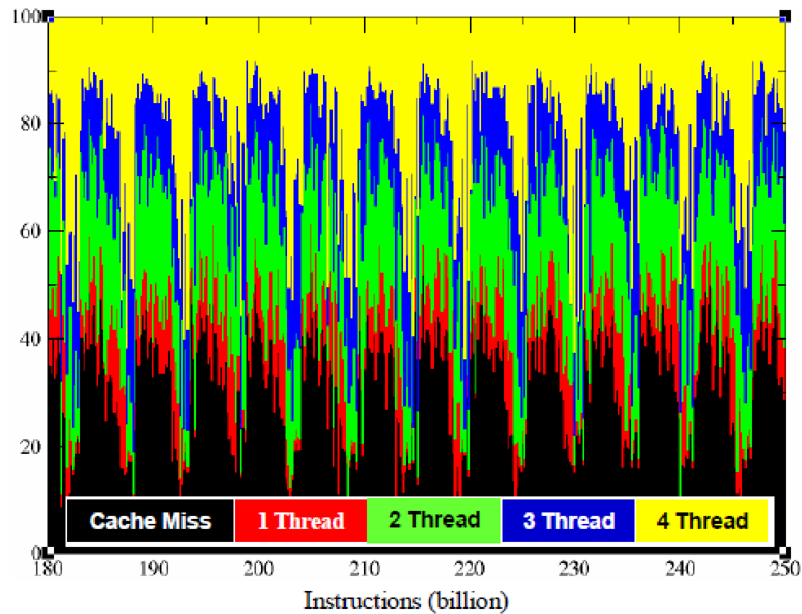


Figure 2.25: Cache Performance and Sharing Characteristics of a Multi-Threaded Workload using CMP$im. Source [?]

Figure 2.25 shows an example of statistics of the last-level cache accesses. We can see that 50-80% of the last-level cache accesses are to lines that are shared by two or more cores. With the significant amount of data sharing between multiple cores, the measured application performs better with a shared last-level cache.

2.6.5.4 Intel VTune

Current version (while writing these lines) is Amplifier XE [22] (formerly Intel VTune Performance Analyzer with Intel Thread Profiler). It can perform various kinds of code profiling including stack sampling, thread profiling and hardware event sampling.

The profiler result consists of details such as time spent in each sub routine which can be drilled down to the instruction level. The time taken by the instructions are indicative of any stalls in the pipeline during instruction execution. The tool can be also used to analyze thread performance. The new GUI can filter data based on a selection in the timeline.

Intel PTU Intel PTU is an experimental performance analysis tool to test new technology before it becomes a product. All New Intel VTune Amplifier XE now includes many Intel PTU features.

2.6.5.5 TAU

The TAU (Tuning and Analysis Utilities) parallel performance system is a framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications.

TAU supports an instrumentation model that allows the user to insert performance instrumentation calling the TAU measurement API at different, multiple levels of program code representation, transformation, compilation, and execution. Supported levels of instrumentation are:

- Source-Based Instrumentation TAU provides an API that allows programmers to manually annotate the source code of the program. Source level instrumentation can be placed at any point in the program and it allows a direct association between language and program-level semantics and performance measurements. Using cross-language bindings, TAU provides its API in C++, C, Fortran, Java, and Python languages.

- Preprocessor-Based Instrumentation The source code of a program can be altered by a preprocessor before it is compiled. Preprocessor-based instrumentation is also commonly used to insert performance measurement calls at interval entry and exit points in the source code. To support automatic performance instrumentation at the source level, the TAU project has developed the Program Database Toolkit (PDT). The purpose of PDT, is to parse the application source code and locate the semantic constructs to be instrumented.

- Compiler-Based Instrumentation A compiler can add instrumentation calls in the object code that it generates.

- Wrapper Library-Based Instrumentation Considering MPI, TAU has a MPI wrapper library that intercepts calls to the native library by defining routines with the same name, such as MPI_Send. These routines then call the native library routines with the name shifted routines, such as PMPI_Send. Wrapped around the call, before and after, is TAU performance instrumentation.

- Binary Instrumentation TAU uses DyninstAPI [14] for instrumenting the executable code of a program.

- Interpreter-Based Instrumentation TAU has been integrated with Python by leveraging the Python interpreter's debugging and profiling capabilities to instrument all entry and exit calls. By including the tau package and passing the top level routine as a parameter to the tau package's run method, all Python routines invoked subsequently are instrumented automatically at runtime. A TAU interval event is created when a call is dispatched for the first time. At routine entry and exit points, TAU's Python API is invoked to start and stop the interval events. TAU's measurement library is loaded by the interpreter at runtime. Since shared objects are used in Python, instrumentation from multiple levels see the same runtime performance data.

- Virtual Machine-Based Instrumentation Support of performance instrumentation and measurement in language systems based on virtual machine (VM) execution. TAU uses the Java Virtual Machine Tool Interface (JVMTI) to provide this support for Java Virtual Machines (JVM).

TAU allows multiple instrumentation interfaces to be deployed concurrently. For better coverage. It taps into performance data from multiple levels and presents it in a consistent and a uniform manner by integrating events from different languages and instrumentation levels in the same address space. In support of the different instrumentation schemes TAU provides, a facility for selecting which of the possible events to instrument. The idea is to record a list of performance events to be included or excluded by the instrumentation in a file. The file is then used during the instrumentation process to restrict the event set. The basic structure of the file is a list of names separated into include and exclude lists. File names can be given to restrict instrumentation focus. The selective instrumentation mechanism is being used in TAU for all automatic instrumentation methods, including PDT source instrumentation, DyninstAPI executable instrumentation, and component instrumentation

Figure 2.26 presents an overview of the TAU parallel performance system.

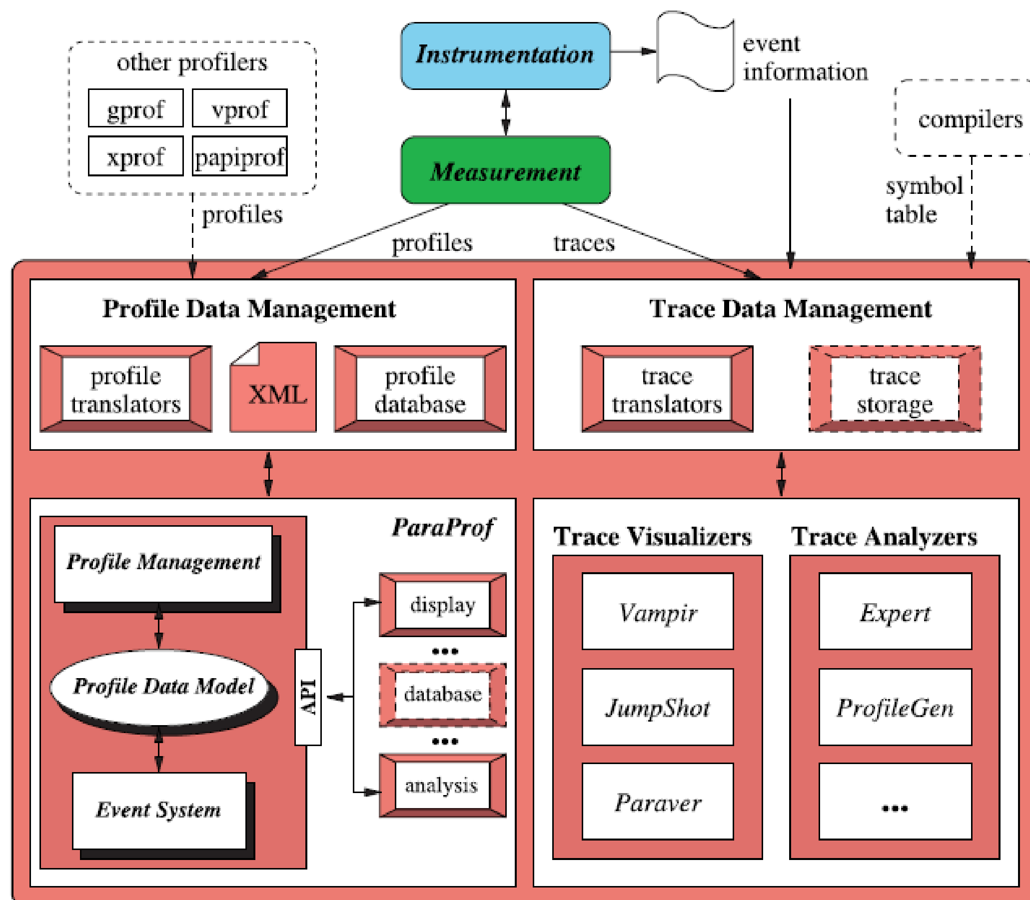


Figure 2.26: TAU parallel performance system overview. Source: [109]

Using the TAU measurement API, event information is passed in the probe calls to be used during measurement operations to link events with performance data. TAU supports parallel profiling and parallel tracing. It is in the measurement system configuration and usage where all choices for what performance data to capture and in what manner are made.

TAU profile measurements compute exclusive and inclusive metrics spent in

each routine. Time is a commonly used metric, but any monotonically increasing resource function can be used. Statistics measured include maxima, minima, mean, standard deviation, and the number of samples. Internally, the TAU measurement system maintains a profile data structure for each node/context/thread. When the program execution completes, a separate profile file is created for each. The TAU profiling system supports several profiling variants:

- Flat profiling
- Callpath profiling
- Calldepth profiling
- Phase profiling

Resulting traces can be visualized with the ParaProf tool. Figure 2.27 shows examples of flat and callgraph profiling.

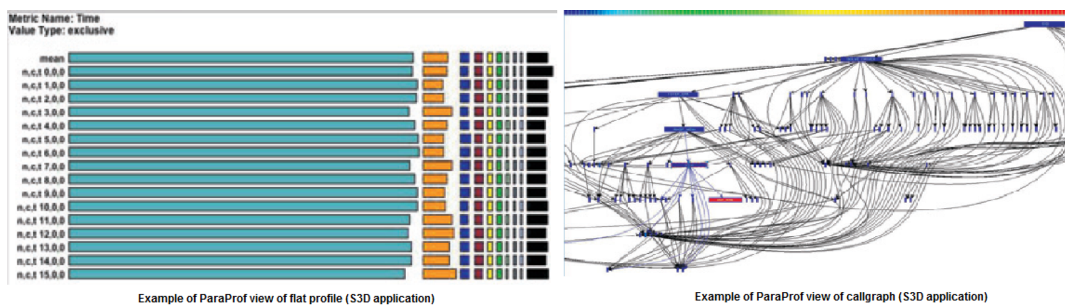


Figure 2.27: Example of display with the ParaProf tool. Source: [109]

2.6.5.6 HPCToolkit

HPCToolkit [2] is an integrated suite of tools for measurement and analysis of program performance on computers. It uses statistical sampling of timers and hardware performance counters (through PAPI). Using HPCToolkit requires to follow a specific workflow (see Figure 2.28) since it is composed of multiple applications.

The first step involves launching an application with HPCToolkit's measurement tool, *hpcrun*, which uses statistical sampling to collect a performance profile. Then, one must invoke *hpcstruct*, HPCToolkit's tool for analyzing the application binary to recover information about files, functions, loops, and inlined code. After that, one uses *hpcprof* to combine information about an applications structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCToolkit's *hpcviewer* graphical user interface.

Figure 2.29 shows an example of hpcviewer display assessing the hotspots.

2.6.5.7 Open|SpeedShop

Open|SpeedShop [40] is a performance analysis for sequential, multithreaded, and MPI applications. It is based on Dyninst for dynamic instrumentation along with binary analysis, and PAPI to collect performance hardware counters results. There

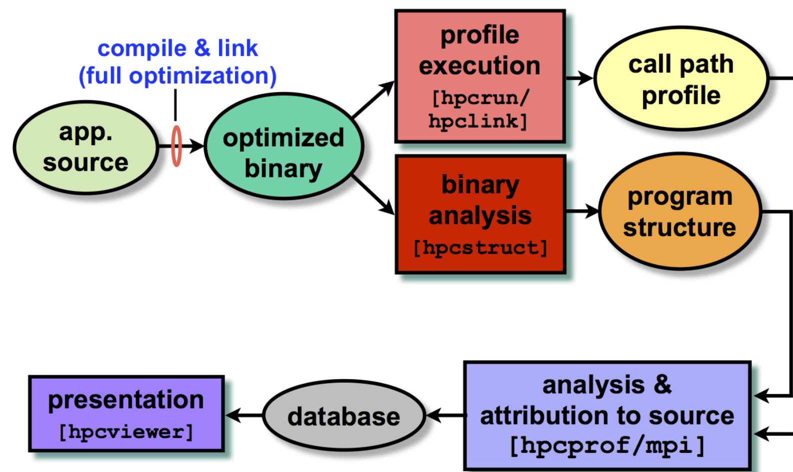


Figure 2.28: An overview of HPCToolkit's primary components and their relationships: Source: http://hpctoolkit.org/

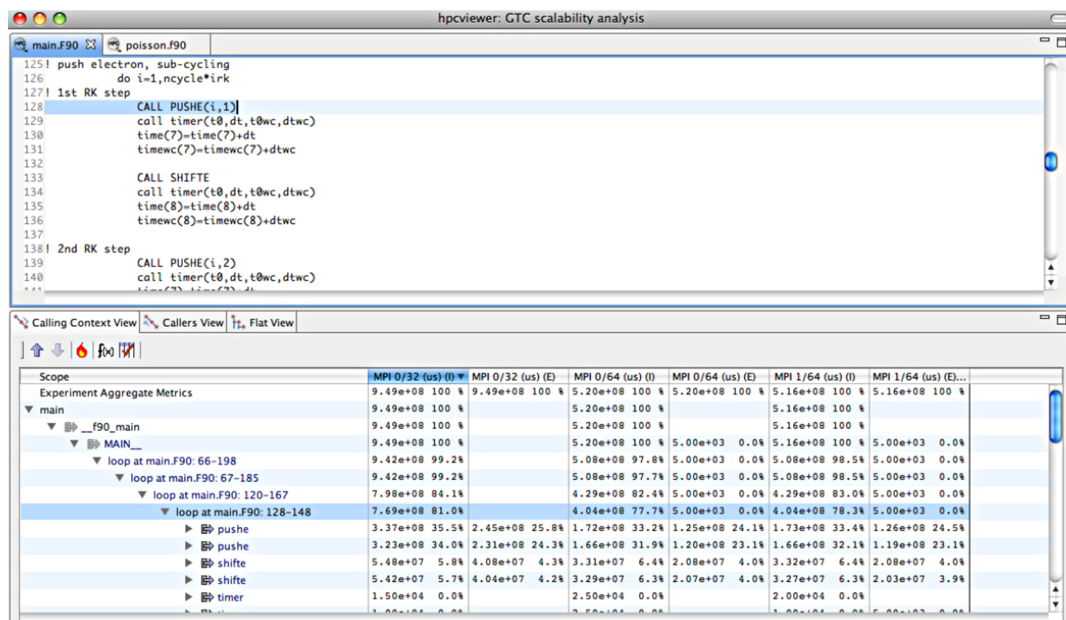


Figure 2.29: Using hpcviewer to assess the hotspots. Source: http://hpctoolkit.org/

are four user interface options: batch, command line interface, graphical user interface and Python scripting API. The GUI uses a wizard-style approach descrived in Figure 2.30

First, an experiment must be selected and then run. There are two classes of experiments, either sampling or tracing.

Sampling experiments There are three available sampling experiments:

- PC Sampling (pcsamp): Records instruction-based (PC) statistics in user defined intervals to provide an overview of the distribution of executed instructions.

- Call path profiling (usertime): Records instruction-based (PC) and call stacks

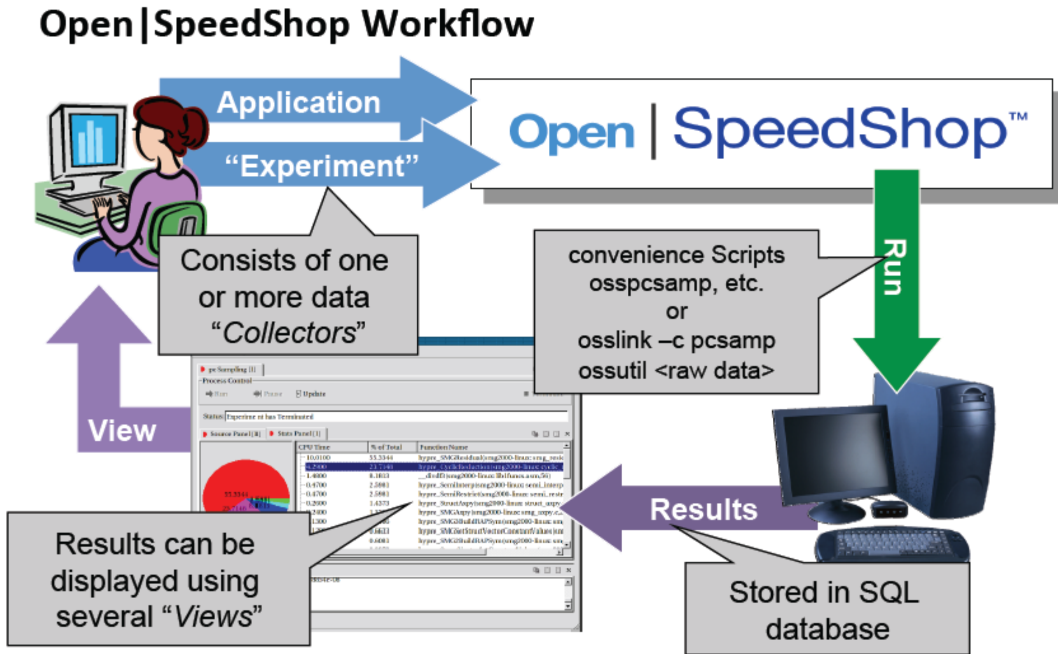


Figure 2.30: Workflow. Source: http://www.openspeedshop.org/wp/category/presentations/

for each sample. Provides inclusive and exclusive timing data. It is used to find hot call paths, caller/caller relations. (user/me)

- Hardware Counters (hwc, hwc/time, hwcsamp): provides access to PAPI hardware counters.

Tracing experiments There are three available tracing experiments:

- Input/Output tracing (io,iot): Records invocation of all POSIX I/O events. It provides aggregate and individual timings along with argument information for each call (iot).
- MPI Tracing (mpi, mpit): Record invocation of all MPI routines. It provides aggregate and individual timings along with argument information for each call (mpit).
- Floating Point Excep/on Tracing (fpe): Intercepts triggered events by any FPE caused by the application. It helps pinpoint numerical problem areas.

Figure 2.31 shows an example of PC Sampling experiment results displayed in the Open|SpeedShop GUI. Results are displayed in the Stats Panel along with the corresponding source lines.

2.6.5.8 Scalasca

Scalasca supports measurement and analysis of the MPI, OpenMP and hybrid MPI/OpenMP programming constructs most widely used in highly scalable HPC applications written in C, C++ and Fortran on a wide range of current HPC platforms. It supports three types of instrumentation:

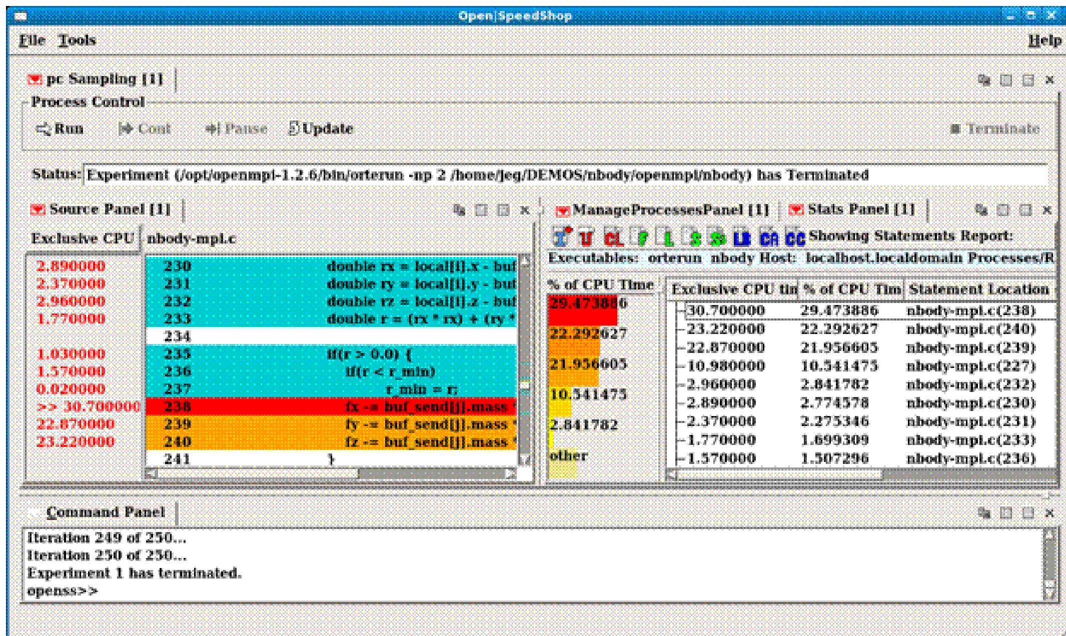


Figure 2.31: PC Sampling experiment results displayed in the Open|SpeedShop GUI. Source: http://www.openspeedshop.org/wp/category/presentations/

- Compiler-inserted instrumentation. Pure OpenMP or Hybrid OpenMP/MPI applications can use the OPARI2 source to source compiler.
- Semi-automatic through the POMP interface.
- Manual instrumentation through EPIK API.

Resulting traces or profiles can then be visualized thanks to the CUBE GUI. Thirdparty profile visualization tools such as ParaProf[13] can also present Scalasca analysis reports.

Figure 2.32 shows an example of runtime summary analysis report in CUBE visualizer.

2.6.5.9 SvPablo

SvPablo is a graphical performance analysis environment for performance tuning and visualization. During the execution of instrumented code, the SvPablo library captures performance data and computes performance metrics including general software statistics and hardware counter data on the execution dynamics of each instrumented construct on each processor. The hardware counter data is captured via the integration of SvPablo with PAPI.

Figure 2.33 is an example of performance statistics for a procedure along with the view of the entire program.

Available procedure metrics are:

- Count
- Exclusive Duration
- Inclusive Duration

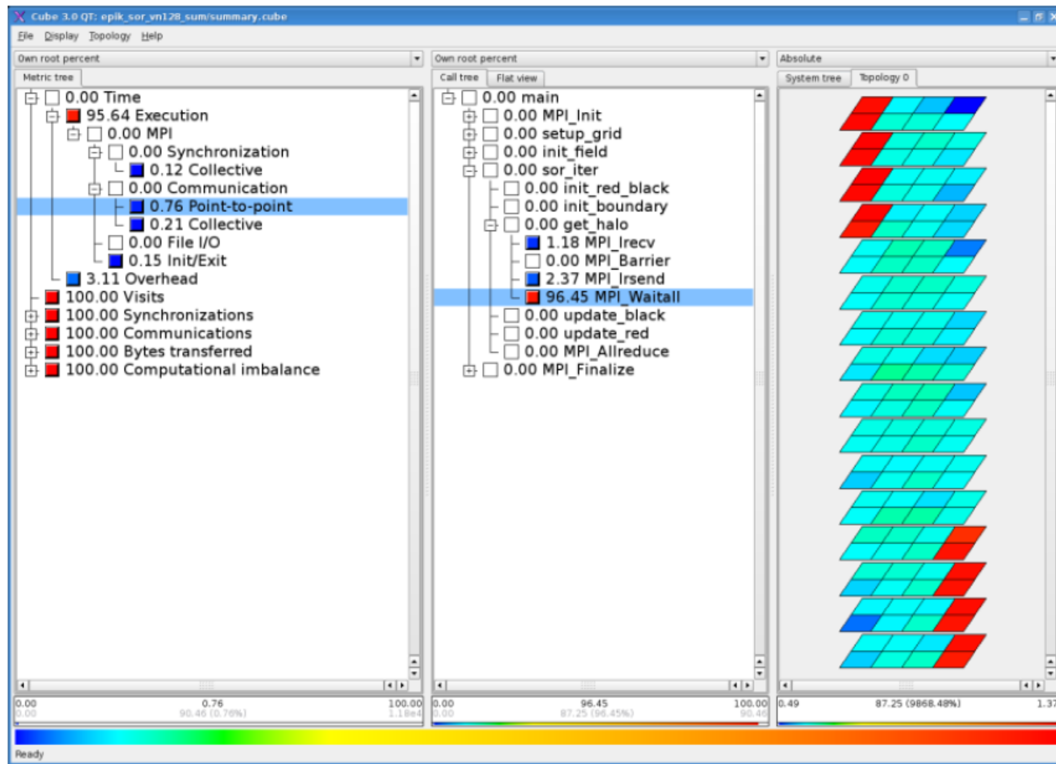


Figure 2.32: Example of runtime summary analysis report in CUBE visualizer. Source: http://www.scalasca.org/.

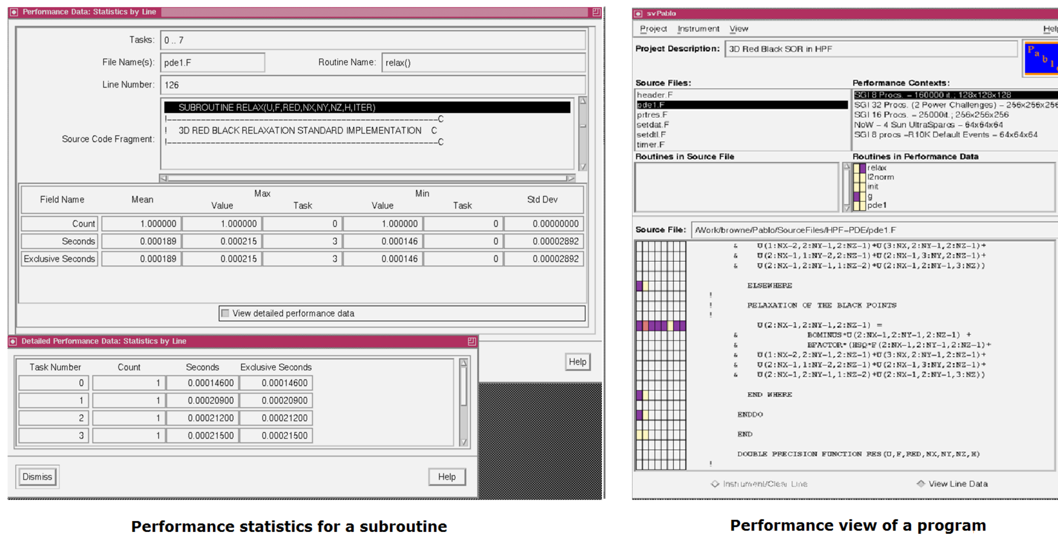


Figure 2.33: Example of performance view. Source: http://www.renci.org/focus-areas/project-archive/pablo

- Send Msg Duration
- Receive Msg Duration

Available line metrics are:

- Count
- Duration
- Exclusive Duration
- Message Send Duration
- Message Send Count
- Message Send Size
- Message Receive Duration
- Message Receive Count
- Message Receive Size

2.6.5.10 PerfExpert

PerfExpert [17] is performance analysis tool that performs automatic bottleneck detection and analysis at core/chip/node level. Measurements are based on HPC-Toolkit. PerfExpert gather performance counter measurements by performing multiple runs with HPCToolkit (PAPI and native counters). It produces sampling-based results for procedures and loops. Figure 2.34 details an example of output using PerfExpert. Results are provided for a given procedure. Performance assessment is given for the following metrics:

- Data accesses
- Instructions accesses
- Floating point instructions
- Branch instructions
- Data TLB
- Instruction TLB

```
total runtime in dgelastic.xml is 75.70 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
               tacc.utexas.edu/perfexpert/

dgae_RHS (59.8% of the total runtime)
---------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- instruction accesses   >>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions    >>
- data TLB               >
- instruction TLB        >
```

procedure identifier
URL to suggested optimizations
overall loop performance is bad
most of runtime is due to data accesses

Figure 2.34: Example of analysis. Source: http://www.tacc.utexas.edu

PerfExpert can correlate results from different runs, as shown in Figure 2.35

When issues are detected, solutions are provided as a list of recommendations. But the user needs to discard the ones that are not relevant himself. Figure 2.36 depicts an example of initial list of suggestions (left) and after eliminating inapplicable suggestions (right).

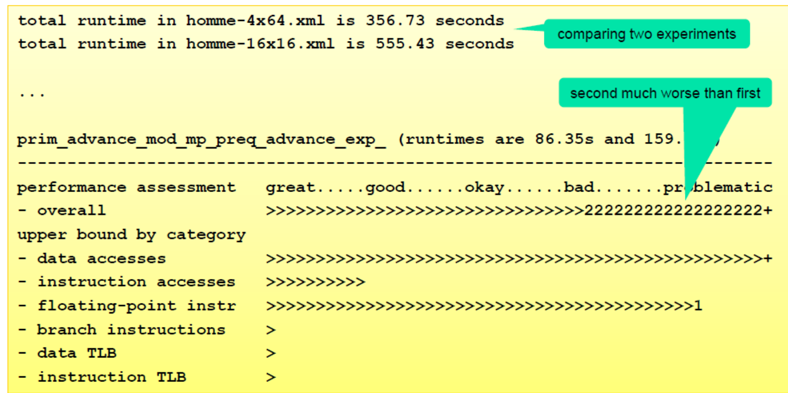


Figure 2.35: Example of analysis with multiple passes. Source: http://www.tacc.utexas.edu

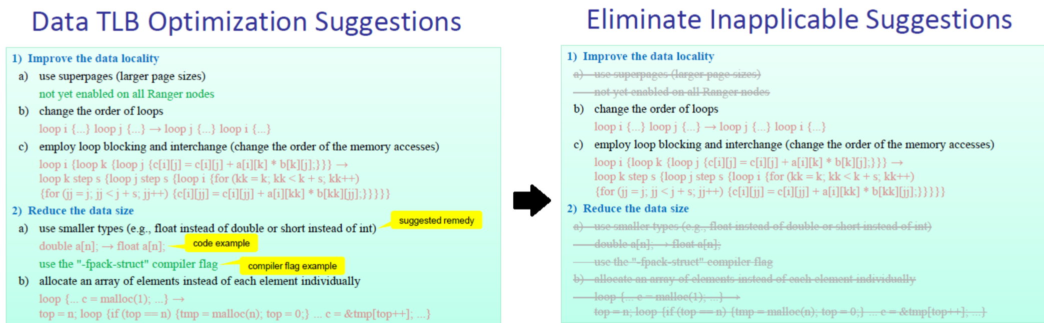


Figure 2.36: Example of suggestion. Source: http://www.tacc.utexas.edu

2.7 Summary

As we mentioned in the introduction of this Chapter, there are many performance evaluation tools. The majority only display collected statistics, but very few provides user-end developers with feedback (hints, suggestions). We can also observe that only a few supports loop-level granularity.

Figure 2.37 presents a summary of the studied performance evaluation tools in previous section in the form of a classification.

Classification / Tools	Code structure granularity					Analysis Method					Insertion Level		User feedback	
								Measure						
	Program	Function	Loop	Block	Inst	Modeling	Simulation	Tracing	Profiling	Sampling	Binary	Source code	Hints	Link to source
gprof		X								X		X		
Cachegring Callgrind		X					X							
CMP$IM	X						X	X			X			
VTune	X	X		X	X					X	X			X
TAU		X						X	X		X	X		X
HPCToolkit										X				
Open|SpeedShop		X												
Scalasca		X	X					X	X		X	X		
SvPablo		X			X					X				X
PerfExpert		X	X										X	X
MAQAO		X	X	X	X		X	X	X		X	X	X	X

Figure 2.37: Classification of studied performance evaluation tools

2.8 Conclusion

In this chapter we introduced and motivated the need for performance analysis in order to harness the ever-growing performance of new multi-core architectures. We first saw the growing discrepancy between modern processors and their memory subsystem. Issues occurring due to the complexity of the memory hierarchy are more and more complex to identify. Similarly, multi-core processors integrate several techniques to increase the instruction-level parallelism. We also insisted on the need for programming models that are adapted to these modern architectures. Parallelization optimizations provided by compilers are not sufficient to cope with the sufficient level of parallelism to exploit the available computing power. In fact, compilers tend to integrate new programming paradigms like OpenMP for instance. But the bottom line is that the programmer needs to be involved even more than in the past. We also pointed the necessity to take into account the choice of data structures and dynamic execution sides effects. Indeed, an architecture will not always behave as it is supposed to and must be characterized. We finally introduced the three main performance analysis approaches before presenting the state-of-the-art performance analysis tools based on these approaches.

In the following chapters we will present the contributions of this thesis to the MAQAO Tool, a domain specific language to easily build performance evaluation tools and a new approach to solving memory related issues.

Chapter 3

MAQAO : Coupling static and dynamic analysis approaches

3.1 Introduction

In general, performance issues are multi-faceted problems. We need multiple angles of view in order to understand and fix them. There are a lot of existing tools, as detailed in Chapter 2, and many of them focus on specific problems. Most of the current approaches focus on only dynamic analysis to collect metrics that pinpoints hotspots. Very few employ static analysis as part of the performance evaluation process. Static analysis can be used for code quality assessment or to lower the overhead of dynamic analysis. We will present our MAQAO tool which couples static and dynamic analyses to cover a broader spectrum of potential issues and provide a better understanding. There is no perfect tool that can detect all the issues that an application may suffer from. That is why we will insist on the extensible aspect of our MAQAO tool, introduced in this chapter, and the ability to easily build new performance evaluation tools tackling specific problems.

Due to overhead and time restrictions, dynamic analysis must be carefully performed. A tradeoff between precision and overhead is always needed depending upon the size (memory and time) of a given application. For instance loop value profiling may provide timing, instances and even iteration information. However, the latter cannot be provided at same time as the former, because resulting timing information would be meaningless. In some cases, it may be necessary to isolate some parts of an application to work on it (when run times are huge).

There are many decisions that must be taken throughout a tuning process. That is why we think that having a toolbox that can address several problems is not an end but the means by which a programmers will be able to improve their application. On top of our MAQAO tool and its modules, we define a methodology for performance application tuning. Depending on the constraints, we can also go through a different method, using in parallel scenarios when possible to maximize the optimization opportunities. For instance if we only consider using hardware performance counters, it is very difficult for a novice programmer to know which counters, or group of counters, he should use. When taking a look at Intel's architecture optimization manual [48] (section 4.3.1), guidances, which recommend to select a set of counters, are given in order of investigation specific issues.

The chapter is organized as follows: Section 3.2 describes our static analysis approach at assessing code quality of an application. Then, Section 3.3 presents our dynamic analysis approach with the aim of capturing the real behavior of an application. Section 3.4 presents the MAQAO tool along with the underlying framework and where both static and dynamic analyses fits. In section 3.5, our methodology of performance tuning will be detailed before concluding in Section 3.6.

3.2 Static analysis : code quality assessment

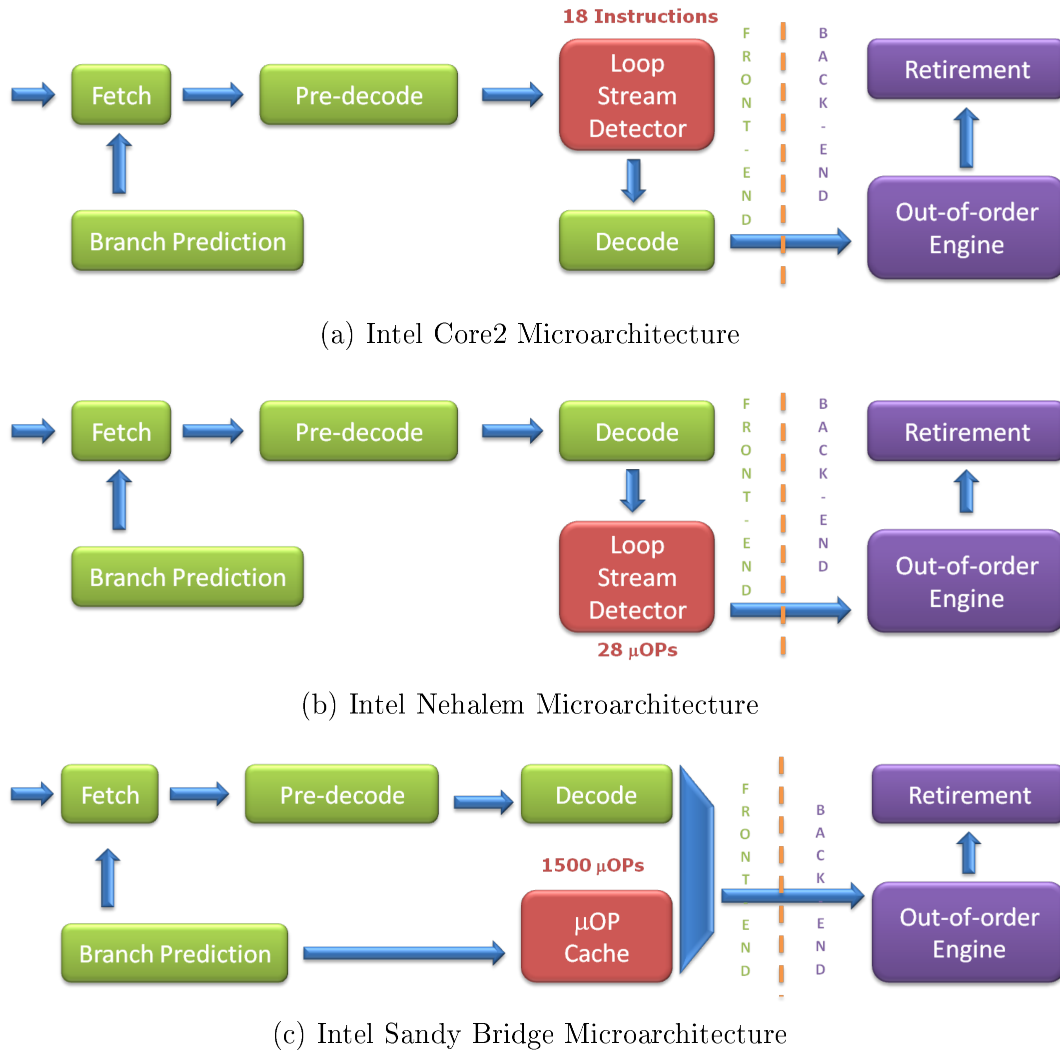
Code quality is tightly coupled with its execution efficiency, on the target architecture it will be executed on. Taking into account the features of a hardware architecture, we want to assess the degree of utilization of these. From a static point of view, code quality on x86 architectures will be mainly defined by the ability to exploit the vectors extensions. Other features that will also represent a part of the degree of quality, are the pressure on registers, size of code, bottlenecks observed on the front or back ends. When applicable, many optimizations can be proposed based on the quality of code.

The aim of code quality assessment will be to leverage the instruction level parallelism (ILP). In order to achieve this goal, we will first present the available architectural speed up levers, before detailing how we can take advantage of them. In the remaining part of this section will focus on the Intel x86 micro-architectures and detail how, from the static analysis of a code, we can take advantages of such architectural levers. The corresponding implementation in MAQAO, i.e. STAN module, will be briefly discussed in section 3.4.

3.2.1 Speedup levers on x86 Microarchitectures

x86 is an instruction set architecture (ISA) originally engineered by Intel and then adopted by AMD, although some extensions, when considering streaming SIMD extensions, may differ (for instance AMD 3DNow! and Intel MMX). AMD was the first to launch a 64 bit extension of the x86 ISA (AMD64). Intel followed some years later (EM64T). When we commonly mention architectures, we usually mean micro-architectures, the hardware units that implements the ISA. In the recent changes made to the Intel x86-64 micro-architectures, some enhancements automatically improved the overall performance of an application, namely, hardware data prefetching, branch prediction, cache size, memory accesses, etc... The remaining features depend upon the quality of the code, that is to say, they mainly rely on the compiler optimizations. Figure 3.1 depicts the three latest Intel micro-architectures. With the Core2 micro-architecture, Intel introduced an instruction queue, that can contain up to 18 instructions, to directly feed the decoders when the program is executing a loop (Loop Stream Detector or LSD in Figure 3.1.a). In the Nehalem micro-architecture (Figure 3.1.b), Intel moved the LSD and its associated instruction queue, after the decoding of instructions. The instruction queue can host up to 28 micro-operations. Intel uses a CISC instruction set that is emulated by RISC. Instructions are decomposed into micro ops by a 4-1-1-1 decoder scheme and instructions requiring more than 4 uops are translated by a microcode sequencer. Notice that when the LSD comes into play, it shuts down the front end execution pipeline, hence saving energy. Starting from the Sandy Bridge micro-architecture (Figure 3.1.c), Intel added a micro-operations cache (also called Decoded Stream Buffer) that can account for up to 1500 micro-operations. With such space, that can be considered as a level zero instruction cache, it is now possible to hold bigger loops and even small functions.

Another major micro-architecture change brought by the Sandy Bridge micro-architecture, is the introduction of the 256 bit Advanced Vector eXtensions (AVX). It is theoretically possible to double the performance of previous applications (actually parts) that were using 128 bit Streaming SIMD extensions (SSE). If we consider a



(a) Intel Core2 Microarchitecture

(b) Intel Nehalem Microarchitecture

(c) Intel Sandy Bridge Microarchitecture

Figure 3.1: Simplified overview of the three latest Intel micro-architectures. Rough description of the front and back ends and the major execution pipeline enhancements

simple operation, non vectorized, it is then possible to obtain a speedup of a factor of 8 for single precision values and 4 for double precision values.

In addition there are reciprocal instructions that exist along with the "full precision" associated instructions. For instance, RCP instruction is an approximation of the reciprocal $(1/X)$ of a single precision floating point value. The aim of this class of instruction is to accelerate the computation of such values. For the previous example computing the reciprocal of a value using the DIV instruction may take up to 80 cycle where the reciprocal RCP instruction only needs approximately 6 cycles. Obviously, this feature will only benefit applications that can still work with a reduced precision, compared to the regular associated computation.

3.2.2 Improving code quality

We will now show how it is possible to improve the quality of a given application by taking advantage of the knowledge we acquired earlier on details about the micro-architectures. The Intel Sandy Bridge one will be used for the example below.

3.2.2.1 Instruction level parallelism

We mentioned previously the ability of a processor to issues multiple instructions per cycle (superscalar) . The Intel x86_64 micro-architectures can issue up to four instructions per cycle. The execution pipeline is fed by the front-end part which sends micro-operations to the back-end (3.1). Increasing the ILP of a code consists in resolving, when possible, all the bottlenecks in the execution pipeline.

Front-end The first global indication that we can extract from a piece of code is the propensity for provoking bottlenecks or under-usages. The front-end is rarely under-exploited. If so we can verify if the considered part of code is not too small or contains only complex instructions (the decoder stage contains four decoders among which only one complex decoder). If a bottleneck is found, we must monitor the micro-operations cache and verify if the part of the code under consideration violates any of inclusion rules (ex: branch density etc). A huge factor of performance can be unleashed if code size is close to the limit and can be reduced to fall into the micro-operations cache. Another indication is the CPI, based on a static performance model that simulates the execution pipeline (by setting as a principle that the out-of-order stages are never a bottleneck and with the assumption that there are no data access stalls).

Back-end Unlike the front end, which is in order, the back-end is out-of-order, although the retirement of instructions remains in order. It means that it is not possible to properly simulate the scheduling stage, since we cannot perform live analysis of memory dependencies. Nonetheless, we can have an insight on the dispatch, considering that the execution is actually in order and the dependency graph between registers (RAW dependencies for critical paths). This way, imbalanced usage of resources can be detected since execution units are specialized (computation, memory addresses, memory data).

3.2.2.2 Vectorization

As stated earlier, vectorization is one of the most important levers when it comes to enhancing the performance of a code. Based on the instructions found in a given piece of code, we can determine it degree of vectorization. Indeed, in order to exploit the vector facility, vectors extensions must be used. The degree of vectorization will be computed as the ratio $\frac{vector\ instructions}{all\ instructions}$. It is even possible to produce multiple ratios by using a two-level categorization of vector instructions. The ISA can manipulate single or double precision values, and for each type we can specify if are using a packed or single group of instructions.

Based on the vectorization degree metric, we can consider three main scenarios.

No vectorization If no vectorization at all is detected, it means that the compiler did not generate any SIMD instructions. Either it is using an x87 code (old scalar floating point-related subset of x86) or the compiler was not able to vectorize because of dependencies or simply because being too conservative. If we detect an x87 code, we must first ensure that we really need it. The only main advantage of x87 is being a scalar unit for numerical calculations sensitive to round-off errors and requiring the 64-bit mantissa precision available in the 80-bit format (IEEE 754-1985). If really needed, the only way to get through is refactoring the algorithm so that it can deal

with the 64-bit format. If the compiler was too conservative, vectorization flags or preprocessors directives may help. The last resort technique is writing the code using intrinsics (which is a low level approach but at least avoids directly writing machine code).

In between vectorization Increasing the vectorization degree of a partially vectorized code relies on the same levers as mentioned above, namely helping the compiler or rewriting the code using intrinsics.

Full vectorization When using double floating-point precision, the only remaining option to increase performance is wondering if single precision can be sufficient. If so, a performance speedup factor of two can easily be achieved by switching to single precision.

Another important fact to take into account when using SSE extensions, is that, due to architectural limitation, mixing integer and floating point operations induces a penalty and should be avoided if possible. AVX does not suffer this problem because on Sandy Bridge micro-architectures, only floating point instructions are available. AVX2 will bring integer extensions.

3.2.2.3 Registers pressure

Most code optimizations relies on the number of available registers. Determining the register pressure of a piece of code may provide interesting indications. For instance, if the register pressure is low then unrolling may be a good option if applicable. Alternatively, if the compiler resorts to spill/fill mechanisms and there are available registers, then it means that its register allocation heuristic failed to use all the available registers. The compiler actually uses an approximation and not the optimal solution since the problem is NP-Complete. One option is switching to intrinsics if it is worth it. Notice that this kind of optimization may really pay only if the L1 cache cannot host spill/fill values.

A static analysis can asses the code quality of an application, relying on the hypothesis that data is in cache. But that is not always the case. That is why we need to capture the dynamic behavior.

3.3 Dynamic analysis : capturing the dynamic behavior

At the time of writing an application, a programmer does not know, and sometimes does not yet wonder how well the application will behave on the machine it will run on. Effective performance of an algorithm is related to its implementation and compilation phases. Dynamic analysis is a method to observe and understand how an application behaves at runtime. To this end, we can use software or hardware techniques to monitor and capture metrics while the program is running. In our case, the study is software-oriented. Nevertheless, the hardware alternative will be briefly considered. In this section we will first establish the importance of architecture characterization before studying code characterization. Then we will discuss coarse and fine grained performance evaluation.

3.3.1 Architecture characterization

Some issues observed when running an application are actually caused by the machine itself, more precisely, the CPU. A part of these are, ironically, a consequence of micro-architecture optimizations and others are not supposed to happen at all. For instance, consider an iterative algorithm that uses values from a previous iteration. Some read instructions will take place after writes. On x86 architectures, a special hardware optimization monitors this kind of pattern and verify if a read instruction is not actually reused in the succeeding write one. If so, the value is directly passed and a memory access is avoided. The only problem is that this mechanism may fail because of the way it compares memory references. When the problem is detected by the hardware itself, the operation is reissued but at a higher cost. That is why we need an architecture characterization phase, in which we will try to verify that the machine is working as it is supposed to. Of course, performing all the possible tests is impossible. As a consequence we only target specific tests that checks important architecture components. For example, we verify memory issues by testing different set of memory alignments or combination of operations on arrays. Others tests measure memory bandwidth limitation using a varying number of available cores. These tests are done thanks to micro-benchmarking test suites that are out of the scope of this thesis. The final result of these tests are integrated in a machine description file that contains all the collected information and can then be taken into account in further analyses.

3.3.2 Coarse grained performance evaluation

Coarse grained analyses are intended to locate, at a macro level, the portions of code that are responsible for consuming an important amount of an arbitrary metric. Usually, time-consuming functions are the target. However, more subtle approaches may be used.

We have developed our own approach through MAQAO Profiler. Not only do we propose multiple profiles, but we also support different programming models. Profiles are predefined coarse grained analyses that employ different methods to time an application. Existing profiles are:

- FX : measures only exclusive time spent by functions
- FI : measures exclusive and inclusive time spent by functions
- LO : measures time spent in outermost loops
- LI : measures time spent in innermost loops
- FXLO : combination of FX and LO profiles
- OPR : measures time spent in OpenMP parallel regions
- OFOR : measures time spent in OpenMP parallel for regions
- OPRFOR : combination of OPR and OPR profiles

Supported programming models are :

- Unicore : one core and one thread

- OMP : multithreaded. Support for Intel and GNU OpenMP runtimes
- MPI : multiprocess
- Hybrid : combination of OpenMP and MPI

Supporting multiple profiles allows a more flexible approach to coarse grained performance evaluation. Moreover some profiles include new optimizations to lower the instrumentation overhead. For instance, The FI profile contains some optimizations compared to equivalent approaches. As mentioned before, The FI profile measures exclusive and inclusive time spent on a function basis. The FI profile has a far higher cost compared to FX because it implies timing call sites. Our optimizations occur during the instrumentation phase. Thanks to a static analysis based on the control flow graph, we determine which call site instrumentation probes can be avoided. Depending upon the number of call sites discarded, the actual gain can be non-negligible. The missing timings for these call sites are actually calculated afterwards, when instrumentation results are processed.

We also propose to modify the way instrumentation compensation is performed. By default we include a micro-benchmark procedure before the application really starts and then use that information to fix the deviation caused by the time spent by instrumentation probes. With this additional lever it possible to control the default behavior. Two additional strategies are proposed, namely, not applying any compensation mechanism or setting an arbitrary compensation penalty.

Figure 3.2 describes a typical output using a The FI profile and the OpenMP programming model.

3.3.3 Code characterization

There are plenty of analyses that can be run on a portion or code. In order to target the relevant ones, we must first perform a characterization process . A piece of code ordinarily contains both arithmetic computations and memory accesses. If the amount of time spent to service memory read and write operations is longer than the computations, then the code is dominated by memory accesses, i.e. memory-bound. Alternatively, if the computations require more time then, it is compute-bound. In between cases may exist and would imply using all the available analyses. The difference between the latter case and no prior code characterization is that you can know if it is worth the effort. To this end, we use DECAN [60], a technique based on decremental analysis, to iteratively identify the individual instructions responsible for performance bottlenecks. A DECAN analysis implies multiple variants (binaries) that actually focus on either memory streams (called MISTREAM) or arithmetic streams (called FPISTREAM). The exiting variants are the following:

- MISTREAM_AS : keeps only memory instructions and delete the arithmetic ones. Arithmetic instructions using memory operands are converted into memory instructions having that same operand.
- MISTREAM_AN1B : same as MISTREAM_AS but replacing the deleted instructions by a 1-Byte no operation (NOP) instruction.
- MISTREAM_AMNB : same as MISTREAM_AS but replacing the deleted instructions by a no operation (NOP) instruction that accounts for the size of the removed instructions.

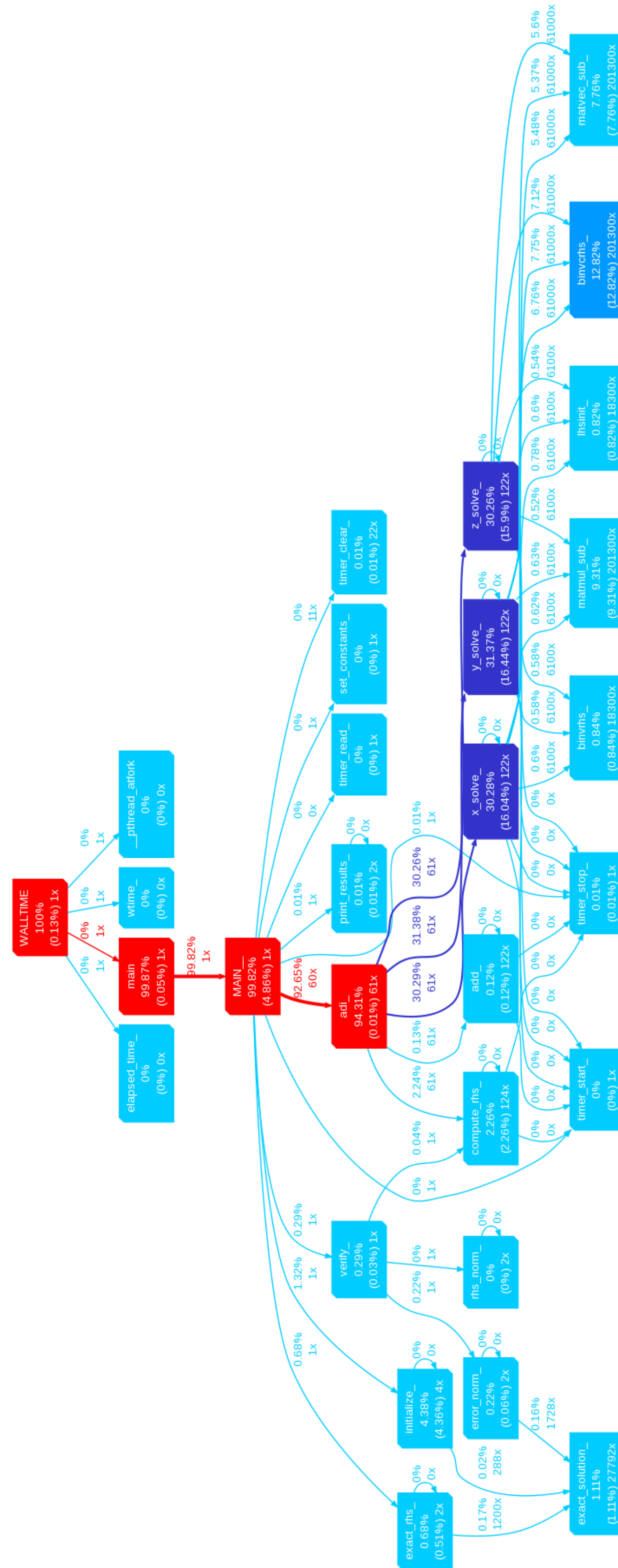


Figure 3.2: MAQAO Profiler output for NPB3.3 bt benchmark (class.S). Each box exposes the name of a function along with its exclusive and inclusive percentage time (related to the walltime)

- FPISTREAM : keeps only arithmetic instructions and replace memory in-

structions by an instruction that is not sensitive to data dependencies (PXOR on x86-64 architectures).

- DSTREAM : Arithmetic instructions using memory operands are modified to point to a same memory reference.

If DECAN is not applicable for any reason, hardware performance counters are used to fill in the missing information. It is possible to aggregate the total number of cycles spent for each instruction of an application. Based on this information, it is trivial to locate time-consuming instructions. Then we can create the two previously mentioned groups, i.e. memory and arithmetic categories.

3.3.4 Fine grain performance evaluation

When hotspots are located and the corresponding piece of code is characterized, we can then apply specific analyses. Static analysis presented in the previous section addresses computation-bound issues. Memory issues are analyzed with our Memory Trace library module. Besides these two main categories (memory and computation), we also identify optimization opportunities using value profiling.

We propose three possible analyses:

- Loop Value profiling : when loop bounds are unknown at compile-time, detecting these at runtime may reveal very small iteration domains. As a consequence, a specialization should be performed.
- Function value profiling : storing histograms containing the values of each parameters may identify constant values or fixed intervals. Such scenarios can profit from a cache mechanism that avoids the call itself by returning previous cached results.
- Evaluating the difference between statically predicted and dynamically measured cycles : since static analyses always consider data to be in level 1 data cache, an interesting comparison is evaluating the discrepancies between static predictions and dynamic results. A significant gap would mean that something is going wrong and further investigations need to be conducted.

3.4 MAQAO tool and Framework

In its early days, MAQAO, the Modular Assembly Quality Analyzer and Optimizer, was a monolithic application for performance tuning on Itanium architectures. With the increased power of x86 architectures, support for Itanium was discontinued and the focus is now on x86-64 architectures. During this transition, the application itself have been split into a Framework containing a set of building blocks on top of which the performance analysis tool is built.

In this section we will discuss how the main components of the MAQAO Framework are organized. Then we will focus on the tool itself, analyzing how it uses the Framework and also briefly describe the existing plugins.

3.4.1 MAQAO Framework

Using a Framework-based approach had appeared to be necessary in order to easily develop new plugins for the tool. Figure 3.3 depicts the framework overview which

can be decomposed into three main layers, namely, binary manipulation, code analysis (static) and plugins.

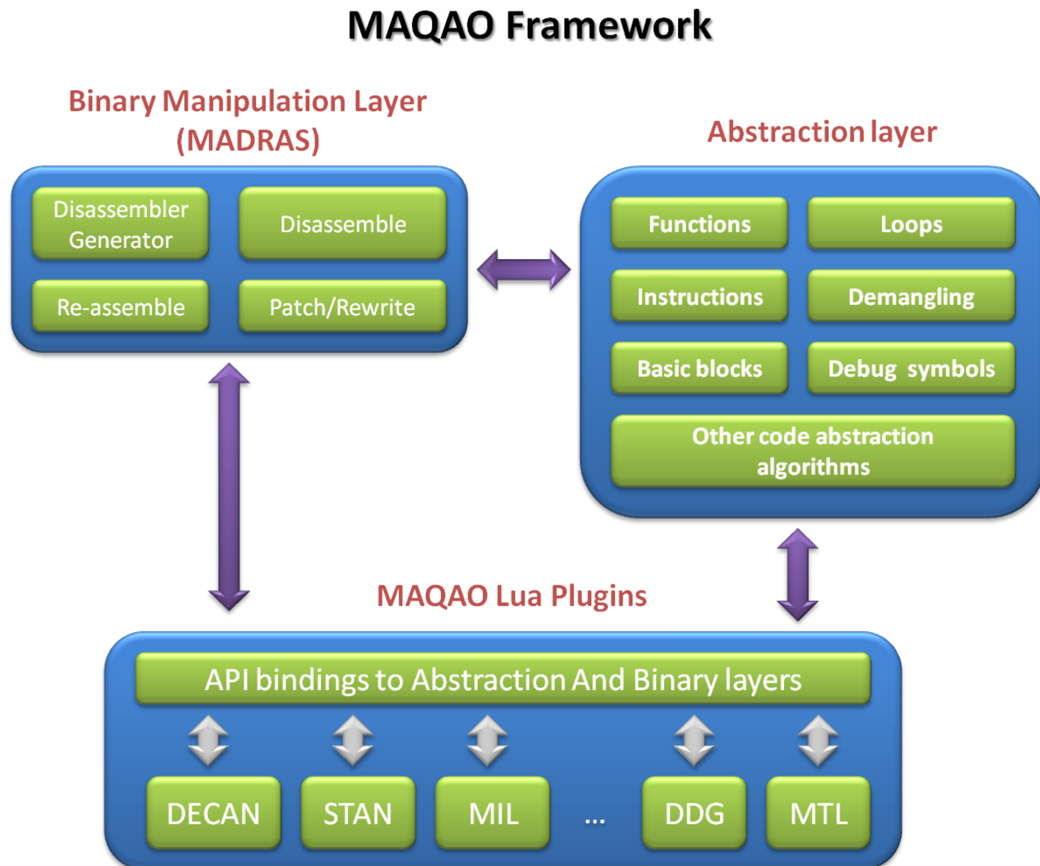


Figure 3.3: MAQAO Framework

Binary manipulation Binary manipulation is always the first solicited layer since it is responsible for exposing the instructions and boundaries of the functions contained in a binary. Static analyses only use the disassembler block. When dynamic analyses are necessary, binary rewriting is involved. The assembler block is actually part of the binary rewriting process if no originally existing instructions must be created or if assembly code is added.

Strcturation and Abstraction layer Once instructions are extracted and categorized (branches, function boundaries, calls, etc ...), then is important to build an abstract representation of the code in order to easily manipulate it. This layer provides means to generate call graphs, control flow graphs, loop detection algorithms amongst others. All our analyses and existing modules are actually built on top of it.

MAQAO API and Plugins The last layer of the Framework provides a simplified interface to the previous ones and enables developers to easily build plugins also known as modules. The main purpose of this layer is to leverage the productivity and simplicity of programming. We have introduced a scripting language based on the Lua language [88]. It features iterators and functions that relates specifically to

the structural abstract objects produced by the abstract layer. Figure 3.4 reveals the hierarchical view of these structural abstract objects.

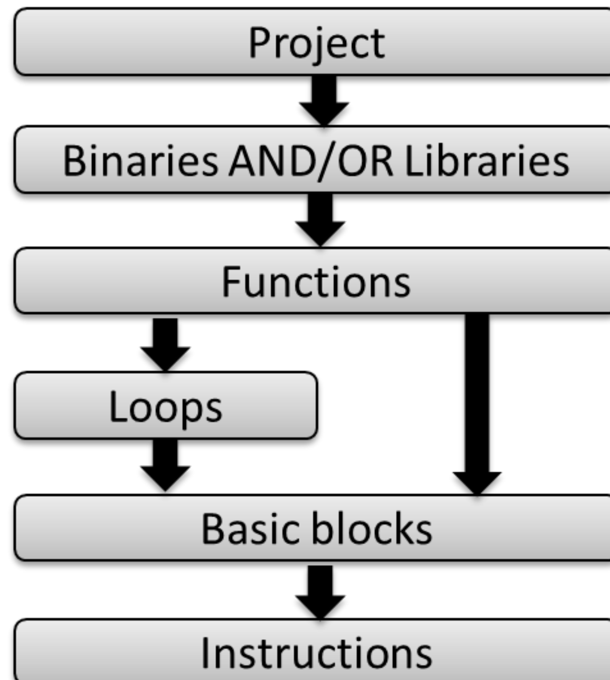


Figure 3.4: Hierarchy of the structural abstractions defined in MAQAO

Figure 3.5 illustrates how easily targeting and manipulating load memory instructions found in innermost loops of an application. It first loads a binary and then iterates hierarchically until instructions are found in innermost loops, before verifying if it is a load memory instruction. Finally it displays the instruction itself and the load memory operand. There are several functions available for each abstract object that can test structural properties (innermost loop here for example) and retrieve specific information (here the first memory operand since it is a load instruction). More examples are available in Appendix A.3.

3.4.2 MAQAO Tool

MAQAO is an open source performance evaluation tool. The philosophy behind MAQAO is based on the analysis of a given binary application, to provide end-user programmers with feedback related to performance issues, We think that it is better to perform our analyses on the binary produced by the target compiler, rather than the source code itself. Indeed, some performance issues are related to the transformations performed by the compiler. By working on the binary, we actually work on the real code which is executed. But it does not mean that source code information is useless. Quite the contrary, any feedback to the user is useless if he cannot link the information we provide with source code lines. We use debug information to connect binary instructions to source lines. Modules that look for performance issues always output reports. Reports generally produce either text advices, pictures or plots representations. Another import point, is the fact that MAQAO is loop-centric. That means that besides functions, MAQAO analyses are focused on loops because, usually, most of the time is spent there. In particular

```
--Create a project and load a given binary
local project = project.new ("targeting load memomry instructions");
local bin = proj:load ( arg[1], 0);

-- Go through the abstract objects hierarchy
-- and filter only load memory instructions
for f in bin:functions() do
  for l in f:innermost_loops() do
    for b in l:blocks() do
      for i in b:instructions() do
        if(i:is_load()) then
          local memory_operand = i:get_first_mem_oprnd();
          print(i);
          print(memory_operand);
        end
      end
    end
  end
end
```

Figure 3.5: Displaying load memory instructions

we group assembly loops that belong to the same source loop. We also think that the compiler should remain the primary optimization tool of the code developer. Many of our analyses aims to explain how to better use the compiler. Sometimes it may be related to data structures. Indeed, a developer may not appreciate how a cache handles a data structure. The choice is usually based on what is thought as the best data structure for a problem and not taking into account architectural considerations (caches structure).

Figure 3.6 describes the majors steps and interactions with an application and its developer. The application is chosen, then disassembled before being structured. Then the user can select a loop and a specific analysis that will provide him with reports that are related to his source code. He can then modify his code, change compiler flags or even parameters of involved runtime libraries and verify the gain obtained, if any. The process can then be repeated until fixing all the exposed issues.

At the time of replacing the old version of MAQAO, we decided to switch the user interface to a web service, following the Client/Server architecture. As a consequence, we can have the server on a target machine (server) and the client on a different (user) machine. MAQAO tool is thus composed of two parts, a server module and a client GUI interface. The GUI itself is a website, thus enabling any user outfitted with a recent web browser to execute the front-end application. Practically, MAQAO can be used either in server mode (webservice for GUI), as described above, or in batch (command line) by invoking each module separately (providing specific flags).

Server The server is actually a layer on top of the Framework layers and is responsible for establishing a link between service requests, which are actually messages, from the GUI and then execute theses request by invoking the concerned modules.

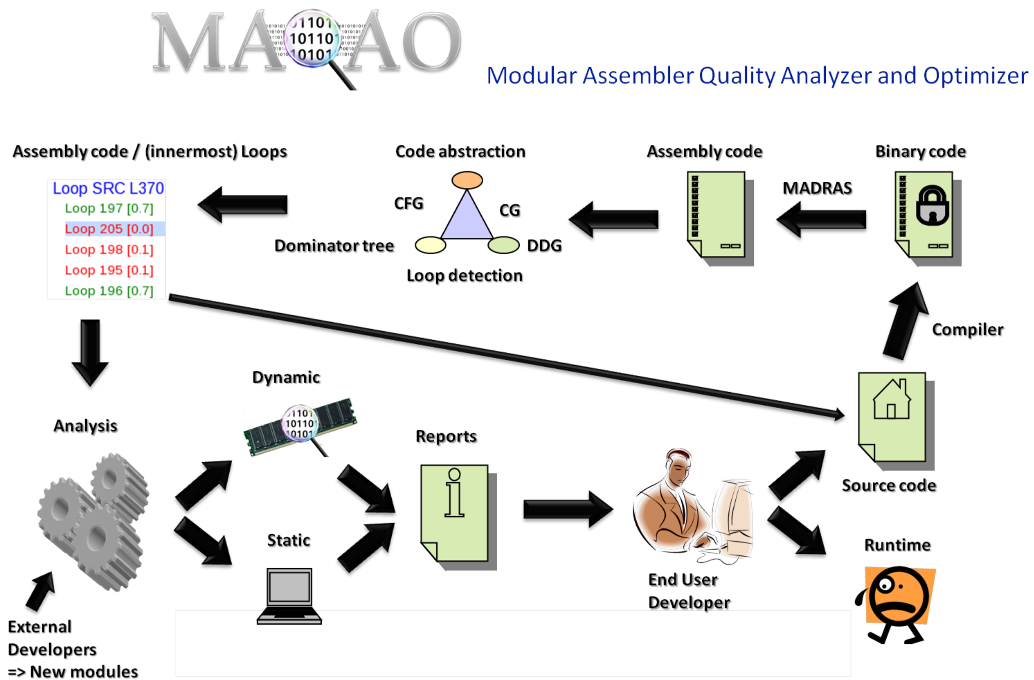


Figure 3.6: MAQAO Tool overview

Results are then sent back to the Web GUI. As mentioned before, it can also be used in command line.

WebUI : Web User Interface front-end The graphical user interface is most of the time the entry point for a user. It may provide plenty of features but must remain simple. In our case, the GUI follows the principles evoked in the overview of MAQAO. Figure 3.7 shows the interface before (Figure 3.7.a) and after having loaded a given binary (Figure 3.7.b). The interface of MAQAO is composed of four main parts:

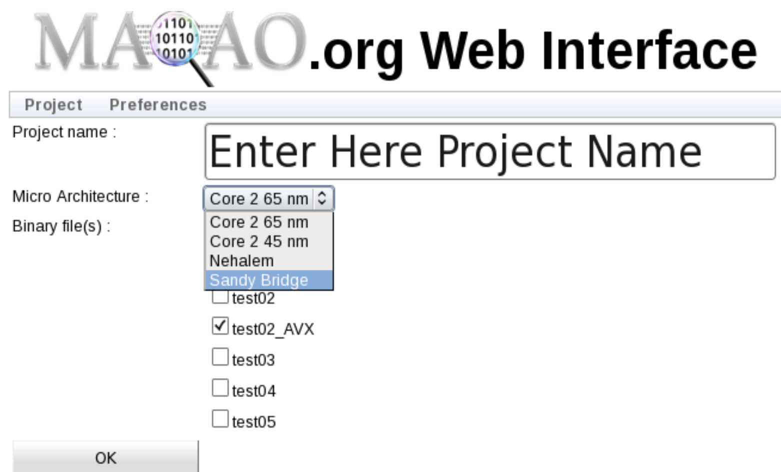
- The left panel presents a hierarchical structuration of the loaded binary application. It is split into functions and then assembly loops which are grouped by source loop. The grouping is based on debug information provided by the compiler.

- The right panel presents the source code if available.

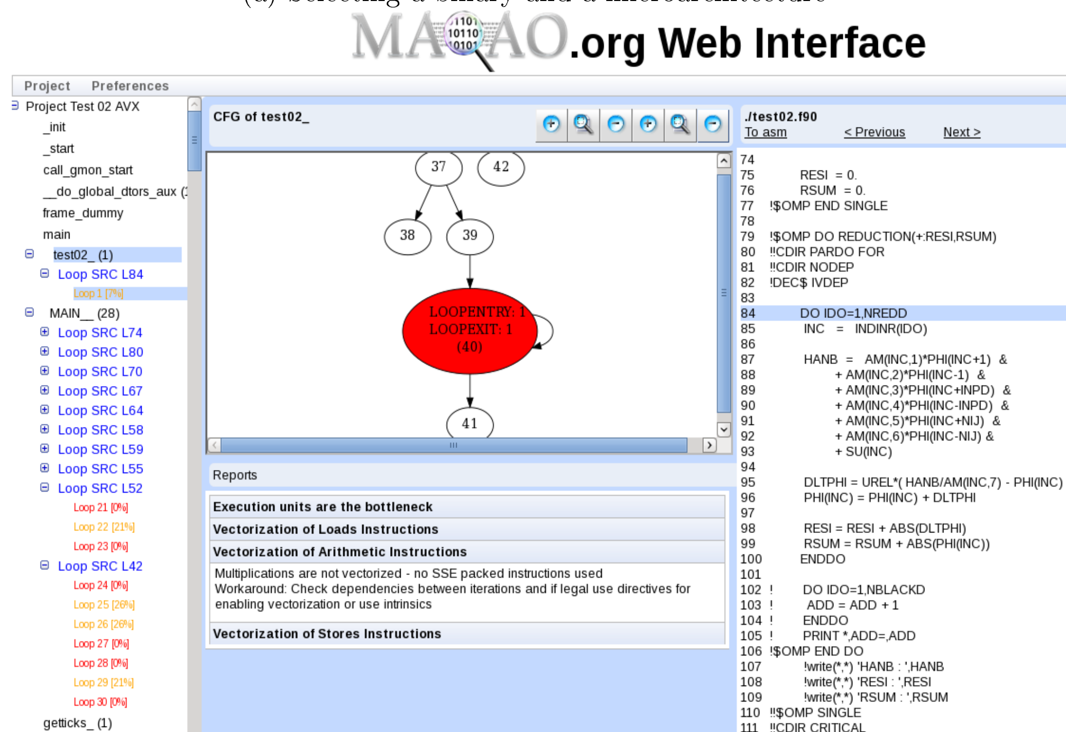
- The bottom panel displays reports (hints) when analyses are performed.

- The central panel displays visual representations of the code. At application level, it displays the call graph. At function level, it displays the control flow graph. At source level it can display the data dependency graph (DDG)

In the remaining part of this section, an overview of each of the existing modules will be given.



(a) Selecting a binary and a microarchitecture



(b) Main interface of MAQAO

Figure 3.7: MAQAO GUI : Web front-end

STAN : STatic ANalysis STAN performs static analyses in order to assess the code quality of a portion of code. It addresses a significant part of the issues exposed in Section 3.2.

The key features of STAN are the following :

- Predict performance based on a static performance model
- Supports different micro-architectures
- Vectorization ratios and potential speedup : enables the prediction of vectorization speedup (when applicable)

- Unroll factor detection : enables static prediction of performance for different unroll factors (given a set of variants produced by the compiler).
- High latency instructions (division and square root) replacement opportunities : identifies potential locations where use of RCP (1/x) and RSQRT (1/sqrt(x)) would prove beneficial.

All these features have a common goal, providing hints and workarounds to improve static performance. A typical output is exhibited in Figure 3.8.

MTL : Memory Trace Library and related analyses Details, along with examples, of the MTL module will be treated in Chapter 5.

The MTL module needs three parameters to work properly, namely, a target machine description file, the associated micro-benchmarking map and the programming model. By default, if one or more of these flags are missing, the corresponding reports will be turned off and the programming module is set to uni-core (one process and one thread). If so, only architecture independent reports will be generated.

MIL : Maqao Instrumentation Language The MIL module accepts an input file, which is a MIL instrumentation file. Details on the instrumentation language and some examples are provided in Chapter 4.

Profiler Details on the MAQAO Profiler module, including examples, have been covered in the previous section.

The first stage consists in selecting a profile. Then a programming model (model flag) can be selected, for instance to specify, OpenMP, MPI or both (Hybrid). Even if the instrumentation process is transparent for the user, the latter indication is very important because it will define which instrumentation library must be used. After that, the user can execute the instrumented binary on a target machine. The final step reads results and output, according to the selected output format, either text information or a call graph.

It can be used in pipelined mode instead of step by step. It means that all the steps are performed in a row on the same machine. The default profile is function instrumentation (time) and the default model is uni-core (one process and one thread).

DECAN : DECremental ANalysis As discussed in the previous section, the DECAN module is used to generate multiple versions of a considered object, generally loops. Each version either removes and/or replaces arithmetic or memory instructions in multiple variants.

Grouping The grouping module focuses on a specific aspect memory accesses, in particular, detecting groups of instructions that have common memory access expressions. In general, that means locating instructions that access to a same portion of memory, e.g. arrays. In practice, this module is a means and not an end in itself. Its results are usually a starting point for other analyses. For instance, DECAN uses the grouping module.

```
Composition and unrolling
-------------------------
It is composed of the loop 0 and is not unrolled or unrolled with no peel/tail code (including
vectorization). Type of elements and instruction set 3 SSE or AVX instructions are processing
single precision FP elements in scalar mode (one at a time).

Vectorization
-------------
Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode).

Matching between your loop and the binary loop
----------------------------------------------
The binary loop is composed of 1 FP arithmetical operations:
1: divide
The binary loop is loading 8 bytes (2 single precision FP elements).
The binary loop is storing 4 bytes (1 single precision FP elements).

Arithmetic intensity is 0.08 FP operations per loaded or stored byte.

Cycles and resources usage
--------------------------
Assuming all data fit into the L1 cache, each
iteration of the binary loop takes 14.00 cycles.
At this rate:
 - 0% of the peak computational performance (0.07 out of 16.00 FLOP per cycle (GFLOPS @ 1GHz))
 - 1% of the peak load performance (0.57 out of 32.00 bytes loaded per cycle (GB/s @ 1GHz))
 - 1% of the peak store performance (0.29 out of 16.00 bytes stored per cycle (GB/s @ 1GHz))

Pathological cases
------------------
Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED.
Since your execution units are vector units, only a fully vectorized loop can use their full
power.
By fully vectorizing your loop, you can lower the cost of an iteration from 14.00 to 3.50
cycles (4.00x speedup).
Two propositions:
 - Try another compiler or update/tune your
current one:
   * gcc: use O3 or Ofast. If targeting IA32, add mfpmath=sse combined with march=<cputype>,
          msse or msse2.
   * icc: use the vec-report option to understand why your loop was not vectorized. If
          "existence of vector dependences", try the IVDEP directive. If, using IVDEP,
          "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
 - Remove inter-iterations dependences from your loop and make it unit-stride.

WARNING: Fix as many pathological cases as you can before reading the following sections.

Bottlenecks
-----------
The divide/square root unit is a bottleneck. Try to reduce the number of division or square
root instructions. If you accept to loose numerical precision, you can speedup your code by
passing the following options to your compiler:
 * gcc: (ffast-math or Ofast) and mrecip
 * icc: this should be automatically done by default

By removing all these bottlenecks, you can lower the cost of an iteration
from 14.00 to 1.50 cycles (9.33x speedup).
```

Figure 3.8: Example of output produced by STAN

3.4.3 Contributions related to MAQAO

In order to clearly describe which parts of MAQAO have been created or enhanced throughout this thesis, my contributions are listed below :

- Very first version of STAN (not used anymore)

- Second version of the WebUI front end (current version)
- Profiler
- DDG : data dependency graph (only visible from the Web User Interface) and actually mostly used as an intermediate module
- MTL
- MIL
- Lua/C API stubs and plugins infrastructure
- MAQAO Standalone executable (neither extra packaging nor dependencies are needed to deploy MAQAO, just one binary)

3.5 Methodology

Performance tuning requires a comprehensive set of tools. Each will tackle one or a set of issues related to a common problem. But having said that, there is still a problem. How and when each of them should be used. Solving performance issues of an application is rarely a one-shot process. Multiple passes are needed in order to gradually identify the issues that matters and are worth investigating. In a nutshell, we need a methodology. Performance tuning actually leaves no room for improvisation, or very few, as a last resort option, when all the others failed. A methodology defines a systematic workflow that a performance tuning process should stick to. In fact, it is a decision tree that guides a user throughout decisions at each step. We can view it as a multi-level scenarios map. Each scenario will combine different tools in order to fix the maximum number of noticed issues. Moreover, we do not think that a magic push-button tuning software can exist. At some level we need the user to make his own decisions, and accept to continue in a particular path. MAQAO first tries to lower down the number of external tools needed by providing always more modules to address specific issues considered as crucial in regard to performance. For instance it uses a combination of static and dynamic analyses where commonly other tools only perform dynamic analyses. It also provides, through its reports, advice on how to proceed with a specific tool.

In this section we will study the main steps of our methodology. First we identify a clear goal to achieve, for instance reducing the overall execution time. Then the focus will be directed towards finding hotspots. Each hotspot should then be characterized in order to select relevant tools. Finally we will see that the process may need to be repeated more than once.

3.5.1 Defining a goal

Most of the time, the objective of an end-user programmer is to optimize his application so that it can run faster, meaning that it gets the best out of the target machine it is running on. Other objectives may use different constraints, like memory space or energy. For instance using less energy has turned out to be an inescapable trend. Sometimes it is possible to have our cake and eat it too, by accomplishing both, for instance in the case of memory-bound applications that can use computation units at a lower frequency (memory one).

3.5.2 Locate hotspots

Locating hotspots is a crucial step in the performance tuning process because all of the remaining choices that will be driven by their identification. For instance, when an application has different input sets, it is very important to know if these are comparable or may produce different behavior. This step happens at coarse grain. The common pattern has two passes. We first find hot functions and then hot loops. The main idea is to locate the smallest parts of the code that account for the largest fraction of the time. It is better to focus on small parts because optimizations can be applied more easily.

3.5.3 Characterize target hotspots

Once hotspots are located, a characterization of these should take place. The following steps heavily depend upon it since different modules are available. If we consider a set of hot loops, we can use DECAN to figure out if that part of code is memory-bound or compute-bound. Then we can concentrate the efforts on specific analyses. If DECAN is not applicable for some reason, hardware performance counters may help albeit more complex to manipulate.

3.5.4 Use relevant tools

As mentioned before, selecting relevant analyses is conditioned by the characterization of the type of code. But we must not forget to also include on the balance the model of programming being used. To illustrate the latter point, we will considers two parallel programming models, MPI and OpenMP. When using MPI, we may include a communication analyses. Concerning OpenMP, a multi-thread analyses, focusing on interactions between threads should be chosen, like MTL for instance. We can even opt for an intermediate profiling level, in particular, (OpenMP) parallel regions. Another major concern is dealing with long-running applications. It is not reasonable to consider running an application that lasts days multiple times. The method we have adopted when facing this kind of case is to outsource a piece of code and fix it. Of course the tool selection will still depend upon a trade-off between accuracy and time required. The most accessible tool remains the compiler. Several analyses will provide hints to modify flags or use specific features like preprocessor directives and intrinsics. This approach involves an iterative tuning process.

3.5.5 Iterating through the process

The tuning process can involve multiple changes to the source code, environment or compiler flags, and may need multiple iterations to be achieved. The main question is not : “is my application perfect now ?” but rather : “Is it worth continuing for the remaining potential gain ?”. It is actually always trade-off unless an application is extremely simple and predictable.

3.6 Conclusion

In this Chapter we presented MAQAO, a tool that enables user-end developers to analyze, understand and optimize their applications. A description of the tool, along with the Framework it is built on top of has been presented. All the existing

modules are briefly explained. We first illustrated how the assessment of code quality of an application, through static analyses, can reveal multiple issues. Thus, helping to obtain more performance out of the target architecture. Then, using a dynamic approach, we showed how to characterize the behaviour of an application at different level of granularities.
By mixing static and dynamic analyses, it is possible to understand complex multi-faceted issues. We proposed a methodology coupling both static and dynamic approach to get the best performance out of a target architecture, in particular, the x86-64 micro-architectures.

End-user programmers may want to look for issues that are not covered by any tool. If neither MAQAO nor any other tool suit ones needs, the only remaining option is building a customized, specific tool. We will see in the next Chapter how it is possible to easily and quickly build customized performance evaluation tools using a domain specific instrumentation language, MIL.

Chapter 4

Instrumentation language

4.1 Introduction

As software complexity increases with the development of multi-core architectures, high performance parallel applications are increasingly difficult to tune for performance, to debug and profile. Due to compiler optimizations, runtime interactions and complex shared memory hierarchies, capturing the runtime behavior of the code is an essential step in code analysis. The purpose of binary instrumentation is to insert new code into an executable in order to collect and analyze information concerning an execution. Tools offering binary instrumentation such as Pin [68], Dyninst [14], Valgrind [83], Pebil [65] are at the heart of code analysis tools used today. For performance analysis, a first coarse grained analysis is usually done in order to identify hotspots, and on these hotspots, a finer grained analysis capturing more details, follows. In order to adapt to the level of details required and to avoid the cost of an indiscriminate and expensive fine-grained instrumentation, several instrumentation languages have been proposed [38, 104, 80]. Instrumentation languages help to define where to insert the instrumentation probes, based on the structure of the binary code in terms of functions, loops, and sometimes blocks or instructions. However, compiler optimizations may change deeply the structure of the code, from the source to the binary, and this limits the effectiveness of such approaches. Indeed, when a function "foo" has been inlined or cloned by the compiler as it occurs for OpenMP codes, current instrumentation techniques cannot instrument the inlined versions of "foo" (no longer functions), nor relate instrumentation results of cloned versions to the "foo" function. Likewise, if the compiler generates two versions of a loop, one vectorized and the other not, the binary instrumentation techniques proposed only report performance analysis for each loop, independently of the other. For optimized codes, the main challenge for instrumentation languages is not only to enable an efficient description of the code fragments to instrument but also to report information relevant for users.

In this chapter we present a domain specific instrumentation language, MIL, for the development of code analysis tools. This language is built on top of MAQAO, a static performance analysis tool[69] and uses its binary rewriting framework. The original contributions of our approach are:

- A low-overhead instrumentation: We combine techniques presented in Dyninst[14] with more aggressive techniques for adding instrumentation code. This makes precise timing of short loops possible detailed in Section 5.6.4, outperforming current instrumentation frameworks and `intel` compiler instrumentation.
- MIL, a versatile language for instrumentation: the language MIL can be used to gather information on large variety of events, from functions to loops, blocks and instructions for control-flow profiling or value-profiling. The probes inserted in the binary code can be user-defined, enabling for instance hardware

counter profiles, and written either in MIL (MILRT runtime), in C (through an external library) or in assembly. Besides the precise location of these probes and their parameters can be defined by scripts, using a rich API of static analysis. In particular, probes can have parameters dependent on some static property of their insertion location.

- A framework for optimized multi-threaded code analysis: compiler optimizations can generate functions for OpenMP codes with complex control-flow. MIL enables the developer of code analysis tools to focus on functions appearing in the source code, independently of any name mangling, inlining or transformation due to OpenMP directives.

The chapter is organized as follows: Section 4.2 presents related work on instrumentation techniques and languages. Section 4.3 and 4.4 describe how the binary code is restructured and the instrumentation language is described. Section 4.5 shows how MIL was integrated into the TAU parallel performance system and proposes new analyses for multi-threaded codes. Finally, section 4.6 details the evaluation of the overhead due to instrumentation and a case study for performance tuning using our tool.

4.2 Related work

Concerning instrumentation techniques, we propose an approach very similar to what is presented in Dyninst [14], Pebil [65] or Saxena *et al.* [102]. More precisely, Dyninst uses two trampolines (two branches) before reaching the instrumentation code. This is due to the capacity of Dyninst to add/remove at runtime instrumentation code. We choose instead to have a static approach, able to insert code with low overhead. Work of Saxena *et al.* [102] uses an offline approach for binary disassembling and a back-end based on nasm assembler for generating machine code for new instructions introduced during rewriting. With MIL, all assembly instructions added or modified are directly modified in the binary form (no textual representation). This enables assembly binary instruction modifications and injection. Comparing Pebil [65] and our work, Pebil focuses on providing a way to insert code snippets (avoiding a call in the trampoline) and minimizing context saving. Besides, instrumentation codes can be inlined using function relocation. MIL offers similar mechanisms with the nowrap option, reducing context saving, direct inlining of user-assembly functions and function relocation (for 1 byte blocks to instrument).

For OpenMP performance analysis, POMP [100] proposes a performance monitoring interface for OpenMP. Tools implementing this interface, such as OmpTrace[19] based on Dynamic Probe Class Library[27] or Opari[77] are based on source to source modifications, with the inherent known limitations: instrumentation may prevent compiler optimization, the code analyzed may not be not the code the user wants to analyze. In INTONE [7], OpenMP directives are directly instrumented by the compiler. The approach taken with MIL is to provide instrumentation able to capture per thread information. OpenMP parallel loops, OpenMP API functions can therefore be instrumented and results of the instrumentation can expose OpenMP parallel execution to the performance system. Besides most compilers implement directives by inserting calls to the runtime. This is dependent of the OpenMP runtime but can provide a larger implementation of the POMP inter-

face. Going further and capturing runtime decisions (size of chunks for instance) is not handled by MIL so far.

Finally, several instrumentation languages have been proposed in the past [38, 104, 80]. Atune-IL instrumentation language corresponds to pragma inserted in the source code. These pragmas are then handled by the instrumentation tool. This approach enforces recompilation of the application. Mussler *et al.* propose an instrumentation tool driven by configuration files. The filters used in their instrumentation language do not handle blocks or instructions and value profiling does not seem possible, as it is in our tool. However, they propose predefined filters. These may apply to a limited class of applications, but selecting *a priori* which parts of the code are of interest is in general intractable. This is not the approach taken here in MIL. We propose user-defined filters, introduced as a more generic and complementary approach to predefined filters.Besides, their tool is based on Dyninst for binary rewriting.

4.3 Instrumentation Language

MIL is a scripting language to define how to instrument a binary code. Running this script through MAQAO with an input executable produces a new instrumented executable. The possible locations where instrumentation can be inserted are called events and the description of what to instrument corresponds to event filters. The definition of the *probes*, i.e. the code to insert, is also given in MIL. We present thereafter how to express both in MIL.

4.3.1 Abstract code structure and Filters

To define instrumentation points, the following structural abstractions can be manipulated: the program itself, its functions, loops, basic blocks, instructions and a particular case of instructions, the call sites. These notions are usual structurations of programs and they are computed by MAQAO static analysis. Loops correspond to only natural loops, and functions may have multiple entries. To define precisely where to insert some instrumentation, we define the notion of *event* as being a particular location in these structures, as for instance the entry and exits of a loop. The events associated to a particular structure are summarized in Table 4.1.

	program	**functions**	**callsites**	**loops**	**blocks**	**instructions**
Events	entry, exit	entries, exits	before, after	entry, exits, backedge	entry, exit	after, before

Table 4.1: Structural abstractions and associated events.

There are two specific events that only apply to the main binary, namely, constructors and destructors. The *at entry* event is triggered when the program starts, *at exit* when the program ends. We qualify these two events as special because 1) they happen only once, and 2) there are technical considerations due to operating system implementation restrictions. The *program entry* event provides a means to avoid the UNIX *LD_PRELOAD* mechanism that does not work with statically linked applications. This is necessary to avoid any potential conflict with a user-defined library that would need to be loaded with that mechanism. In the same way,

the *program exit* event permits bypassing the limitation of the system exit handlers (e.g., maximum number of calls). In both cases, it is not possible to pass arguments to calls that are inserted. Our program entry and exit events are not subject to this limitation. Technically, instrumentation is added to `_init` and `_fini` that are present in ELF binary files. This both work for dynamic and static binaries.

	program	**functions**	**callsites**	**loops**	**blocks**	**instructions**
Whitelist, blacklist	name	name	target name	id	id	address
Built-in				depth innermost, outermost		
User	user-defined function					

Table 4.2: Structural abstractions and associated filter mechanisms.

The structures defining the events are described in a hierarchical way, reflecting the nesting of functions, loops, blocks and instructions (Figure 4.1). In MIL, structures are nested tables with the LUA syntax[88], as described in Figure 4.2.

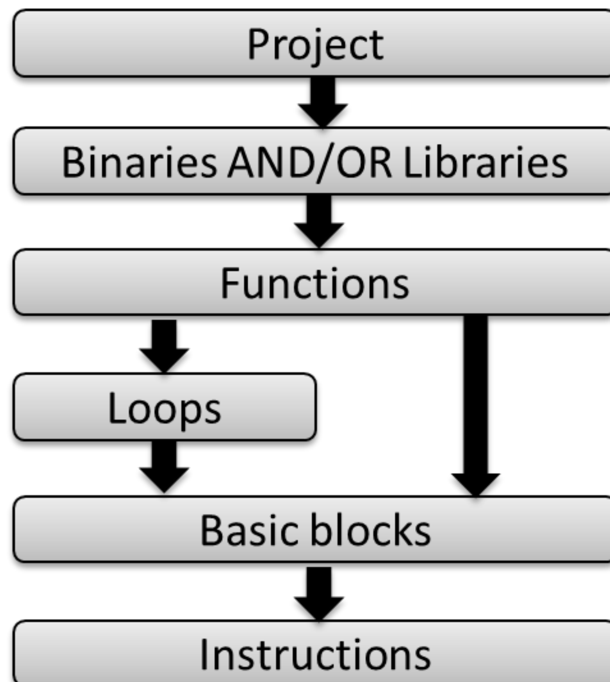


Figure 4.1: Hierarchy of the structural abstractions defined in MAQAO

Instrumentation overhead is one of the dominant concerns when considering how a binary rewriting tool is used to enable performance measurement. There are two ways to reduce the overhead of instrumentation at binary level: reduce the time taken by the probes themselves or reduce the number of structures instrumented by applying filters. In order to be able to restrict the field of objects to be processed, a filtering mechanism is mandatory. The filtering mechanisms associated to a particular structure are summarized in Table 4.2. A set of filters may be defined for every structure and can be either a list, a built-in filter or a user-defined filter. A filter using lists can define whitelists and blacklists for any structure. Depending upon the given structural object, its main attribute is used to apply the filtering. For

instance, the filter defined by `{whitelist = "^calc"},{blacklist = "_test$"}` selects all functions with name beginning with `calc` that do not end in `_test`. For callsites, the filter `{whitelist = "^calc"}` selects all instructions calling functions beginning with `calc`. It is also possible to select a set of structures based on their structural attributes (built-in). For instance, at the function level, it is possible to select loops at given depth, or innermost loops. For example, loops can be filtered with the predefined filters `innermost` and `outermost`, and also according to their depth ({depth = 3}).

```
 <structure-name> = {
   { filters      = { { <filter-specification> }, ... },
     <event-name> = { { <action-specification> }, ... },
     ...            // other events and actions
   },
   ... // other events and filters,
       // or nested structures
}
```

Figure 4.2: Definition of an event with its action.

Moreover, a special attribute, `user`, exists for all structures. This attribute corresponds to a user-defined boolean function that evaluates to true only if the structure should be considered for instrumentation. User-defined filters provide more flexibility when simple filters fail to identify precisely the code fragment to instrument. These functions are written in LUA. These functions are for more advanced filtering and can manipulate the structure through MAQAO API (in LUA).

4.3.2 Complex instrumentation queries

Events and pattern matching filters should suffice for common instrumentation needs. For more advanced instrumentation, MIL supports actions. The idea behind actions is to provide developers with a means to write functions that will be able to manipulate structural abstract objects associated to a given event. The function can then manipulate the object to perform a variety of queries and operations. Given an object, MIL is able to provide an access to the internal MAQAO Framework API though the MAQAO Lua API. It is then possible to directly interact with any existing modules as, for example, the low level binary rewriting layer, the abstraction layer, etc. It is beyond the scope of this chapter to describe how this occurs internally. Detailed documentation is available on the MAQAO website [69].

4.3.3 Instrumentation Probes

After having selected target instrumentation locations through events, it is possible to describe the probes to insert into the binary. It is possible to define the probes either in LUA, or to provide the name of the probe function with the name of a shared library containing it, or to define a string with inlined assembly code. The first method, defining probes in LUA, enables MIL to define in only one file everything needed for the instrumentation. In this case a call to this function, through LUA interpreter is inserted in the binary and the script of the function is appended to the binary. As many calls to an interpreter may generate large

overheads, the LUA JIT (just-in-time compiler)[90] is added to the binary instead of the interpreter.

External calls to pre-compiled libraries containing the probes have been used by other tools, such as Dyninst [14]. Concerning the insertion of assembly text, we propose a gcc-like inline assembly that handles loops and global variables. Note that both external calls and inline assembly can be used at the same time for the same instrumentation point. When inserting a call to an external function, it is possible to disable context saving. This may be useful when the inserted function already saves and restores all the registers it will be using. Even if most of the users will only use the insertion to external calls or LUA functions, it is important to be able to insert assembly code because it enables significant optimization opportunities, as described in the next section.

Given an event, any number of probes can be inserted. For each event, the following attributes can be specified:

- Inline assembly code to insert before other probes,
- Inline assembly code to insert after other probes,
- `nowrap`, to avoid saving the current context,
- Library containing the function to be called,
- Name of the function, for functions defined in libraries or in LUA,
- Parameters of the called function if any,

The available parameters types are:

- Immediate
- String
- Global variable: Global variables are declared at the beginning of the specification file and can have default values (Immediate or string). They are usually used to store the return values of inserted functions and then passed to others.
- Macro: Macros are predefined functions that enable access to values from within the instrumentation process. Each event has a set of available macros. If we consider for instance the entry function event, we could retrieve the starting and ending source lines of the instrumented function, the address of the insertion point, a unique identifier for the instrumented function, and so on.
- Memory: the default behavior is to return the target address of the instruction (of a jump, a load or a store for instance). This enables instrumentation of memory streams or capture complex control flow in case of indirections. But it is also possible to get the value pointed by the target address by specifying an additional option. This type of parameter is only available for instruction level events.
- User defined function.

User defined functions enable passing any value computed statically from the analysis of the binary to the probes. Note that while the probe is executed during the execution of the application, the evaluation of the parameters of these probes is at instrumentation time. These functions receives a reference (object pointer) to the instrumented structures in order to perform a variety of queries and operations. Given a structure, MIL is able to provide an access to the MAQAO Framework API (in LUA) that appears in Figure 4.3. It is beyond the scope of this paper to describe how this occurs internally. Detailed documentation is available on the MAQAO website [69]. Such user-defined parameters can depend on the instrumentation site and can be used to pass the information to the probe that the current loop is vectorized, unrolled, etc.

4.3.4 Using MIL to reduce instrumentation overhead

The first step towards reducing overhead is to limit the number of code fragments to instrument. The filtering mechanism proposed in Section 4.3.1 proposes an simple way to instrument only some parts of the code. Other analyses may require more elaborated filters. Mussler *et al.* [80] uses a predefined group of structural properties on the code as filters. Predefined filters may apply to limited class of applications, but selecting *a priori* which parts of the code are of interest is in general intractable. We propose user-defined filters, introduced as a more generic and complementary approach to predefined filters.

Optimizing instrumentation time also impacts the way the instrumentation is inserted into the code. Inserting a function call in a binary application has a cost, namely, the call instruction itself and the instructions to save the context before the call and restore it after. Inserting assembly instructions instead of calling a function removes this overhead. It may seem an extreme optimization, but it could be effective in cases where an instrumented routine contains loops that themselves call other functions. Although this kind of optimization requires architecture-specific considerations (even if some concepts are generic), it could reduce significantly the cost of inserted instrumentation calls. We will show later how this can enable us to measure more precisely and with less overhead very short loops.

4.3.5 Configuration and environment

MIL allows the user to control its default behavior by setting global properties and environment variables. Global properties can, for instance, selectively turn on/off each level of instrumentation (enable specific instrumentation objects without commenting parts of the instrumentation file) or change blacklist and whitelist order (which one has priority). Thanks to environment settings, it is possible to set output and run options such as the output folder, instrumented binary name, launch script containing environment variables, and parameters for the binary. The aim of this feature is to simplify the process of executing the instrumented binary. You can also configure MIL so that it does or not operate a distinction between interleaved functions and the function actually holding them. In case a distinction should be done, a suffix can be appended to the name of the function containing them in order to be easily distinguished.

4.4 Instrumented Code Generation

The instrumentation language is developed as a new module for MAQAO [69]. MAQAO is a framework for analyzing and optimizing binary codes and it combines binary disassembly, rewriting, and assembly with analysis to identify code semantics and reconstruct control flow. The framework relies on usual compiler algorithms to detect functions, loops and basic blocks. With this information captured in the abstract representation, MAQAO instrumentation component can be implemented. Besides, a module of MAQAO, named MADRAS, performs disassembly and assembly of x86 code. A very low level API for instrumentation is also part of MADRAS, that considers only instructions (no loops, functions, or blocks).

Figure 4.3 shows the components of the instrumentation language and its integration in the MAQAO framework. Blacks arrows describe the components involved in the basic workflow of MIL. Gray arrows depict the additional possible interactions with the MAQAO framework.

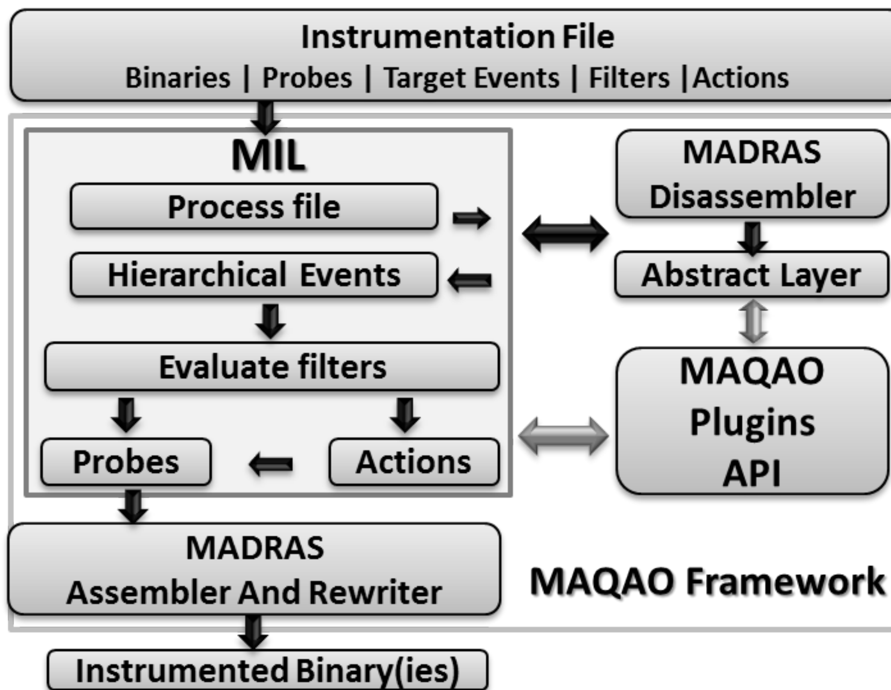


Figure 4.3: MIL : MAQAO Instrumentation Language and its integration in the MAQAO framework.

4.4.1 Static binary instrumentation

Several approaches have been considered for instrumentation. In contrast to source code instrumentation (such as proposed in Opari[77], DPCL[27] for instance), or instrumentation that operates at an intermediate language level, binary analysis and instrumentation starts with the program code in its final executable form. Source instrumentation, while flexible, has the disadvantage of requiring recompilation of the application. Besides, the modification of the code can alter the effects of compiler optimizations. Working with the binary code avoids recompilation and preserves any optimization performed by the compiler. However, it does present additional

challenges that might have to be overcome to deliver a robust instrumentation solution. Below we discuss the issues that arise, the different alternatives, and our approach.

In general, static binary rewriting has two important advantages compared to dynamic instrumentation. First, because the whole executable code is available when inserting instrumentation, static rewriting is more robust, able to perform more instrumentation requests, and can implement optimization methods more easily. Second, instrumentation occurs only once and before execution. Subsequent runs of the program will include the instrumentation.

We have developed the MADRAS (Multi Architecture Disassembler, Rewriter and ASsembler) tool to support binary code analysis of x86-64 executables. MADRAS is able to disassemble a binary file in ELF format and return a sequence of structures containing information about the assembly instructions it contains. In particular, this information determines whether an instruction is a branch or not, the size of its operands and whether the operands are written or read. The MADRAS disassembler performs a linear sweep to retrieve the assembly code of a binary file. It is also able to retrieve debug information to complement its disassembly results.

MADRAS patches executable files through binary rewriting. It can insert function calls or assembly instructions, delete instructions, or modify them by changing their opcode or operands. Inserted function calls can be wrapped with instructions for saving the context (contents of registers and stack frame) and restoring it after the inserted call, thus ensuring that the execution of the inserted function is transparent for the executable. MADRAS is also able to insert global variables and reference them in inserted or modified code.

4.4.2 Advanced static analysis

Multi-threaded and optimized binary codes can present some specificities that introduce challenging binary analysis problems. In the analysis of applications we encountered several major issues,in particular: handling indirect branches, exit handlers, interleaved functions, inlining and probe insertion in some cases. Our solutions to these issues are discussed below. To illustrate our explanations, we will be using *bt* and *dc* (class A) benchmarks from NPB-OMP3.3 [3], and 312.*swim* benchmark (Medium) from SPEC OMP 2001 [5], all compiled by the Intel Fortran compiler (ifort) with -O3 optimization.

4.4.2.1 Indirect Branch Resolution

There is a major issue with indirect branches in that they can hide exits of functions. In order to obtain the complete set of exits of a function, we need to resolve indirect branches within functions. We introduce the concept of conditional probes. It consists of a regular probe combined with a set of conditions. The core idea is to set a condition on the target of the indirect branch once resolved. When considering function boundaries, the condition holds on the set of intervals that describes the limits of the function in terms of addresses. If the target is outside these intervals, then it is an exit and the probe would be executed. This algorithm is implemented internally and requires no input from the users. If desired, users can disable indirect branch resolution in the configuration part of their MIL script.

4.4.2.2 Exit handlers

A call to the system function *exit* must be considered as an exit of the function containing it. We support a user-defined list of functions that should be considered as exits and can be adapted from one system to the other. These exits are flagged as returns in the abstract layer.

4.4.2.3 Multiple entry functions

We think that there are two main approaches when considering how to detect the entries and exits of a function.

- On the one hand, a function can only have a unique entry. When, for some reason, two functions share some basic blocks, then a copy is present in each function.

- On the other hand, a function can have multiple entries. Any basic block belongs to only one function.

We have selected the second approach because it has two advantages, namely, handling external pointers (not jumping on the main entry block) and keeping the code structure unchanged. Besides the main entry of a function, we consider every block of the CFG that has no predecessor as a potential entry. In general, numerous entries are the consequence of indirect branches coming from within the function or branches (or calls) from other functions. We will detail the latter case. Most of the time, when a function is called, the call destination is the first instruction of the entry block of that function. But there are two exceptions to this rule.

- sometimes functions may share some blocks due to specific optimizations. Lets call $F1$ the function containing the shared block and $F2$ the other one. This kind of sharing adds a new exit to $F1$ and a new entry to $F2$.

- indirect branches or calls may point a different block than the natural entry block.

In order to properly handle functions that have at the same time multiple entries and contain indirect branches, we use a special method. We first insert the requested probes in a new basic block preceding every existing entry blocks. Exit blocks are instrumented without adding any new basic block. For every indirect branch, we use special insertions (conditional) that jumps at the entries of the function. Adding a new basic block implies updating all branch instructions in the function that points on the block where probes must be inserted.

With this method (inserting new basic blocks for probes), we also handle the case where an entry block of a function is the entry of a loop. There is no natural predecessor block, but since we a add a new basic block, this case is supported.

In the *dc* benchmark, we can find the *"CalculateVeiwSizes"* function that contains two indirect branches that jump to one of the entries of the same function. With our method, no unnecessary entry event is triggered since they are not executed.

4.4.2.4 Interleaved functions

At source level, it is relatively straightforward to identify the structure of a function and functions have one entry and multiple exits (returns). While exits can be a little tricky to instrument in source, when we consider the general problem at binary level, optimizations achieved by the compiler may produce a more complex code structure. In binary, functions are only labels and it is even possible for two functions to share common blocks (due to compiler optimizations). These *interleaved* functions make the abstraction of the code more complex to handle and are generated for instance by the Intel compiler for OpenMP codes. When it comes to instrument a specific function, specific measures have to be taken.

To detect interleaved functions, we apply a connected component search on the control flow graph (CFG) of a given function, in our static analysis phase, and make the interleaved functions appear as separate components. If we consider the *bt* benchmark, the multi-threaded part of the code in functions containing OpenMP directives (i.e., that part that will be called by the OpenMP runtime) is inlined. Figure 4.4 reveals a part of the control flow graph of one of the most time-consuming functions of the *bt* benchmark after MAQAO binary analysis. MAQAO has successfully separated each component of the CFG. MIL default behavior is to consider each component as a regular function. The name of the function is the same as the container function concatenated with a unique suffix and may be different when, for instance, inlining is detected.

Let us consider a more complex example with the *swim* benchmark. This application contains four main (most consuming time) functions called from the program entry function. Taking a closer look at the code, we observe that three of these functions are actually inlined in the *main* routine. The inlined functions are called from the OpenMP runtime and entry/exit points are merged with the ones of the *main* routine. If only the *main* routine entry and exits points were instrumented, we would miss accounting for the three inlined routines. In fact, basic time profiling methods show only *main* and the routine not inlined as the two dominant time-consuming functions. The connected component analysis of MAQAO can discover the inlined functions and correctly apply function level instrumentation. The most important point of this approach is solving the problem using a static analysis, which is essential to reduce the instrumentation runtime overhead.

4.4.2.5 Inlined functions

In the previous paragraph, we considered the *swim* benchmark example where functions where inlined. In the OpenMP class codes, that's what usually happens because the multi-threaded part of the code is actually called by the OpenMP runtime. As far as Intel compilers are concerned, the starting address of the multi-threaded code executed is a pointer passed as a parameter of the function of the runtime library which is responsible for calling that code. In general, detecting inlining is at least a challenging task and may be impossible at binary level. In MAQAO, we added a new heuristic that uses debug symbols, when available, to detect inlined functions. The instructions of functions that are inlined have a specific source line, the call site source line. Given the CFG of a function, we look for subgraphs with basic blocks that have a majority of their instructions that have that property. If we consider again interleaved functions, this heuristic works almost every time and helps with figuring out the name of the inlined function. Actually the name is given

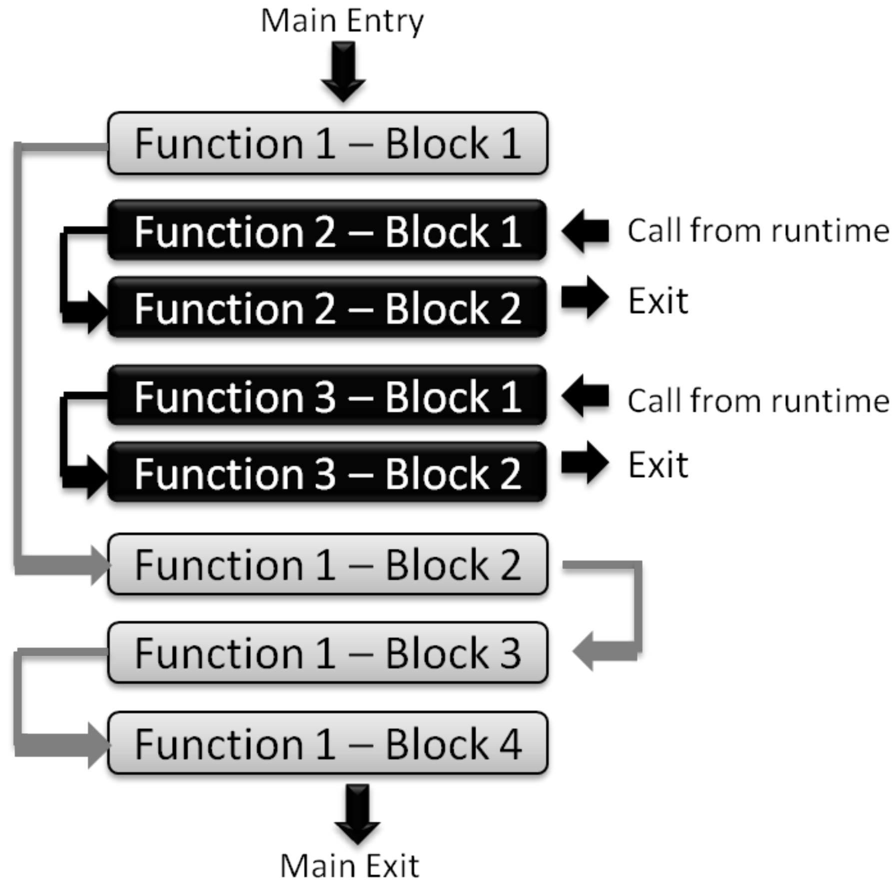


Figure 4.4: Part of the CFG of the main function (MAIN__) from the bt benchmark (using Intel fortran compiler with -O3)

by the destination function of the call site.

4.4.2.6 Probe insertion issues

We introduce the concept of conditional probes in order to control the conditions under which a probe should be executed or not. After inserting a probe, a set of conditions can be applied on it. Thanks to this approach, it is possible to solve challenging issues.

One recurrent concern when dealing with instrumentation, without naively considering the relocation of a whole function as a global solution, is the ability to insert probes wherever the users ask for. Indeed, that is not always possible at reasonable cost. For instance, on x86-64 architecture, the *dc* benchmark contains 11 functions where there is insufficient space for probe insertion is observed. The common workaround is the concept of trampolines. It is used to find the required space close to the instrumentation site. It works most of the time. But there is one case where trampolines cannot solve the issue: one byte instructions (isolated return instructions). Indeed, trampolines need at least two bytes instructions. Therefore, when trampolines cannot be inserted (due to a lack of place) or instructions are too short, the only existing technique is to resort to a trap instruction which has a size of one byte. The induced overhead is unfortunately huge (factor of 15 compared to a regular probe).

We use a new algorithm that solves most of these cases, including the issue observed in the *dc* benchmark. Algorithm 1 details our approach. We perform a control flow analysis to figure out the predecessors of the current block where instrumentation should have taken place and instrument them. We go through these predecessors and verify that there is enough space to insert probes. If not, we have no choice but to insert a trap instruction. If there is enough place in all the predecessor blocks, then we have to determine for each of them if their target is the current insertion block or not (where the probe must be inserted). One case is quite complex, when considering a conditional branch. Since we only can insert the probe before it, we must add a condition so that the probe is only executed if we are sure that the flow is going to the current instrumented block. When considering the previous example, that means when branching to the exit block of the function. Figure 4.5 presents the solution to that case.

```
input  : probe to insert
output: SUCCESS or FAILURE
inst ← GetInsertInst(prob);
block ← GetBlockFromInst(inst);
if IsSmallBlock(block) then
    predBlocks ← GetPredBlock(block);
    foreach pb in predBlocks do
        if IsSmallBlock(pb) then
            Insert a trap instruction (INT3)
        end
        LIB ← GetLastInstOfBlock(pb);
        if InstIsCall(LIB) then
            ProbInsert(inst,prob,AFTER);
        else if InstIsUncondBranch(LIB) then
            ProbInsert(inst,prob,BEFORE)
        else if InstIsCondBranch(LIB) then
            brTargB ← GetBranchTarg(LIB);
            if brTargB == block then
                BC ← GetOppositeBranchCond(LIB);
                CV ← ExtractCompareVal(BC);
                ProbCondInsert(inst,prob,CV,BEFORE);
            else if brTargB == block then
                ProbInsert(inst,prob,AFTER);
            end
        end
    end
end
```

Algorithm 1: InsertProbe

In a nutshell, our method minimizes, and even removes, the number of trap instructions needed to correctly instrument a function. Hence, a huge performance gain.

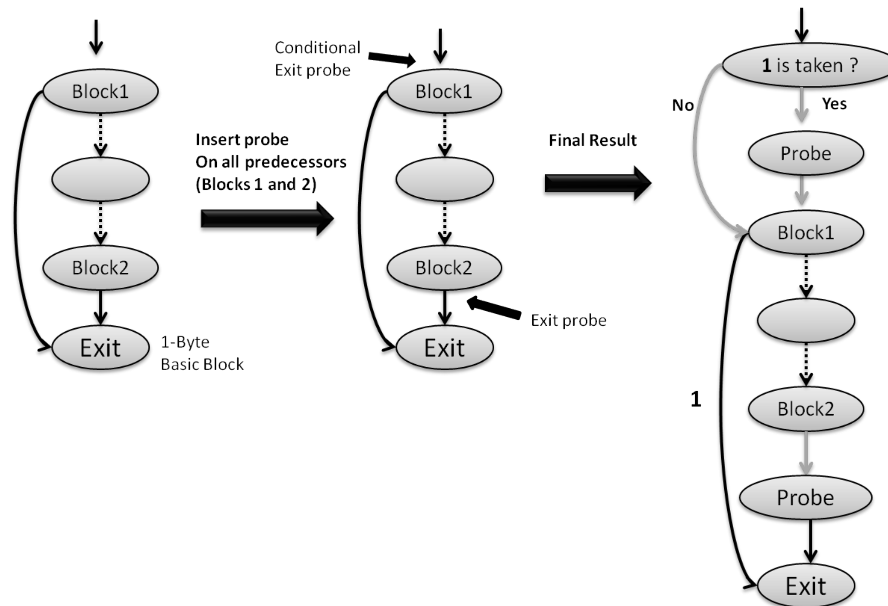


Figure 4.5: More efficient solution than int 3 trap handler on x86-64 architectures

4.5 Building Performance Tools

A systematic performance analysis approach must adopt a measurement methodology where critical performance bottlenecks can be identified at a coarse level and then instrumentation at a finer level can pinpoint more precise performance issues. The challenge is to create a performance analysis system that supports both flexible instrumentation that preserves code properties relevant to the user and lightweight performance measurement that keeps overheads to a minimum.

4.5.1 Integration in the TAU Performance System

The TAU Performance System [109] from the University of Oregon is a performance evaluation toolkit that supports several instrumentation, measurement, and analysis alternatives. TAU presents a good target to prototype a MIL-based instrumentation tool because it has challenging requirements and it offers wide opportunities for applications. We created a tool, *tau_rewrite*, to add instrumentation to binary files and dynamic shared objects (DSOs). The tool permits a user to inject a specified TAU measurement library while rewriting the executable. Our goal in integrating TAU with MIL was to simplify the usage of TAU and create an efficient binary rewriter for multi-threaded applications. Besides binary instrumentation, TAU also supports source-level instrumentation using the Program Database Toolkit (PDT) and OPARI-2, the OpenMP source restructuring toolkit. Being able to compare different instrumentation methods in a single measurement and analysis system was another reason for selecting TAU.

Figure 4.6 shows the instrumentation file in MIL for the TAU performance tool. This is all that is necessary to enable binary instrumentation for the TAU performance measurement demonstrated below. The code achieving the same functionality, for Dyninst, requires around 200 lines of code.

Figure 4.7 depicts a simple standalone profiler completely written in MIL using embedded probes. The aim here is to show that we can easily and quickly implement

```
run_dir = "/PATH_TO_OUTPUT_FOLDER/",
at_exit={{
    name = "tau_dyninst_cleanup",
    lib = "libTauHooks.so"
}},
main_bin = {
  properties={
    enable_function_instrumentation = true,
    distinguish_interleaved_functions = true,
    distinguish_suffix = "_omp"
  },
  path= "/PATH_TO_main_binary",
  output_suffix = "_i",
  envvars="LD_LIBRARY_PATH=/PATH_TO_tau_library/",
  functions={{
    entries = {{
      at_program_entry = {{
  name = "trace_register_func",
  lib = "libTau.so",
  params = {
    {type = "macro",value = "fct_info_summary"},
    {type = "macro",value = "profiler_id"},
  }
      }},
      name = "traceEntry",
      lib = "libTau.so",
      params = { {type = "macro",value = "profiler_id"} }
    }},
      exits = {{
      name = "traceExit",
      lib = "libTau.so",
      params = { {type = "macro",value = "profiler_id"} }
    }}
  }}
};
```

Figure 4.6: TAU instrumentation file using MIL.

a performance tool without having to actually manipulate complex data structures.

Performance analysis of parallel applications requires the ability to generate performance measurements in the form of profiles and traces. The rest of this section will show how it is possible to build evaluation tools based on specific needs that are not possible to build easily with other frameworks. We present a few scenarios to show the flexibility of our approach.

4.5.2 Loop centric profiling

Depending on the methodology one wants to implement when looking for performance opportunities, it may be interesting to target a dynamic profiling based on the analysis we are able to do. In our framework, we have modules that can only work with simple (one block) inner-most loops. For instance, the static loop analyzer detects the degree of vectorization. It supports user-defined filters to exclude

```
--## Runtime code section ##
milRT.meta_info = {};
milRT.results = {};
milRT.myfreq = 2799489000;
-- Get current clock cycles
function milRT.timer() return timer() end
-- Gathers meta information initialize structures
function milRT.register_function(fct_name,fid)
  milRT.meta_info[fid] = fct_name;
  milRT.results[fid] = {start = 0,inc_time = 0,calls = 0};
end
-- Start counting time at function entry
function milRT.fct_start(fid)
  milRT.results[fid].start = milRT.timer();
  milRT.results[fid].calls = milRT.results[fid].calls + 1;
end
-- Stores/Accumulates time at function exit
function milRT.fct_stop(fid)
  milRT.results[fid].inc_time = milRT.results[fid].inc_time +
  os.difftime(milRT.timer(),milRT.results[fid].start);
end
-- Show profiling results
function milRT.fct_dump()
  print("Simple profiler results");
  print("Function name\t| Calls \t| Inclusive time");
  for id,result in pairs(milRT.results) do
    fct_name = milRT.meta_info[id];
    print(fct_name.."\t"..result.calls.."\t"..
         (result.inc_time/milRT.myfreq).." seconds");
  end
end
-- Summarize a given function information into one string
--## Events sections ##
--Static functions invoke from probes
function fct_info_summary(func)
  local fname = mil:fct_main_attribute(func);
  local fsrcf = finsn:get_src_file_path();
  local fct_start,fct_stop,fct_info_summary;
  fct_start,fct_stop = func:get_src_lines();
  return fname..
  " [{"..fsrcf.."} {"..fct_start..",0}-{"..fct_stop..",0}]";
end
--functions present in Lua Runtime Environment
declares = {
  {cname = "get_rdtsc",clib = "libmilrt.so",luaname = "timer"}
};
--Events table
local mil_out_path = "OUTPUT_FOLDER";
events = {
  run_dir = mil_out_path,
  at_exit= {{ name = milRT.fct_dump }},
  main_bin = {
    properties={
      enable_function_instrumentation = true,
      distinguish_interleaved_functions   = true,
      distinguish_suffix = "_omp",
      enable_runtime = true,
    },
    path = mil_out_path.."binary_name",
    output_suffix = "_iRT",
    functions={{
      entries = {{
          at_program_entry = {{
              name = milRT.register_function,
              params = {
                {type = "function",value = fct_info_summary},
                {type = "macro",value = "profiler_id"},
              }
          }},
          name = milRT.fct_start,
          params = { {type = "macro",value = "profiler_id"} }
      }},
      exits = {{
          name = milRT.fct_stop,
          params = { {type = "macro",value = "profiler_id"} }
      }}
    }}
  }
}
```

Figure 4.7: Simple profiler all in MIL

fully vectorized loops. With MIL, it is easy to target such loops. In addition, one may want to collect some hardware performance counters data for these loops. The Linux kernel has introduced the perf tool, exposing the kernel performance counters subsystem to user-space. Using the libperf [98] library at the entries and exits of

these loops is sufficient to collect information on the selected hardware performance counters. Further, it is also possible to achieve value profiling on loops. Collecting at runtime the number of iterations (an instance) is straightforward using the loop *backedge* event. Figure 4.8 reveals the distribution of instances in a form of a histogram. For instance, the 2^2 bar shows the number of instances of a loop that have a number of iterations between 2^2 and 2^3-1. When observing a high number of instances for small and huge numbers, it may be useful to resort to specialized versions.

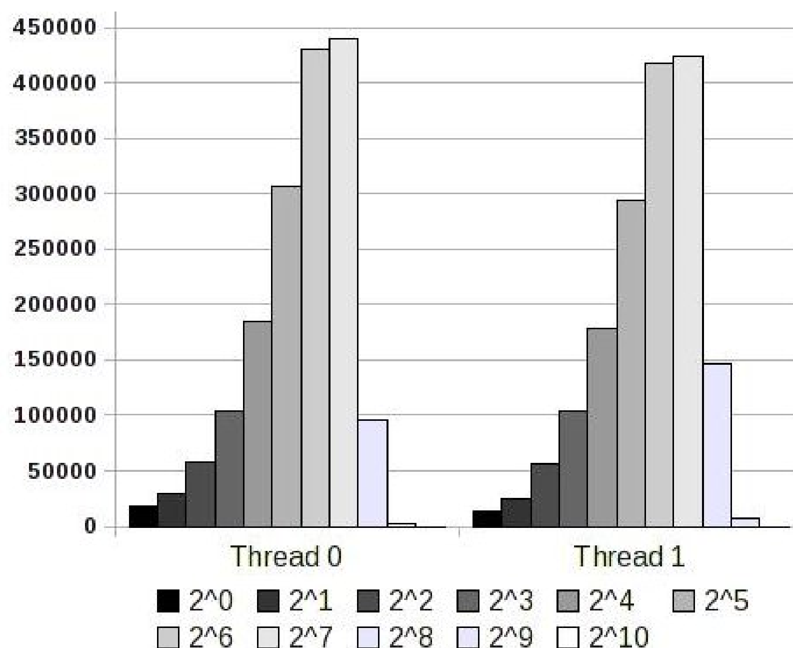


Figure 4.8: Distribution of the number of iteration per instance of a loop and per thread. Y axis reports the number of instances

Figure 4.9 shows the corresponding instrumentation file.

4.5.3 Memory tracing of OpenMP codes

Recently, one of our main concerns was to capture the memory behavior of OpenMP codes [9]. We used MIL to build a prototype to instrument load and store instructions, before investing time in the development of a standalone module. We created a small external library to collect these memory operation and print them. The MAQAO framework features a module that implements an induction variable detection algorithm. In conjunction with a proper user filter, we have been able to even optimize our prototype by guessing some memory values instrumenting only outer loops. That allowed us to reduce experiment collection times from 3 days to 2 hours on a forty-core. Tracing the memory addresses of inner-most loops is very time-consuming. Being able to guess the addresses at higher level in the loop nest dramatically reduced the overall overhead.

4.5.4 Dynamic extension of static prediction

In Chapter 3 we presented STAN, the static analyzer module of MAQAO. We saw that it works under the hypothesis that manipulated data is in the first level cache.

```
events = {
  run_dir = mil_out_path,
  at_entry={{
      name = "instrul_load",
      lib = "libinstrulight.so"
  }},
  at_exit={{
      name = "instrul_unload",
      lib = "libinstrulight.so"
  }},
  main_bin = {
      properties={
enable_loop_instrumentation = true,
      },
      path= mil_out_path.."lu.A",
      output_suffix = "_i",
      functions={{
  filters={{
      type = "whitelist",
      filter = { {subtype = "stringlist",value = {"jacu_","jacld_","buts_"}} }
  }},
  loops = {{
      filters = {
                {
  type = "builtin",
  filter = { {attribute = "nestlevel",value = "innermost"} }
},
{
  type = "whitelist",
  filter = { {subtype = "numberlist",value = {153,168,170}} }
}
      },
      entries = {{
  name = "instrul_loop_start_vp",
  lib = "libinstrulight.so",
  params = {{type = "macro",value = "id"} }
      }},
      exits = {{
  name = "instrul_loop_stop_vp",
  lib = "libinstrulight.so",
  params = { {type = "macro",value = "id"} }
      }},
      backedges = {{
  asm_before = "LEA -0x200(%RSP),%RSP\nPUSH %R10\nPUSH %R11\n",
  nowrap = true;
  name = "instrul_loop_countiters_vp",
  lib = "libinstrulight.so",
  asm_after  = "POP %R11\nPOP %R10\nLEA 0x200(%RSP),%RSP\n";
      }}
  }}
      }}
    }
}
```

Figure 4.9: MIL file for a loop value profiling example

The purpose of this dynamic extension is to provide a means to check the difference between the static prediction and the actual performance.

Because STAN works on loops, we used two different scripts to get the number of iterations and the number of cycles along with instances. Based on the results provided by these two MIL scripts we created a tool to compare static prediction and actual execution time of loops. This small tool is further described, along with its sources, in AppendixB.

4.6 Experiments

In addition to evaluating MIL from a functional standpoint, it is important to compare the quality of the instrumentation framework on real applications and against existing binary rewriting tools. The OpenMP NAS parallel benchmarks [3] are used for testing execution overhead. Then, a real-world example, QMC=Chem, is used to evaluate the difference between the ICC compiler integrated profiler and a profiler created with MIL.

4.6.1 Instrumentation Overhead on Parallel Applications

Parallel applications such as OpenMP codes are optimized and transformed by the compilers in a way that hampers static binary instrumentation. Besides usual compiler optimizations, parallel regions are transformed into new functions and called through function pointers.

In the following experiments, we compare the overhead of instrumented parallel NAS benchmark codes with TAU, using Dyninst [14], using MIL and using OPARI [77] tool, to allow a contrast with source-based instrumentation.

The following experiments are run on a dual socket six-core 2.27Ghz Xeon Westmere-EP X5650 (total of twelve cores) machine. The Intel Fortran compiler was used to compile the benchmarks and execute them with the OpenMP runtime. All benchmarks are compiled with the maximum optimization level (-O3) with debug symbols (-g). The TAU profiling measurements were the same in each case. Only the overhead of invoking the measurement code is different, and this is a result of how the instrumentation was done by each tool. MIL and Dyninst similarly use static binary rewriting. We have also configured MIL in order to separate the new functions generated for the OpenMP runtime, from the functions containing them and add an _*omp* suffix. OPARI's source-to-source translation automatically adds all necessary calls to the POMP runtime library, which interfaces with TAU to make performance measurements. All measurements could be viewed using the TAU *pprof* profile analysis tool.

Figure 4.10 summarizes the obtained results for a Class A run of the OpenMP NPB suite. MAQAO has a lower or equivalent overhead compared to Dyninst in all cases. Interestingly, *bt.A* and dc.A reveal an important overhead factor for all tools. For *dc.A*, the difference between MIL and Dyninst (a factor 7) comes from the different ways to handle one-byte basic blocks. We added instrumentation probes in all predecessors of the block, while Dyninst resort to the INT3 mechanism, much more expensive.

We were systematically expecting higher overheads compared to OPARI, but surprisingly we observe several cases where the binary approach outperforms the source-to-source instrumentation. We also see the potential problems that can arise with source-based methods not being able to handle all code cases. (A zero figure means that either the concerned benchmark failed to compile or crashed at runtime.)

After having studied the overheads, the next aspect we want to verify is the quality of the results. Figure 4.11 exhibits the output obtained with MAQAO and Dyninst profiling results for two threads (out of twelve) on the *bt* benchmark. For instance, for thread 0, both MAQAO and Dyninst correctly detect the three most consuming functions. Let us now consider the results for thread 1. Dyninst fails to display the function names. As mentioned before, since we are able to statically identify the new OpenMP functions, we can provide more accurate information to TAU. Since both tools find the same hotspots within the same proportions (roughly 32% for each dominant hotspot), we expect a higher number of instructions inserted at binary level by Dyninst. That is exactly what is happening since Dyninst uses a *trampoline* mechanism inducing multiple branches for one insertion. Our binary rewriting layer only inserts one level of indirection, directly adding instrumentation instructions to the displaced basic block without additional branches. Furthermore, since whole basic blocks are moved when adding instrumentation, our approach reduces the overhead when multiple instrumentations are to be performed in the

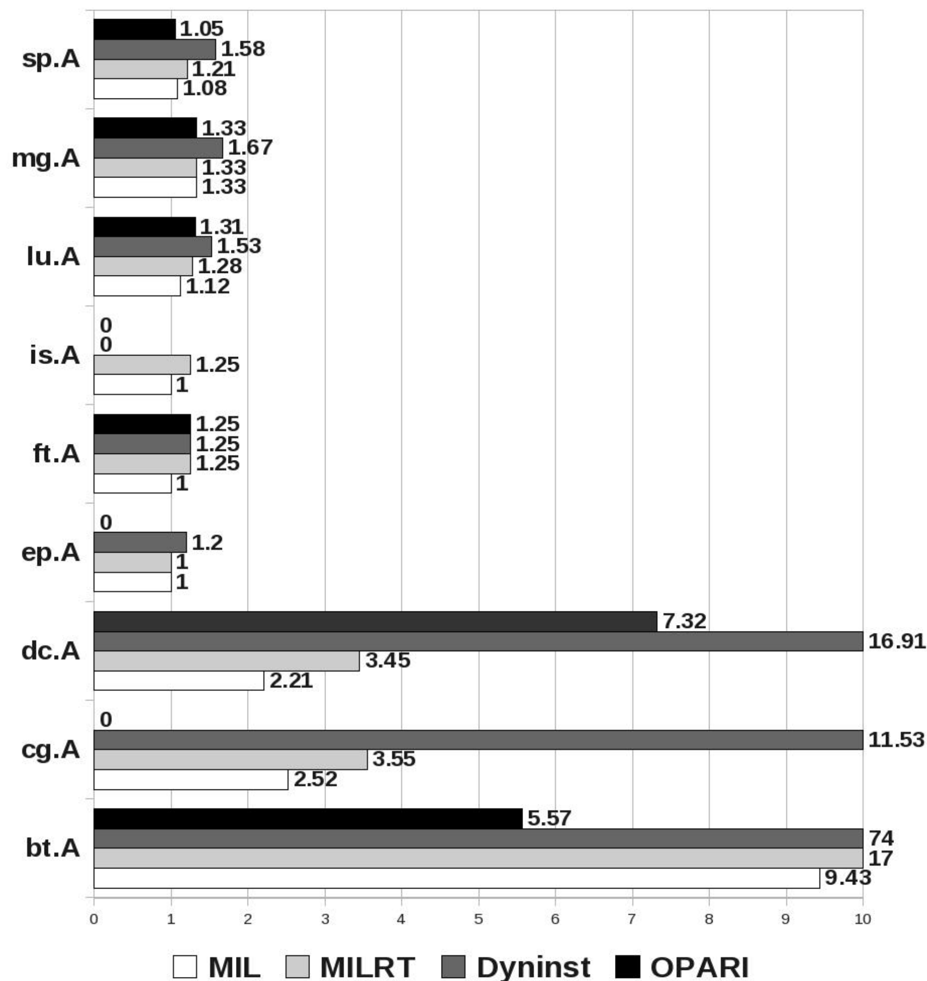


Figure 4.10: Comparing overhead time on NAS benchmarks for MIL with probes in external library, MIL with probes in LUA (MILRT), Dyninst and Opari using TAU. X axis reports the overhead ratio compared to the original run. Lower is better. Overhead ratios greater than 10 are cut. A zero ratio means that either the concerned benchmark failed to compile or crashed at runtime

same basic block. Observed results on the *bt* benchmark highlights this kind of case.

4.6.2 Real case example : QMC=Chem

QMC=Chem [103] is an application that uses Quantum Monte Carlo (QMC) methods to solve chemical problems. Due to its properties, it targets massively parallel machines.

Experiments concerning this application have been run on a single socket quad-core 3.30GHz Xeon Sandy Bridge E31240 machine. Sandy bridge architecture is a requirement because the application uses the latest SIMD instruction set (AVX). In order to verify our results, the only colleague having access to the source code used the Intel IFORT compiler. The Intel compilers can profile loops starting from the recent version 12. After a first profiling pass, two interesting observations where

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	0.244	1:07.801	1	1	67801041	main
100.0	1	1:07.800	1	232	67800797	MAIN__
99.8	1	1:07.656	201	1005	336599	adi_
32.5	21,776	22,012	201	174297	54758	x_solve_
32.3	21,745	21,914	201	122693	54513	y_solve_
32.2	21,697	21,835	201	103617	54317	z_solve_
2.5	1,670	1,670	202	202	4135	compute_rhs_
0.3	229	229	201	201	572	add_

(a) pprof tool output for thread 0 using MAQAO

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	0.27	9:30.290	1	1	570290086	main
100.0	2	9:30.289	1	232	570289816	MAIN__
99.1	1	9:25.004	201	1005	2810970	void adi()
33.2	3:00.363	3:09.113	201	201	940864	void y_solve()
32.8	2:58.219	3:07.058	201	201	930637	void z_solve()
32.7	2:57.994	3:06.689	201	201	928805	void x_solve()
1.5	8,762	8,838	201	136524	43974	void targ419ff9()
1.5	8,662	8,749	201	155905	43531	void targ4161f9()
1.5	8,578	8,695	201	207576	43261	void targ4146f8()
0.7	21	4,061	2	2	2030887	void initialize()
0.7	3,979	4,040	2	100001	2020153	void targ402c52()
0.3	40	1,805	202	202	8939	void compute_rhs()
0.3	1,765	1,765	202	0	8741	void targ40be37()

(b) pprof tool output for thread 0 using Dyninst

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1,164	1:07.796	1	1012	67796835	.TAU application
31.9	21,416	21,649	201	174096	107709	x_solve_omp
31.8	21,400	21,569	201	122492	107309	y_solve_omp
31.7	21,359	21,496	201	103416	106948	z_solve_omp
2.4	1,649	1,649	202	0	8167	compute_rhs_
0.2	156	156	201	0	776	add_

(c) pprof tool output for thread 1 using MAQAO

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	7,845	9:30.284	1	1012	570284826	.TAU application
32.6	3:04.814	3:05.867	201	155905	924715	void targ4161f9()
32.4	3:03.927	3:04.840	201	136524	919605	void targ419ff9()
32.4	3:03.162	3:04.547	201	207576	918145	void targ4146f8()
0.7	3,357	4,058	2	100001	2029312	void targ402c52()
0.3	1,763	1,763	202	0	8728	void targ40be37()

(d) pprof tool output for thread 1 using Dyninst

Figure 4.11: Comparing the pprof profiling tool output of MAQAO and Dyninst for two threads

made:

- calls to external functions are consuming 34% of the walltime.
- three loops have been discovered as the most time consuming. The estimated time of execution of an instance of these loops is around 300 cycles. Further steps would consist in optimizing these three loops and verify the gain.

The rest of this section will be dedicated to understanding how using methods seen in section 4.5 helped us in investigating these two aspects.

4.6.2.1 Short Loops Instrumentation

Codes containing small objects like loops that only last a few hundred cycles and executed repeatedly are hard to measure. Sampling is definitely not the path to follow. For x86 architectures, the only efficient way to study such objects is to time at cycle level using the RDTSC [47] instruction. Table 4.3 shows the differences between MIL and IFORT on cumulated cycles for the three target loops. IFORT has an integrated compensation feature to remove the additional instrumentation time introduced by their probes from the actual time. We can see that the accounted cycles for these three loops is higher than our measure without compensation. IFORT uses heavyweight probes that take more time to execute (around 350 cycles) than one instance of these loops. What is really measured on such short loops is not clear. The IFORT compensation feature only worsens the quality of the measure. In order to obtain a more accurate measure, we use lightweight probes. Our probe uses the *nowrap* attribute, which is actually directly written in assembly language and only uses two registers and three global variables. That is why we obtain a more accurate and less intrusive measure.

Compensation	Without	With
MAQAO	$134653 * 10^6$	$126273 * 10^6$
IFORT 12.1		$135486 * 10^6$

Table 4.3: Comparison between MAQAO and IFORT on the number of cycles measured for the three most time consuming loops

Moreover MAQAO has a lower overhead since it can use filters to target only these three functions. IFORT can only filter on a per-file basis and systematically enables function profiling. Table 4.4 reveals the higher overhead of IFORT over MAQAO.

	Original	MAQAO loop	IFORT light loop
Walltime	97	101	112
Overhead	-	4%	15%

Table 4.4: Comparison of execution times (in seconds) between MAQAO and IFORT

Thanks to our approach, studying these loops is cheaper and more accurate.

4.6.2.2 Value profiling on external functions

One of the worst cases one may face in performance analysis is finding that a call to an external library function is consuming a lot of time. In most cases, the external function is already optimized and nothing can be done. A good check consists in verifying the distribution of the values passed as parameters to the called function. Such a function has been identified in our studies application. Both important calls to the *exp* (exponential) function takes 34% of the walltime. When applying a value profiling targeting the parameter of this function, we observe that the same

parameter value is being used. In order to capture the parameter of the *exp* function, we placed a probe with the *nowrap* attribute before the callsite and inserted a call to an external library we defined. That way, our call would receive the same parameter, make a context backup, process the value, restore the context and go back to normal flow of execution.

% time	cumulative seconds	self seconds	name
25.58	7.59	7.59	sparse_full_mm_
23.26	14.49	6.90	__svml_expf8.R
10.85	17.71	3.22	__svml_sexp_cout_rare
7.31	19.88	2.17	bld_ao_oned_block_
6.10	21.69	1.81	bld_ao_oned_prim_block_
4.58	23.05	1.36	bld_ao_axis_block_
3.00	23.94	0.89	bld_jast_u1_simple_
2.43	24.66	0.72	expf.L
1.01	24.96	0.30	__intel_ssse3_rep_memcp

(a) Before

34.21	6.52	6.52	sparse_full_mm_
11.02	8.62	2.10	bld_ao_oned_block_
10.81	10.68	2.06	bld_ao_oned_prim_block_
6.51	11.92	1.24	bld_jast_u1_simple_
5.30	12.93	1.01	bld_ao_axis_block_
2.52	13.41	0.48	expf.L
1.99	13.79	0.38	mkl_blas_avx_dtrmv_in
1.84	14.14	0.35	__intel_ssse3_rep_memcpy

(b) After

Figure 4.12: Profiling results of the QMC==Chem application before and after transformation

Taking this issue into account, the algorithm was changed and is now able to avoid most of the calls to *exp* function. Figure 4.12 reveals that calls to the exp function have been so drastically reduced that it disappeared from the most time consuming functions set.

4.7 Conclusion

In this chapter, we presented MIL, a rich instrumentation language that reduces the complexity of writing performance analysis tools for high performance computing. Using static binary instrumentation that does not require additional compilation pass, MIL offers a rich interface to instrument parallel OpenMP applications for a large range of uses, from profiling of functions, hardware counter analysis to value profiling. MIL provides a filtering mechanism to instrument only some specific code fragments, from functions, loops, to individual assembly instructions, and different types of instrumentations can performed simultaneously on different code fragments. When using MIL runtime probes, all performance measurements are reported for

each thread independently. Besides, a scripting mechanism relying on an API for code analysis, offers the possibility to implement other approaches to performance analysis.

We believe MIL offers a way to implement and investigate new performance analysis techniques that can cope with the challenging complexity of performance tuning for multi-threaded applications. We have demonstrated the flexibility of MIL through the study of different scenarios, exploring different granularities and successfully integrating it in TAU performance analysis framework. Our instrumentation framework features a rich abstraction layer based on static analysis and a robust binary rewriting tool. Execution overheads have been evaluated on NAS Benchmarks and compared to Dyninst, a similar instrumentation framework, and OPARI, a specialized OpenMP source-to-source tool. They show that the instrumentation provided by MIL has a lower overhead. MIL is integrated into the MAQAO tool, it is self-contained and packaged as a standalone and open-source software [69].
In chapter 3 we described how static analysis is able to provide us with considerable feedback when tuning compute bound applications, in particular, by assessing code quality. When dealing with memory bound applications, we will be using our instrumentation framework to precisely understand the memory behavior of an application. The next chapter presents our work on the characterization of memory behavior.

CHAPTER 5

Memory behavior characterization

5.1 Introduction

From a programmer perspective, for OpenMP multi-threaded applications [24], thread interactions result from the choice of the data structures, from the expression of parallelism, from compiler optimizations and from runtime strategies. Deciding how to drive the compiler and runtime to obtain the best performance is quite a challenge, and it is difficult to focus on the performance limiting factor without a clear picture of thread interactions. For instance, enforcing some thread affinity, mapping threads sharing data on cores sharing caches leads to better performance only if the reuse distance is compatible with the cache size. Too many threads sharing the same cache can lead to cache trashing situations: data loaded or prefetched by other threads may evict some useful data. Some loop transformations, such as tiling, fusion, loop distribution or interchange can then be applied to reduce the size of the working set of parallel threads and improve the effectiveness of thread affinity.

To illustrate the importance for a characterization of thread behaviors, consider the measurements on SPEC OpenMP benchmarks shown in the data from Figure 5.1 and Table 5.1 is collected by our analysis and tool. The graph 5.1 displays for several benchmarks (and functions of these benchmarks) the percentage of memory accesses on shared data compared to the total number of memory accesses. The min/max percentage on 8 threads is shown. For `equake`, in function `smvp`, line `1310`, between 10 and 12.5% of the memory accesses are on data shared with another thread. It shows in this case the importance for this function to take advantage of any thread affinity and to choose carefully the mapping of the threads on the architecture. Yet, to better characterize the data shared among the threads and to decide whether or not there is potential performance issue here, it is necessary to know if this traffic generates cache coherency traffic (this would be the case if at least one of the accesses is a write) or if there is any load imbalance issue.

BENCHMARK	**(Function,line)**	**KB/THREAD**
swim	(calc1,278)	12209
galgel	(systn,98)	1032
fma3D	(solve,3766)	1997
equake	(smvp,1310)	265
mgrid	(resid,205)	4790
apsi	(dudtz,1773)	4094
art	(computevaluesmatch,934)	703

Table 5.1: Table showing for each benchmark (same as before) the mean size of the working set of each thread.

For `mgrid` benchmark, there is no memory access on data shared with another

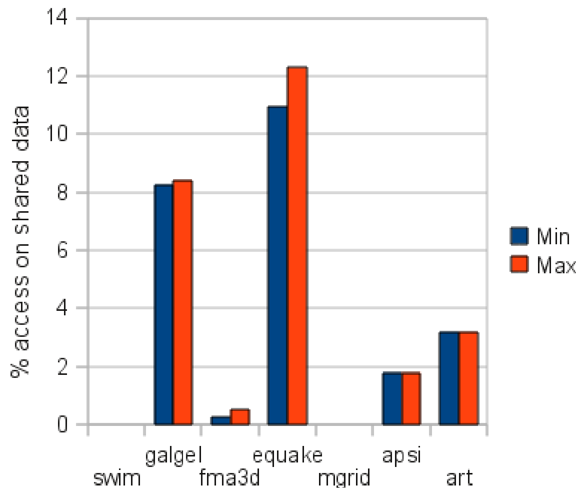


Figure 5.1: Percentage of memory accesses performed on shared blocks of data in hotspot loops of SPEC OpenMP benchmarks. Min and max correspond to the minimum and maximum ratios considering all the threads. The architecture used is a 2 socket quad-core 2.26Ghz Nehalem, with 8MB L3 and 256KB L2.

thread. This could show that there is no performance issue or thread interaction in this case. Table 5.1 shows that each of the 8 threads of the function access 4790 KB of data, by far exceeding the capacity of the $L2$ (one per core) and of the $L3$ (one per 4 cores). The threads therefore interact by sharing the same memory bandwidth and prefetch mechanism. For this case, checking if there is any intra-thread reuse, if the address stream can be prefetched are then necessary steps in the performance tuning process. This shows that detecting and characterizing the main performance issues in thread interactions are essential for tuning multi-threaded application, in order to adapt compiler and runtime strategies accordingly.

To address such multi-threaded-application performance tuning issues, previous works and tools have focused on the analysis of hardware counters [22, 4] or cache simulators (see [51, 70, 71, 113, 82, 83] for instance). While giving a precise description of the hardware events occurring during an execution, the hardware counters can generate tremendous amount of data that is difficult to relate directly to compiler optimizations or runtime parameters. We consider hardware counters as a useful complementary source of information to confirm the correctness of our data sharing metrics (e.g : miss rate to confirm false sharing). Concerning cache simulator approaches, they mostly focus on prediction of cache misses or prefetched streams. Some of them [71, 113] characterize memory access patterns of an application but they only consider mono-thread applications or do not relate these patterns to OpenMP scheduling strategy parameters.

We present in this chapter a new method and analyses to characterize thread memory behavior, in order to provide the necessary information for multi-threaded performance tuning. The objectives are to identify memory performance issues of multi-threaded applications and to apply source code transformations or change OpenMP scheduling strategies according to the observed memory access patterns. Our approach is based on memory traces and their analysis according to a simple cache model. We show that by combining a static induction variable analysis on binary code with an efficient trace compaction technique, the cost of collecting

memory access traces per thread and per instruction remains low for some benchmarks. Implementing this analysis in our performance tuning tool MAQAO [69], we describe the results obtained on multi-threaded OpenMP benchmarks and identify key performance factors, like detecting inefficient patterns and generating reshaping hints, from their memory behavior.

The chapter is organized as follows. First, we present the related work concerning memory behavior characterization. Then, section 5.3 describes the Nested Loop Recognition (NLR) method for trace compression [59] and we present its extension for multi-threaded codes and its composition with a static induction variable technique. After that, section 5.5 and 5.6 shows the metrics and related analyses we use to define the thread memory behavior. Section 5.7 presents the analysis of collected data and the generation of reshaping hints, before finally concluding in section 5.8.

5.2 Related work

Several studies have used trace-driven methodologies to characterize the memory behavior of multi-threaded applications. Jaleel et al.[51] propose an on-the-fly simulator based on PIN[118], modeling a three level cache hierarchy with LRU replacement policy. While the cache model is more elaborate than the one proposed in this paper, the statistics collected by their simulator are application-wide and not detailed for each loop. They mostly focus on miss rates and characterization of multi-threaded workload. There is no characterization of reuse across threads. Marathe et al. [70] characterizes with a simulator called ccSIM the coherence traffic for OpenMP programs. This simulator capture trace of memory accesses, function calls and OpenMP synchronizations and simulates LRU replacement policy. It analyzes parallel loops and our approach is very similar to this one. The main distinction correspond to the focus of their tool, on coherence traffic, whereas the metrics shown in this paper focus on the characterization of memory access patterns. Besides, there is no analysis of the memory access strides.

Lee et al. [66] propose a method to dynamically adjust the number of thread in an application. It relies on offline profiling information, collecting memory accesses per thread. It analyzes potential communication cost, that is load-store, and store-store inter-thread traffic. According to some cost model, threads are coalesced into sequential code dynamically if needed. This could be considered as an alternative to OpenMP scheduling strategies.

Finally, Kandemir et al. [58] describe a compiler based, cache topology aware code optimization scheme for multi-cores. The method described maps iterations of parallel loops according to the data accessed and the topology of the memory hierarchy. There is no analysis of an existing multi-threaded code, but the technique proposed offers a new approach for the distribution of parallel iteration that extends the scope of what is proposed in OpenMP parallel loops. Profile information as presented in this paper would complement such static approach, overcoming the difficulties to analyze irregular codes.

5.3 Compact multi-threaded trace collection

Binary code has the advantage, compared to source code, of being the real code executed on the parallel machine, once compiler optimizations have been performed.

A locality analysis based on binary code takes into account transformations such as padding, vectorization or loop transformations (such as interchange or tiling for temporal locality). Besides, parallel memory traces capture runtime decisions concerning the number of threads or the scheduling strategy. We assume in the following that the instrumentation focuses only on the most important parts (hottest parts) of the application and we consider only OpenMP applications.

5.3.1 Trace compaction techniques

The first approach has been to try to leverage spatial locality in program execution. The basic principle is to take advantage of the fact that two successive memory address accesses are likely to access nearby memory locations. The idea is thus to encode the stride instead of the full memory address. Compression is achieved because strides take fewer bits to encode that full addresses. MACHE [101] is prototypical of this approach: it compresses memory access traces by first distinguishing between reads, writes and fetches (encoding separately three sub-traces), and second by storing strides instead of addresses. When the stride is too large for the allocated bit size, a special value is output, followed by the full address, which also serves as a new base for future addresses coding. The PDATS and PDATS II algorithms [54, 53] further improve on the original strategy by incorporating several optimizations, like various default strides, and by using run-length encoding of stride values (which is a way to represent elementary loops) to overcome the inherent limitations on the compression rate.

The second approach is completely different and, from our point of view, almost exactly symmetric to the first approach. It consists in extracting higher level structure from the program traces. The prototypical system here is WPP (Whole Program Paths) [62], which is based on the SEQUITUR sequence analysis algorithm. SEQUITUR [84] is an incremental algorithm, of linear complexity [10], that builds a context-free grammar from a sequence of incoming symbols. The grammar is built with the help of two rules: no couple of consecutive symbols can appear twice, and no symbol can be used only once. These rules ensure that the grammar stays small, and because of its hierarchical nature, it is expected that the grammar will have a size proportional to the logarithm of the original trace size. This method has been successfully applied to compression of control flow information, but seems difficult to apply to numerical data.

The most recent approach uses value predictors, and is based on the observation of all previous values. The VPC algorithms [16, 15], in particular VPC4, use this strategy and achieve the best known compression rates currently. The basic strategy consists in maintaining a set of value predictors that are updated with incoming values. VPC uses two main kinds of predictors: ssimple value predictors which predict the most likely value among the last values seen and finite context method predictors which proceed in the same way except that they maintain several contexts and select the most appropriate depending on recent history. VPC also includes differential versions of these predictors (i.e., using strides instead of values). Whenever a new value is read, every predictor is exercised, and the index of the one that predicted the correct value is output. Unpredictable values are output to a separate stream. Because the number of predictors is small, the index of the correct predictor requires fewer bits than a full value (e.g., address), and so the trace is compressed. Moreover, VPC includes several heuristics to choose one predictor when several of

them predict correctly, with the explicit goal that the predictor index sequence itself exhibits regularity. Each of the output streams is finally piped into a second stage, a general purpose compressor that performs further compres- sion, usually using bzip2. Another predictor-based compressor was proposed by Barr and Asanovic but specialized for control-flow traces. Other criteria could be used to categorize trace compression techniques. One important factor is the trace format, and the kind of information it includes. Put succintly, MACHE and VPC algorithms are especially well suited for traces containing addresses or other numerical quantities, and SEQUITUR works better on control flow information. Also, in most cases, combined traces (containing several attributes per entry) are split into several streams that are compressed separately. In some cases, a trace is broken down into small subtraces: SBC (Stream Based Compression) [74] compresses instruction streams" (i.e., contiguous trace entries with no intervening branching) with a technique similar to PDATS. The SIGMA system [26] uses basic blocks with various attributes as trace elements. A recent development on VPC, called SCT (Seekable Compressed Traces) [78], improves compression by providing specialized predictors (e.g., a branch predictor dedicated to branches in the trace). Both VPC and SCT require a description of the trace format (number of fields, and type of fields for SCT). SCT is interesting also in that it adds “reset markers” in the compressed traces, so as to allow extracting some part of the trace without first decompressing everything that precedes the targeted extract.

5.3.2 Our choice : Nested Loop Recognition

The technique proposed by Ketterlin and Clauss represents memory address streams as a union of Z-polytopes which are represented by (nested) loops. The idea of using loops to characterize an accessed region was first introduced by Elnozahy [35]. Simpler representations have been proposed using triplets (starting addresses, stride, number of references) and their extension to multidimensional triplets [71]. This is a natural approach since most of the execution time of an HPC application is spent in loops, and many memory accesses are regular, i.e. depending linearly of loop counters.

The NLR method captures in a few polyhedra the stream of addresses obtained when iterating the elements of a multidimensional array. Moreover, the order of enumeration is represented through the loops representing the Z-polytope (the sequential schedule is captured). For irregular data structures (indirections for instance) that a multidimensional Z-polytope, as for instance when iterating over all elements of a multidimensional array, the quality of the compaction degrades and the trace consists in a possible large union of Z-polytopes and of singletons. Each memory stream is assigned an internal stack that stores either regular and irregular patterns. Regular patterns are stored in the loop format described above. Irregular patterns, which correspond to a sequence of numbers without any affinity, are kept as is. The stack size management is controlled by three factors:

- the maximum stack size (length)
- the maximum number of terms within the loop body representation (breadth)
- the number of elements to remove when the size limit is reached

The algorithm is lossless as long as the stack is large enough to store all memory streams. On some huge programs it may be necessary to voluntarily limit the stack

in order to prevent consuming all the available memory. In this case the algorithm becomes lossy.

In terms of space complexity, consider a Z-polytope of d dimensions and containing n references. Assuming the Z-polytope is a cube of size $n^{1/d}$ in each dimension, a loop iterating one of these dimensions requires $O(1/d \log n)$ characters. The NLR trace has a size of $O(\log n)$, representing the Z-polytope with at most d loops.

Figure 5.2 shows an example of NLR representation of the memory accesses for a store instruction of a matrix multiplication.

Source Code

```
for (j=0;j<NRA;j++)
  for(i=0;i<NCB;i++)
    for (k=0; k<NCA; k++)
      c[i*NRA+j] += ...
```

NLR Representation for c[i*NRA+j] accesses

```
for i0 = 0 to 7
  for i1 = 0 to 63
    for i2 = 0 to 63
      0x62cd5a0 + 8*i0 + 512*i1
```

Figure 5.2: DGEMM 64x64 Source loop and the NLR representation of its store statement

5.3.3 Memory Trace

Although we could directly present the most advanced method we are using in our Memory Trace Library (MTL) tool in order to trace memory streams, we find value in detailing the stages we have been through before getting to our final solution. We will present the former naive and new enhanced methods. Both rely on MADRAS [117], the binary manipulation layer of the MAQAO framework, in order to patch a given binary file.

5.3.3.1 Naive method

A call to the MTL *store* function is added for every load and store instructions belonging to the identified hot loops. Each call passes the memory reference of the current instruction (from a given thread) being executed and uses a modified version of the NLR[59] compaction algorithm to store the accessed addresses. Each thread has its own data structures. This means that the same instruction may be involved in different traces depending on which thread is executing it. For instance, traces recorded from OpenMP applications for different threads may contain different instructions if the code involves control structures.

5.3.3.2 Enhanced method

The previous method reaches its limits when dealing with benchmarks using reference (real) input sets. That is why we started working on an enhanced method. There is actually no magic bullet to make memory tracing faster without resorting to sampling. The only possible method is reducing the number of instrumented instructions and when considering loops, insert instrumentation as high as possible in the loop hierarchy. The best scenario being the case where we can resolve statically all needed values in order to generate the memory stream values of each instruction afterwards. We also need to be sure that all threads are going to execute the same

instructions. That is not the case when control structures are present. For instance, traces recorded from OpenMP applications for different threads may contain different instructions since the execution path may differ. Since we are only interested by memory addresses, we achieve a lower instrumentation overhead by only instrumenting registers and stack values involved in address computation. Addresses are often computed as the sum of a base register or stack address (a loop invariant) and an offset. This offset is an induction variable that can easily be detected by a static analysis, when applicable. In order to find out which registers and stack values must be instrumented we use a strength reduction algorithm, which main steps are as follows:

- Find loop invariants (registers and stack values);
- Find induction variables (affine expressions only, since the trace has only Z-polytopes);
- Find all memory accesses based on induction variables and loop invariants;
- Instrument all loop invariants and all memory accesses that are not built from induction variables and loop invariants;

Doing so reduces the number of instrumented instructions and moves instrumentation outside of the hot loop when all addresses are induction variable based. In this case, the total overhead if significantly reduced. Moreover, the space requirements are lowered: for a d dimensional space containing n references, if all references are built from a base register and a induction variable and if M memory accesses depend on one base register, the NLR trace has a size of $\frac{d-1}{dM} \log n$ characters.

Practically, the induction variable detection analysis requires to find the functions, blocks and loops of the binary file. This is achieved thanks to the code abstraction layer of MAQAO [69].

Instrumentation timings using both methods are confronted in the next subsection.

5.3.4 Instrumentation time

Our analyses are based on traces produced by instrumenting a given binary (parallel) program. This means that our approach is applicable only if the overhead of instrumentation is reasonable. Figure 5.3 and 5.4 respectively shows, for the naive and enchanced instrumentation methods, the overhead of instrumentation observed on some benchmarks that have different types of regularity. On the one hand, we clearly observe the naive method limitations. On the other hand, most of the time, the overhead remains acceptable with the enhanced method.

The results presented here were run on a 2 socket quad-core 2.26Ghz Nehalem E5520, 8MB L3 and 256KB L2. The Hyperthreading feature was disabled.

Benchmark	Original time	Naive instrumentation	Overhead
314.mgrid_m ref	4m03s	16h14m55s	237x
312.swim_m ref	2m56s	3h57m50s	80x
NAS PB ft.B	11s	5h55m10s	1936x

Figure 5.3: Instrumentation overhead using the naive method

Benchmark	Original time	Enhanced instrumentation	Overhead
314.mgrid_m ref	4m03s	37m56s	8.36x
312.swim_m ref	2m56s	3m03s	0.04x
NAS PB cg.B	17s	35m29s	58.88x
NAS PB ft.B	11s	1h04m10s	349x

Figure 5.4: Instrumentation overhead using the enhanced method.

For `swim`, memory accesses are very regular and the code takes full advantage of the static induction variable analysis to reduce the execution overhead. For the other codes, time and space consumed by instrumentation depends upon code regularity.

5.3.5 Reconstructing address streams

When using the enhanced method, the collected traces contain Z-polytopes that represent values of either registers, stack values or memory accesses. The latter corresponds to the plain memory stream and needs no reconstruction (only scenario with the naive method). Thus, there is nothing to do. But for the others, the memory streams must be reconstructed based on the captured dynamic values and the affine expression found statically for each instruction. In a nutshell, the aim is to reconstruct the Z-polytopes describing the memory stream values of each instruction, the ones that would have been traced if only the naive method were used.

To illustrate the details below, we will be using the content of a memory trace obtained from the tracing of SPEC OMP 2001 312.*swim* benchmark. In particular, focus will be directed towards one instruction of the most time consuming loop. Figure 5.5 exhibits the assembly code of that instruction.

```
andps   -30416(%R14,%RBX,1),%XMM4
```

Figure 5.5: Assembly code of a memory instruction on x86_64 architectures. In general : MNEMONIC OFFSET(BASE,INDEX,SCALE),REGISTER

In order to achieve the reconstruction of the address stream of an instruction, we perform the following steps:

- Load the affine expression, found at the instrumentation stage along with constants. The induction variable (we will name it i2) found for this loop is the R13 register and its step is 8. In our example the affine expression is :

  ```
  for i2 = 0 to (3128 / 8) - 1
    -30416 +  R14 + RBX + 8 * 8*i2.
  ```

  The bound is divided by 8 because in NLR, the step of loops is by default 1 (8 in the real loop). We also substract 1 because the high loop bound of NLR representation is inclusive.

- Extract and categorize Z-Polytopes from trace : find each loops' starting and bound values, induction variables and invariants (registers). Figure 5.6 shows the values extracted from the memory trace.

- Merge constants and Z-Polytopes that are not constants. By merging the Z-Polytope representing R14 values and the previous expression we obtain :

```
for i0 = 0 to 799
  for i1 = 0 to 33
    for i2 = 0 to 165
      -30416 + 374589856 + 30416 + 30416*i1 + 64*i2.
```

- Merge constant values of the affine expression to obtain the starting address of the memory stream.

- Resulting Z-Polytope is :

```
for i0 = 0 to 799
  for i1 = 0 to 33
    for i2 = 0 to 165
       val 374589856 + 30416*i1 + 64*i2
```

```
Loop starting value :

  R13 Register Z-Polytope :
  Constant : 0

Loop bound value :

  R8 Register Z-Polytope :
  Constant : 3128

Invariants :

  R14 :
  Constant : 374589856

  RBX Register Z-Polytope :
  for i0 = 0 to 799
    for i1 = 0 to 166
      30416 + 30416*i1
```

Figure 5.6: 312.swim SPEC OpenMP 2001 benchmark : types and values of Z-polytopes of the most time consuming loop extracted from the memory trace

Figure 5.7 concludes this section by giving an overview of our multi-threaded trace collection layer.

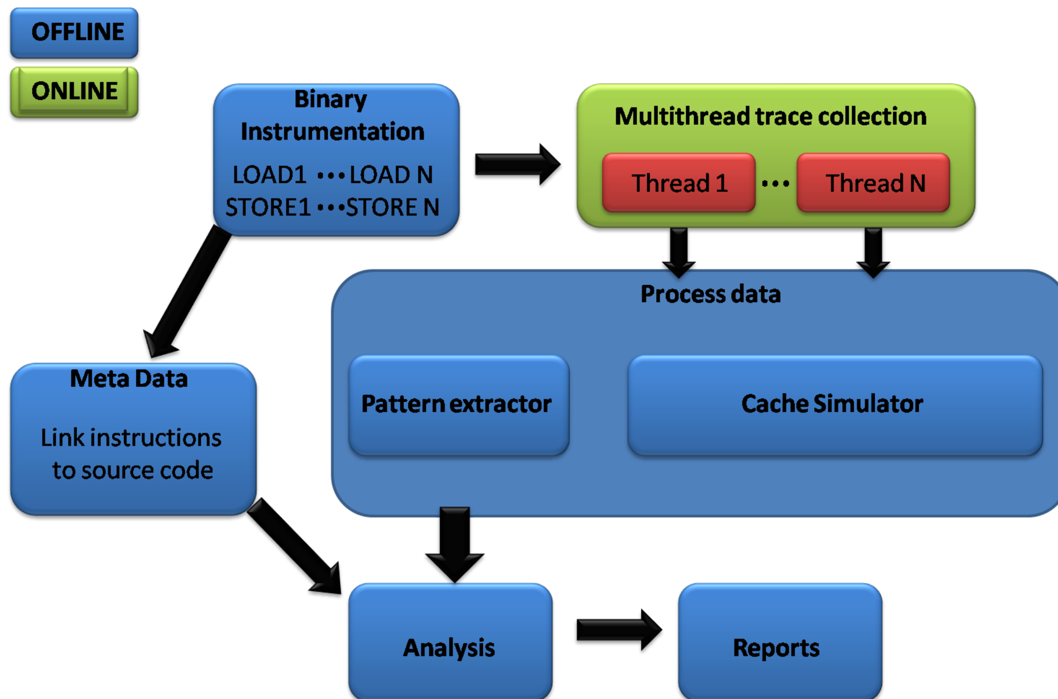


Figure 5.7: Overview

5.4 Simplified cache simulator

The trace is replayed thereby reproducing the virtual address flows of the application. Each address populates the same simplified cache structure (address components are described in figure 5.8). By simplified we mean it has no coherency protocol nor data replacement policy, thus having a full associative cache (all accesses are hits). Each cache line contains a collection of the instructions (and thread belonging) that have accessed the cache line along with the access count (hits) Figure 5.9. This level of accuracy provides us with the ability to link each instruction to the source code.

Note that with very little adjustments, we could consider pages instead of cache lines. Thus being able to detect issues related to memory paging (DTLB cache). For instance If we consider 4KB pages, it is easy to track down access patterns that use strides greater than 4KB and report which part of the code leads to this issue.

Tag	Set	Offset

Figure 5.8: Components of an address.

5.5 Single thread memory behaviour analysis

It is difficult for a developer to establish a relationship between the way he built his code and the memory behavior of his application, which can lead to severe

```
Tag 89675538 (7 threads)
        Thread 1 (2 instructions)
                store INSN 2(300 hits)
                store INSN 4(256 hits)
        Thread 2 (1 instructions)
                store INSN 4(224 hits)
        Thread 3 (2 instructions)
                store INSN 2(288 hits)
                store INSN 4(52 hits)
        Thread 4 (2 instructions)
                store INSN 2(304 hits)
                store INSN 4(208 hits)
        Thread 5 (2 instructions)
                store INSN 2(360 hits)
                store INSN 4(156 hits)
        Thread 6 (2 instructions)
                store INSN 2(304 hits)
                store INSN 4(192 hits)
        Thread 7 (2 instructions)
                store INSN 2(384 hits)
                store INSN 4(224 hits)
```

Figure 5.9: Content of a cache line of the cache simulator.

performance loss. It is more natural to think in an algorithmic way rather than in terms of a cache structure. Even with a cache aware algorithm, when the program relies on input sets, the behavior can be unpredictable. The reports' aim is to help the user better understand the memory behavior issues of his application. Note that it is easy to relate the access pattern (affine expressions) of an instruction to the source code because we are able to link each instruction to a source code line thanks to debug information (dwarf). In our approach, access patterns are collected for each thread, and for each of its instructions.

In this section we will focus on the study of the analyses that can be performed based on access patterns. These are extracted from the Z-polytopes of the trace and corresponds to affine expressions defining them, just omitting the bound values. Two pieces of information are then very useful, namely, strides and initial values (constant part). Actually, we do not get strides but bytes. But since we know which type of data we are manipulating (single or double precision for instance), thanks to the concerned assembly mnemonic, we actually easily deduct strides. In the rest of this section we will continue to use the term stride for the sake of simplicity, even when dealing with Bytes displacements.

5.5.1 Analyzing Strides

Because strides reveal the way the cache is accessed, it is important to track them down. They are caught at instruction level since load and store instructions defines the memory behavior of an application. The affine expressions generated by the NLR algorithm provides an easy way to retrieve stride information. Figure 5.2 is an example of the NLR representation of a traced source loop. Strides are expressed in bytes and patterns must be read from right to left (innermost access to outermost). i0 and i1 provides a depth (nesting) indication. Thus $8 * i0 + 512 * i1$ means that consecutive stores are done every 512 bytes (64x64 times) and repeated every 8 bytes (8 times). We easily recognize the 64x64 DGEMM column traversal. Actually each loop instruction can have several multiple stride access patterns (Figure 5.11). The strides may be obvious sometimes when dealing with very regular code, which is not the case of irregular codes even if behaving like regular codes (regular accesses even though indirect access).

A common case leading to bad locality is nested loop traversal order(Figure 5.10). It is a common mistake for programmers migrating from C to FORTRAN or inversely. The accesses are done on i, j indecies (`C[i*NRA+j]` which corresponds to `C[i][j]`). On the left part, traversal is performed in the correct order (contiguous data) whereas in the right side traversal, indexes are inverted (`C[j][i]`). Hence the wrong access pattern (bad locality because not reading contiguous data). From an analysis point of view, the access pattern `8*i0 + 512*i1` shows that the innermost accesses (`512*i1`) have a higher stride that the outermost ones (512 $>$8). As a consequence, the accesses should be switched if possible. So from right to left, a loop interchange solves the problem.

```
64x64 DGEMM
#pragma omp for schedule (static)
for (i=0;i<NRA;i++)
  for(j=0;j<NCB;j++)
    for (k=0; k<NCA; k++)
      c[i*NRA+j] += a[i*NRA+k] * ...;
         8 * i0
```

```
64x64 DGEMM
#pragma omp for schedule (static)
for (j=0;j<NRA;j++)
  for(i=0;i<NCB;i++)
    for (k=0; k<NCA; k++)
      c[i*NRA+j] += a[i*NRA+k] * ...;
    8 * i0 + 512 * i1
```

Figure 5.10: Wrong access pattern

Hence, strides can also be very helpful in figuring inefficient memory data layouts. Data structures layout has a considerable impact on performances. It will be discussed in section 5.7.

5.5.2 Address based analysis

A lot of issues that will cause high cycles per interaction penalties are related to architectural concerns, in particular, memory management and cache accesses. The most usual issues observed are, namely, memory alignment, address aliasing, bank conflicts and set associativity issues.

```
for i0 = 0 to 7
  for i1 = 0 to 39
    for i2 = 0 to 15
      for i3 = 0 to 127
        for i4 = 0 to 127
           val 0x2ba1a3db4428 + 135200*i2 + 1040*i3 + 8*i4
    for i2 = 0 to 7
      for i3 = 0 to 7
         val 0x2ba1a5100b68 + 80*i2 + 8*i3
    for i2 = 0 to 15
      for i3 = 0 to 15
         val 0x2ba1a50f5568 + 144*i2 + 8*i3
    for i2 = 0 to 15
      for i3 = 0 to 15
         val 0x2ba1a50f5f88 + 144*i2 + 8*i3
    for i2 = 0 to 3
      for i3 = 0 to 31
        for i4 = 0 to 31
           val 0x2ba1a50a89a8 + 9248*i2 + 272*i3 + 8*i4
```

Figure 5.11: Multiple strided access for one instruction

Unaligned accesses All memory accesses should be aligned in order to get the best performance. Using unaligned accesses has a cost depending upon the processor (more precisely the micro-architecture). Even if this penalty have been lowered on the latest x86-64 architecture, there is still a price to pay for using unaligned accesses.

Inefficient aligned accesses Even when memory accesses are aligned, a penalty may be observed. This is due to architectural issues. For instance, on the Intel Sandy Bridge processors (micro-architectures), when more than two store operations are involved, even if using aligned memory accesses (on vectors) are used, we can notice up to a 10 cycles penalty. Figure 5.12 shows such a scenario. We can see that when using a 32Bytes alignment (from a 4KB page boundary) for the third stream, a 10 cycle penalty is present no matter the alignment used for the fourth stream. Based on micro-benchmarking test suites to detect such issues, we lookup for such cases by analysing the strides of the access patterns. We then propose the best measured alignment. In this case, all but 32-Bytes offset.

4K aliasing When a load instruction follows a store instruction using the same data which is being committed to memory, x86-64 architectures, uses a mechanism called store forwarding to directly feed the load instruction. Actually there are conditions that must be met so that store forwarding occurs (such details can be found in the Intel 64 Optimization Reference Manual [48]). In order to hide the load latency, the memory disambiguator always assumes a dependency between loads and earlier stores that have the same address bits 0 to 11. 2^{12} - 1 covers the 0 to 4095 interval which is 4K. 4K false store forwarding, better know as 4K

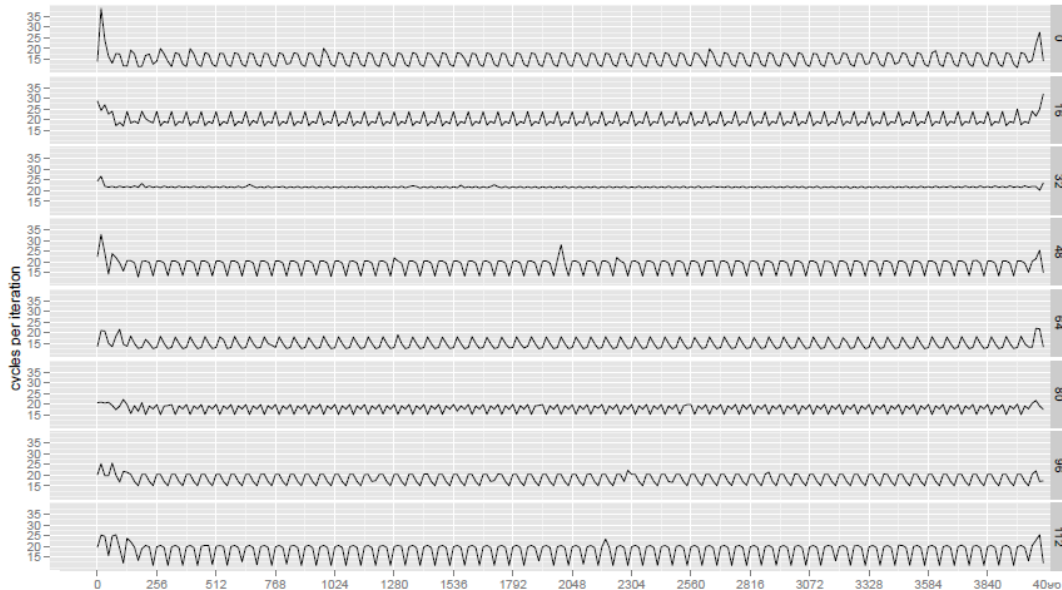


Figure 5.12: Memory stream performance of a Store/Store/Store/Load pattern on A SandyBridge machine (Xeon E5). Left Y-Axis carries cycles per iteration. Right Y-Axis accounts for alignment of the third stream and finally X-Axis is the alignment of the fourth stream. First and second stream are fixed to 0.

aliasing, may happen when the processor performs store forwarding. If the data of the store and load instructions are separated by a 4-KByte offset. If we consider for instance addresses 0x600010 and 0x602010, they have the same value for bits 5 - 11 of their addresses. The accessed byte offsets can have partial or complete overlap. Until the Intel micro-architecture code name Nehalem, penalty was about 20 cycles. With the introduction of split registers, post-Nehalem micro-architectures are able to handle loads and stores that span two cache lines in a faster manner, as long as split registers are available (store buffer not full). The penalty is then lowered to 5 cycles. This penalty may be significant when 4K aliasing occurs repeatedly and the load instructions are on the critical path.

The first method to get rid of 4K aliasing issue is to change offsets between input and output buffers, if possible. Data should be aligned on 32 Bytes boundaries. If it is not possible, 16-Byte memory accesses should be preferred.

Set Associativity issues Set Associative Caches are organized in two ways. If two data elements are in memory at addresses that are 2^N apart such that they fall in the same set, then they will take two entries in that set on a 4-way set associative cache. The fifth access that tries to occupy the same slot will force one of the other entries out of the cache, turning a large cache into a N entry cache. Level one, two, and three caches of different processors have different cache sizes, line sizes, and associativity. Skipping through memory at exactly 2^N boundaries (2K,4K,16K,...,256K) will cause the cache to evict entries more quickly. The exact size depends on the processor and the cache. Once again detecting such patterns with our access patterns helps solving this potential issue.

Memory bank conflicts For best performance, computers are installed with multiple physical memory modules for the main memory. No single physical memory module can keep up with requests from the CPU. The system determines which data goes to which physical memory module via some bits in the address. Skipping through memory, large 2^N may cause all or most of the accesses to hit the same physical memory module. Since 16-byte loads can cover up to three banks, and two loads can happen every cycle, it is possible that six of the eight banks may be accessed per cycle, for loads. A bank conflict happens when two load accesses need the same bank (their address has the same 2 to 4 bit value) in different sets, at the same time. When a bank conflict occurs, one of the load accesses is recycled internally. In many cases two loads access exactly the same bank in the same cache line, as may happen when popping operands off the stack, or any sequential accesses. In these cases, conflict does not occur and the loads are serviced simultaneously. Detecting such patterns with our access pattern analyses helps solving this potential issue.

5.5.3 Experiments

Experimental setup Experiments were run on a 2 socket quad-core 2.26Ghz Nehalem E5520, 8MB L3 and 256KB L2. Hyperthreading feature is disabled.

5.5.3.1 APSI ; inefficient access patterns

For this benchmark, two regions (arrays) exhibit an inefficient access pattern (Figure 5.13). Such a huge (200704 Bytes distance) stride will prevent hardware prefetch from retrieving data. Moreover it will produce data TLB misses, thus worsening data locality. Based on array splitting (described in the previous section), we first test if it is worth applying this optimization. If all references to the concerned regions have the same traversal properties, then we can propose a loop interchange (reversing the order). If it is not applicable we stick with splitting.

5.5.3.2 NAS Parallel Benchmarks 2.3 C

The analysis of memory addresses used by the store instruction at line 436 of Figure 5.16 produces the access pattern in Figure 5.15.

Our analysis reveals an inappropriate access pattern leading to poor spatial locality. Indeed, the $262144 * i2$ access pattern reveals that prefetching cannot be used and Data TLB misses will be triggered. Taking a look at source code, we can see that parallelization is done on the outermost loop, whose induction variable is i. However i is used to index the innermost dimension of the indexmap array (at line 436). According to this information, we transform the code in Figure 5.16.a by changing the traversal order. The result is shown in Figure 5.16.b. We also provide the trace of the same store instruction before and after transformation in Figure 5.17.We can observe that the indexmap array is now accessed linearly. Executing the new application, we get a 5% improvement over previous version (19 seconds V.S. 20 seconds).

5.5.3.3 ITRSOL : a solver from Dassault Systems

ITRSOL is a solver that is used in a tool developed by Dassault Systems. Thanks to prior timing analyses, it appears to be one of the most time consuming hotspot.

```
Trace of thread 1
Instruction 1
for i0 = 0 to 69
  for i1 = 0 to 1567
    for i2 = 0 to 54
       val 47089115907088 + 8*i1 + 200704*i2
Instruction 2
for i0 = 0 to 69
  for i1 = 0 to 1567
    for i2 = 0 to 54
       val 47089104667664 + 8*i1 + 200704*i2
Instruction 3
for i0 = 0 to 109759
  for i1 = 0 to 54
     val 47089127059848 + 16*i1
Instruction 4
for i0 = 0 to 109759
  for i1 = 0 to 54
     val 140737138460920 + 16*i1
```

Figure 5.13: SPEC OpenMP 2001 benchmark : trace of the first most time consuming loop

From our analysis, we get the plot in figure 5.19. It reveals a false sharing issue spread among all the threads. A closer look at the source code (figure 5.18) reveals that the traversal order is not efficient. This is confirmed by the our pattern report which detects the following pattern : $4000 * i1 + 8 * i2 + 792584 * i3 + 3170336 * i4$. We clearly see that the innermost strides, $792584 * i3 + 3170336 * i4$, are huge.

5.5.3.4 RECOM-AIOLOS software: a real-world application

The RECOM-AIOLOS application [99] is a 3D-Combustion Simulation software for the modeling of industrial furnaces and boilers. It illustrates how data locality can impact performance of an application. Consider the FORTRAN source code in Figure 5.20, corresponding to one of the hot loops of this code.

Each loop uses distinct elements from the same data structure. Actually it is a checkerboard pattern (Red/Black). Figure 5.21 shows an example of the natural data structure layout from the developer point of view that results in a performance degradation. The access pattern found shows that only one element out of 2 is used. The obvious solution is to split the main array into two smaller arrays (one for each loop). This reshaping solution brings up a 30% performance gain for the loop.

5.6 Understanding interactions between threads

In the previous section, we detailed how access patterns could help in the characterization of the memory behavior of one thread. In this section, we will go one step further and study how it is possible to understand the interactions between multiple threads. Hence, being able to characterize the memory behavior of multi-threaded applications.

```
2032 !$OMP DO
2033       DO 30 J=1,NY
2034          DO 40 I=1,NX
2035             HELP1(1)=0.0D0
2036             HELP1(NZ)=0.0D0
2037             DO 10 K=2,NZTOP
2038                IF(NY.EQ.1) THEN
2039                   DV=0.0D0
2040                            ELSE
2041                   DV=DVDY(I,J,K)
2042                ENDIF
2043                HELP1(K)=FILZ(K)*(DUDX(I,J,K)+DV)
2044 10          CONTINUE
2045C
2046C       SOLVE IMPLICITLY FOR THE W FOR EACH VERTICAL LAYER
2047C
2048             CALL DWDZ(NZ,ZET,HVAR,HELP1,HELPA1,AN1,BN1,CN1,ITY)
2049             DO 20 K=2,NZTOP
2050                TOPOW=UX(I,J,K)*EX(I,J)+VY(I,J,K)*EY(I,J)
2051                WZ(I,J,K)=HELP1(K)+TOPOW
2052                WWIND1(MY_CPU_ID)=WWIND1(MY_CPU_ID)+WZ(I,J,K)
2053                WSQ1(MY_CPU_ID)=WSQ1(MY_CPU_ID)+WZ(I,J,K)**2
2054 20          CONTINUE
2055 40       CONTINUE
2056 30    CONTINUE
2057!$OMP END DO
```

Figure 5.14: SPEC OpenMP 2001 benchmark : 324.apsi_m WCONT in apsi.f

```
forall t = 0 to 7
 for i0 = 0 to 31
  for i1 = 0 to 255
   for i2 = 0 to 127
    val 6470624 + 128*t + 4*i0 + 1024*i1 + 262144*i2
```

(a) Before

```
forall t = 0 to 7
 for i0 = 0 to 14859264
   val 6470624 + 8388608*t + 4*i0
```

(b) After

Figure 5.15: Merged trace of threads for the store instruction, before and after transformation

In order to achieve this goal, we need to be able to characterize the traffic and the shared resources between threads. Traffic can be caused either by the coherence protocol or remote data accesses (NUMA). Bad access patterns, due to data traversal and layout, determines the quality of the locality and must be monitored. Note that since we are not considering time in our approach, we will focus on spatial locality.

We defined three metrics, namely, data sharing, workload balancing and affinity, that allow the inspection of the previously mentioned issues. Traces return a large

```
427  #pragma omp for
428  for (i = 0; i < dims[2][0]; i++) {
429    ii =  (i+1+xstart[2]-2+NX/2)%NX - NX/2;
430    ii2 = ii*ii;
431    for (j = 0; j < dims[2][1]; j++) {
432      jj = (j+1+ystart[2]-2+NY/2)%NY - NY/2;
433      ij2 = jj*jj+ii2;
434      for (k = 0; k < dims[2][2]; k++) {
435        kk = (k+1+zstart[2]-2+NZ/2)%NZ - NZ/2;
436        indexmap[k][j][i] = kk*kk+ij2;
437      }
438    }
439  }
```

(a) FT.B compute_index in ft.c

```
#pragma omp for
  for (k = 0; k < dims[2][2]; k++) {
    kk = (k+1+zstart[2]-2+NZ/2)%NZ - NZ/2;
    kk2 = kk*kk;
    for (j = 0; j < dims[2][1]; j++) {
      jj = (j+1+ystart[2]-2+NY/2)%NY - NY/2;
      kj2 = jj*jj+kk2;
      for (i = 0; i < dims[2][0]; i++) {
        ii =  (i+1+xstart[2]-2+NX/2)%NX - NX/2;
        indexmap[k][j][i] = ii*ii+kj2;
      }
    }
  }
```

(b) FT.B after transformation

Figure 5.16: NAS Parallel benchmarks FT

```
Trace of thread 1
for i0 = 0 to 1048575
   val 6470656 + 4*i0
Trace of thread 2
for i0 = 0 to 1048575
   val 10664960 + 4*i0
Trace of thread 3
for i0 = 0 to 1048575
   val 14859264 + 4*i0
Trace of thread 4
for i0 = 0 to 1048575
   val 19053568 + 4*i0
Trace of thread 5
for i0 = 0 to 1048575
   val 23247872 + 4*i0
Trace of thread 6
for i0 = 0 to 1048575
   val 27442176 + 4*i0
Trace of thread 7
for i0 = 0 to 1048575
   val 31636480 + 4*i0
Trace of thread 8
for i0 = 0 to 1048575
   val 35830784 + 4*i0
```

Figure 5.17: Trace of each thread for the store instruction after transformation

amount of data to processes and only relevant information is kept.

```
!$OMP  PARALLEL DO DEFAULT(NONE)
!$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
```

Figure 5.18: Source loop of the Dassault code

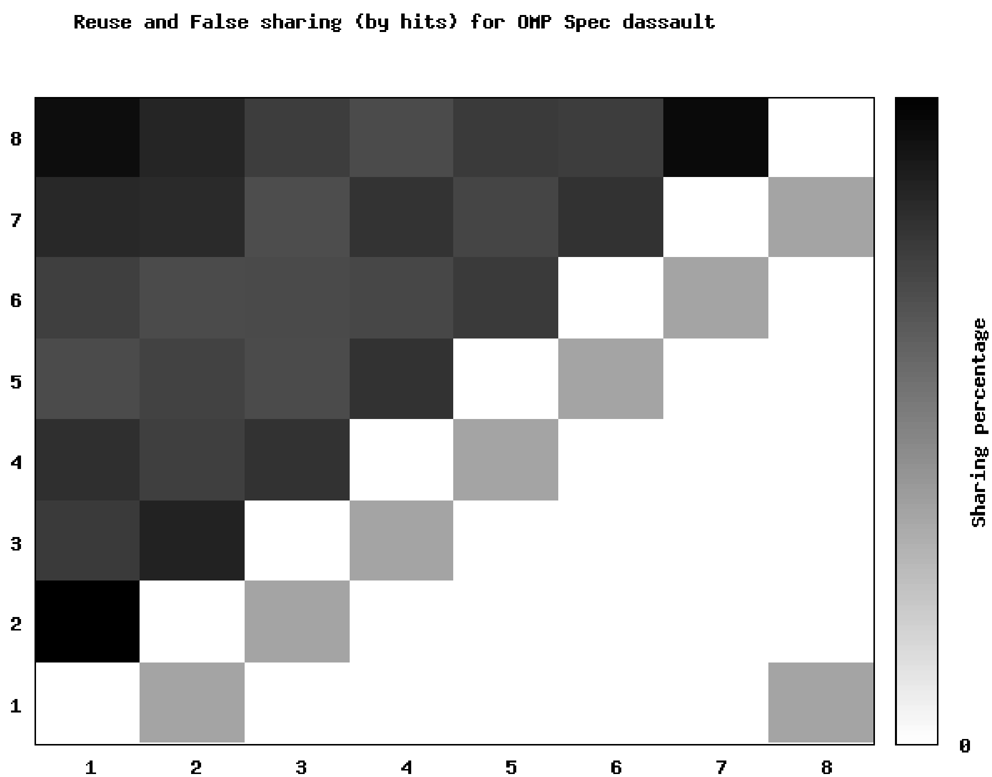


Figure 5.19: Dassault plot

5.6.1 Data sharing

The memory data sharing report sums up the nature and amount of information shared between threads. We consider three types of sharing, exclusive reads (share occurs only on read instructions), exclusive write and a mix of read and write.

```
DO IDO=1,NREDD
INC   =   INDINR(IDO)

HANB  =   AM(INC,1)*PHI(INC+1)  &
+ AM(INC,2)*PHI(INC-1)  &
+ AM(INC,3)*PHI(INC+INPD)  &
+ AM(INC,4)*PHI(INC-INPD)  &
+ AM(INC,5)*PHI(INC+NIJ)  &
+ AM(INC,6)*PHI(INC-NIJ) &
+ SU(INC)

DLTPHI = UREL*( HANB/AM(INC,7) - PHI(INC) )
PHI(INC) = PHI(INC) + DLTPHI

RESI = RESI + ABS(DLTPHI)
RSUM = RSUM + ABS(PHI(INC))
ENDDO
```

Figure 5.20: Original code of the main hot loop of RECOM application. Note that the increment `INC` results from an indirection.

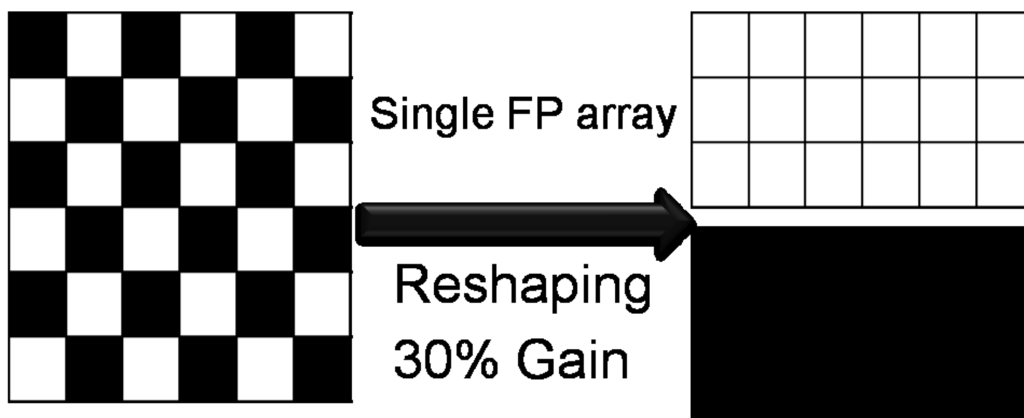


Figure 5.21: On the left, we can see the 2D access pattern iterating only black tiles of a checkerboard. On the left, the splitting transformation solution.30% of gain has been obtained on this loop.

We can sum up these three cases as follow :

- Reuse : detecting data reuse between load instructions exclusively (load from different threads on the same cache line).
- Cache line thrashing :

  detecting potential false sharing (stores from different threads on the same cache line).

  detecting potential cache line thrashing due to load and store instructions using the same cache lines.

A common example of thrashing is using a structure of tables instead of a table

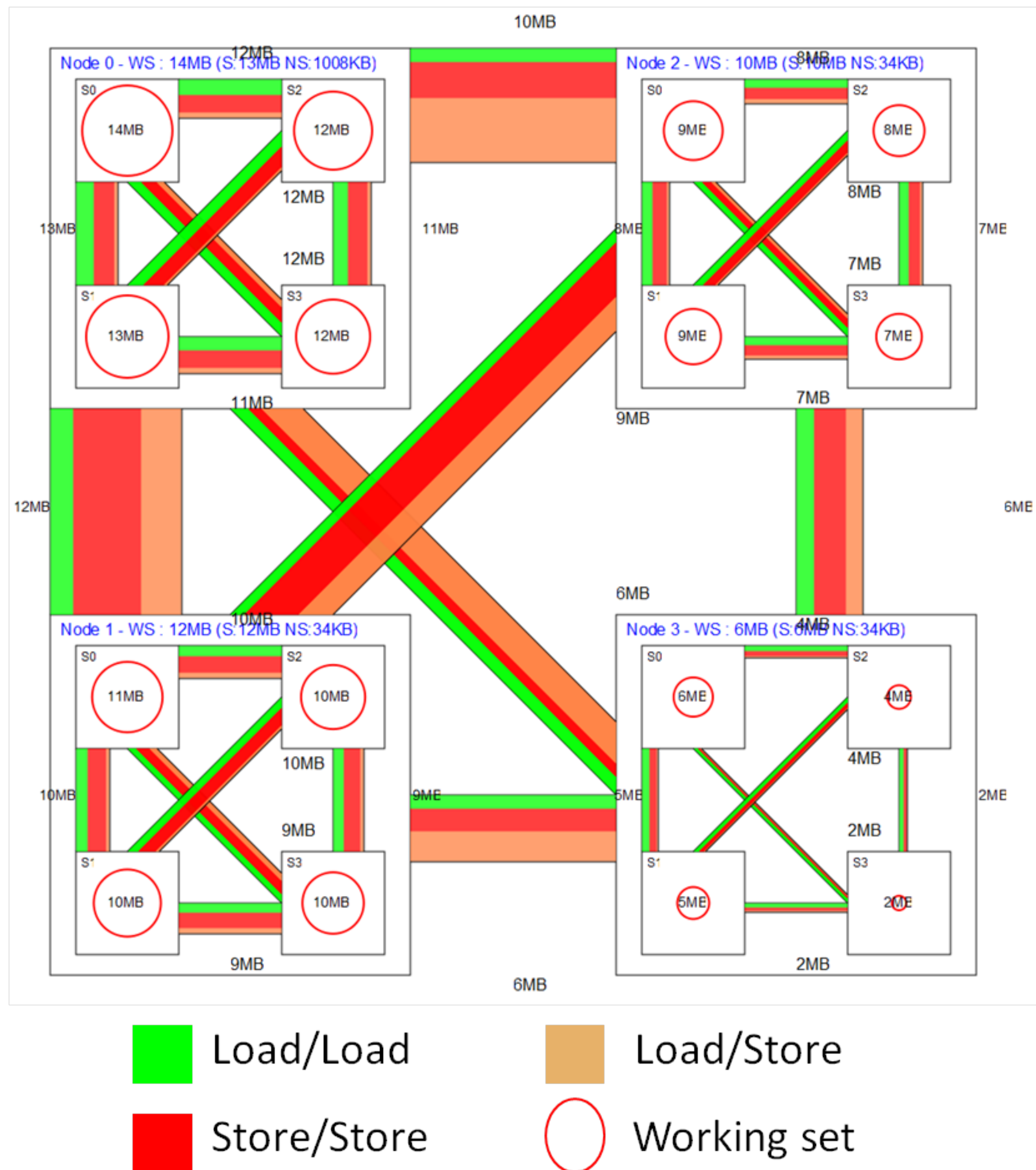


Figure 5.22: LU decomposition application (OpenMP) on a 96 cores machine (4 nodes,16 sockets). Evaluates data sharing between Nodes,Sockets : Working set (shared,not shared) and Coherence based on shared cache lines (worst case).

of structures when multiple threads perform writes. In this case, working with synchronized threads can lead to false sharing.

Based on this categorization we consider the thread that shares the least amount of data with other threads and the thread that shares the most. This provides us with a global metric (used in Figure 5.1).

Modifying OpenMP schedule policy can improve the amount of shared data because it controls the way the work is spread among the threads.

This report is very important when no data sharing is detected because it reveals the amount of space of the cache that is used just like an independent cache (which make it virtually smaller).

After performing this analysis, it is possible to further test some transformations

in order to preview a different sharing scheme. Possible transformations are;

- Swapping threads manually : This analysis is performed again but without starting again the whole process (replay of traces).
- Rearranging threads: Automatically find and swap candidates. Thread swapping is actually a sub-case of this one.
- Reduce the number of threads: In the case of a STATIC scheduling, traces of removed threads can be spread among other threads. For instance if we want to reduce the number of threads from 96 to 56, traces of threads 57 to 96 will be spread over threads 1 to 30.

5.6.2 Workload balancing

It is important to verify that the workload of the application is appropriately spread among all the available threads. This report shows the amount of work (memory accesses) of each thread. OpenMP runtime parameters can help addressing this kind of issues, essentially through four levers, namely, work share strategies, number of threads involved, size of data chunks and affinity. Some applications definitely cannot scale linearly and may waste resources. Depending on how well an application scales, some threads may be freed.

Throughout the evaluation of the NAS Parallel and SPEC OMP 2001 Benchmarks, we have noticed that the most effective strategy remains $STATIC$. In the best scenario, $GUIDED$ schedule performed as well as $STATIC$. Ayguadé *et al.*. [6] also wonders if the schedule clause is really necessary in OpenMP. Besides proposing their own scheduling strategy, they noticed the same behavior in terms of speedup.

Figure 5.6.2 depicts a simple benchmark case with a triangle traversal. From left to right :

- Affinity : Compact (first three columns), Scatter (last three columns)
- Schedule : Static, Dynamic and guided
- Chunk size : 1,2,3,4,6,8,10,12,16

We can observe that $DYNAMIC$ and $GUIDED$ generates a lot of overhead. Besides, we also notice that chunk size affects the execution time.

Figure 5.24 contains a set of examples balancing issue due to a wrong strategy.

If we focus on $SPECOMP$324.*apsi*, we can clearly see a load balancing issue. 16 threads do 50% more work than the others. Figure 5.6.2 reveals that the best execution time is obtained when using 56 threads. We can observe the load is evenly spread among all the threads.

5.6.3 Affinity

Thread affinity is considered between pairs of threads. We compute from the cache simulator the number of accesses of a thread that are data shared with any other thread. More precisely, we compute affinity between threads in terms of the number of accesses for load/load accesses (read-only), write/write accesses and load/write accesses. The first kind of affinity is constructive if both threads are executed on cores that share some cache. For the second affinity, this corresponds to false sharing.

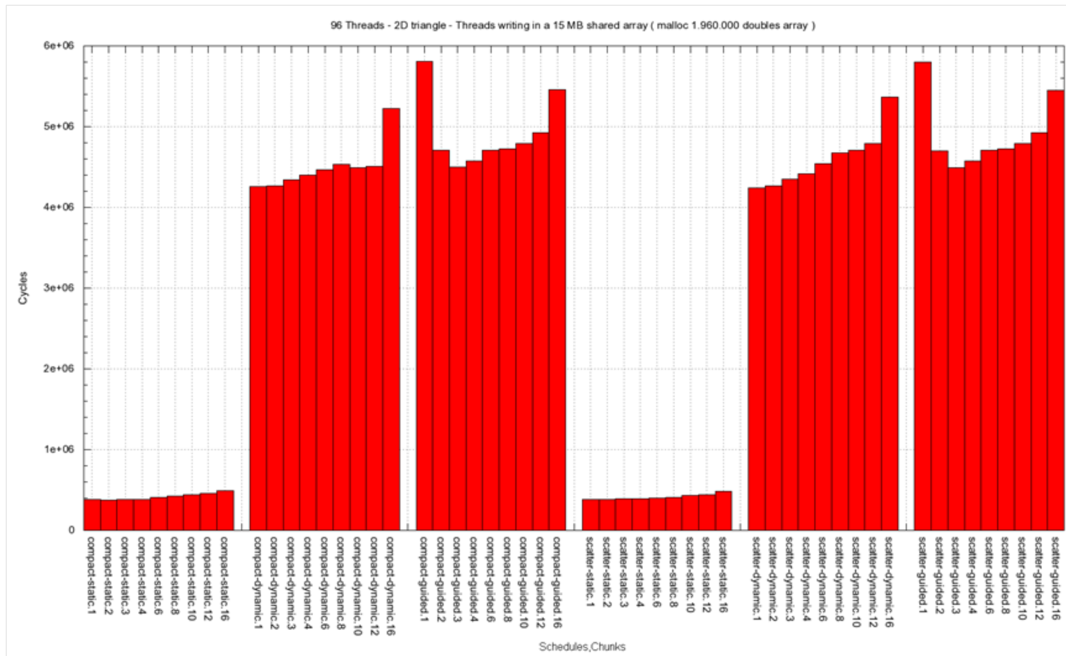


Figure 5.23: Evaluating work share strategies on a simple triangle traversal example on a 96 threads machine. Threads are sharing a 15MB array. X-Axis caries Affinity,Schedule,Chunk size triplets and Y-Axis accounts for executed cycles

Consider two matrices where element i, j represents the minimum number of accesses done by threads i, j on shared data between each thread. We define two sharing types (one in each matrix), load/load accesses and write/write accesses. These matrices are symmetric, and we choose to represent these by a 2D plot. Figure 5.26 shows an example of such a plot.

The top left triangle shows the common written (write/write) data between each thread whereas the bottom right shows the common read (read/read) data. Each intersection of the X and Y axes gives the percentage of shared data between two threads. The more they share, the darker is the intersection (location). The scale value is relative to the maximum percentage on the 2D plot.

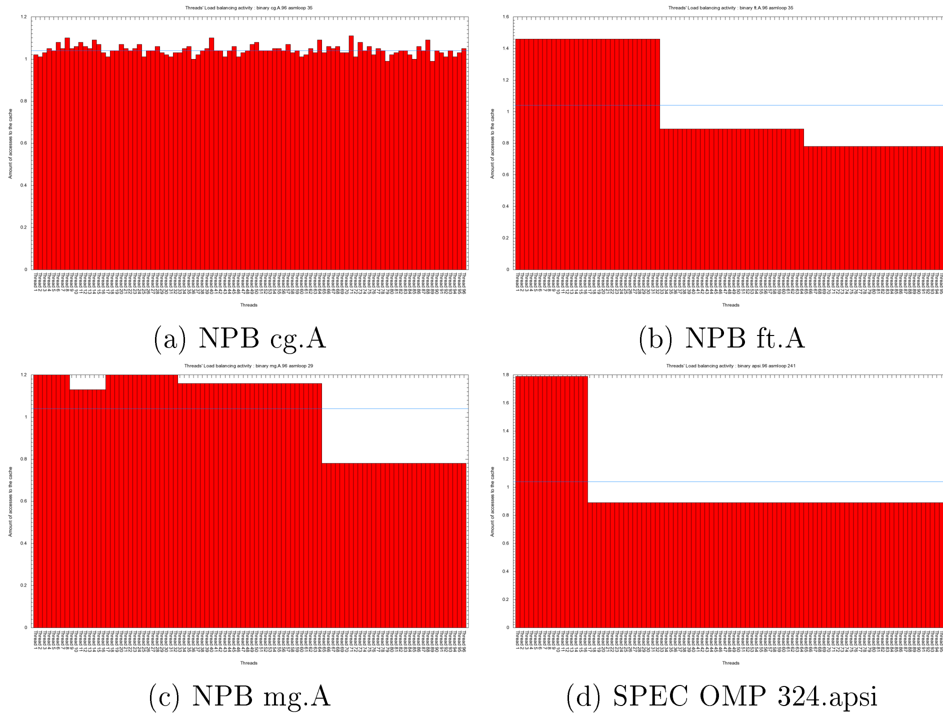
The advantage of such representation is that it exhibits sharing patterns for parallel OpenMP loops that correspond to the memory behavior of threads.

5.6.4 Experiments

Experimental setup Experiments were run on an IBM eX4 machine, with 4 nodes of 4 socket hexa-core 2.66Ghz Dunnington X7460, 3MB L2 and 16MB L3. Nodes are connected through high speed interconnect.

5.6.4.1 SPEC OMP 2001 318.galgel_m

Figure 5.27 shows the data shared among pairs of threads for a loop of galgel benchmark. As only the lower diagonal is represented, this shows that read-only data is shared. The number of accesses on shared data remains evenly distributed for all threads even when the number of threads increases, due to a large loop iteration count.



(a) NPB cg.A

(b) NPB ft.A

(c) NPB mg.A

(d) SPEC OMP 324.apsi

Figure 5.24: Examples of workload balancing report between threads. X-axis refers to memory accesses (% of total accesses) and Y-axis to the work of each thread (from 1 to 96). The blue line represents the amount of accesses that each thread should do in order to have a uniform access over all the threads.

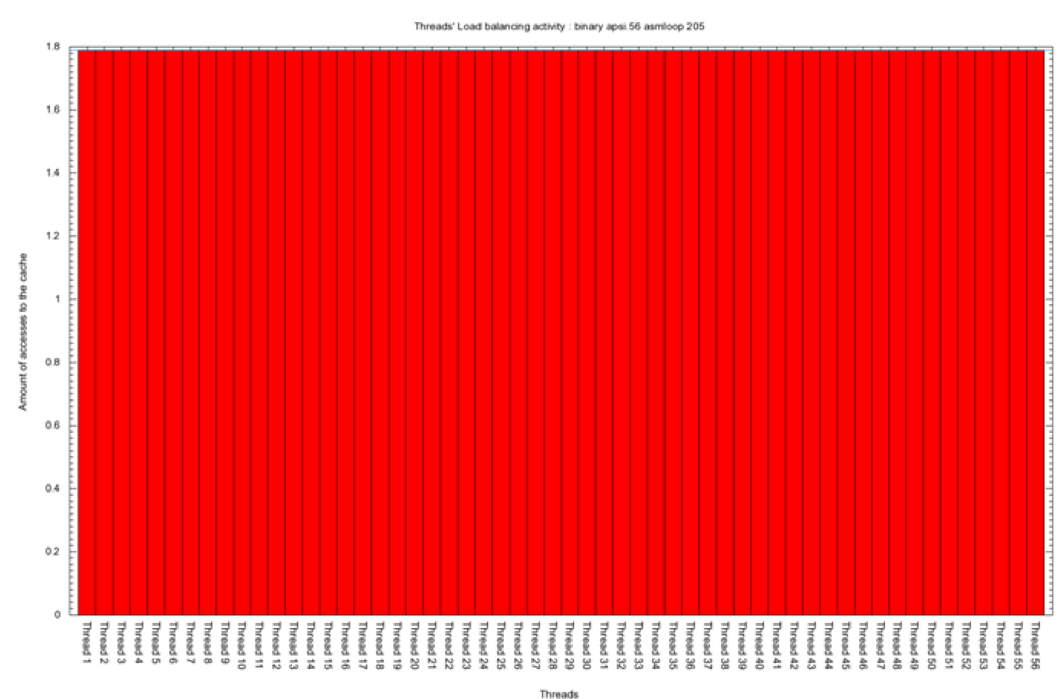


Figure 5.25: SPEC OMP 324.apsi run on 56 threads

Figure 5.6.4.1 shows a pattern in the upper diagonal where all threads write

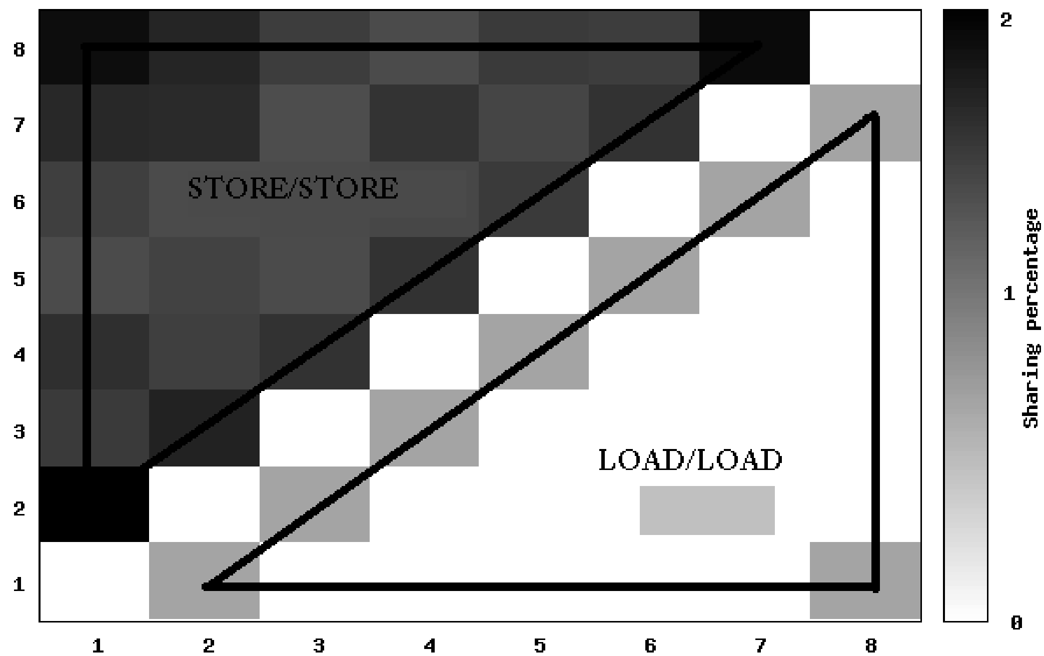


Figure 5.26: Example of a sharing data plot

some shared data. Actually, this is a case of false sharing. There is a high cache coherence cost associated to such scenario. For the STATIC scheduling, large chunks are given to each threads and the false sharing effect is limited to the extremities of the chunks. For GUIDED and DYNAMIC strategies, the chunks are smaller, of varying sizes (for GUIDED), and any thread can take the chunk following the chunk of another thread. This implies that the number of potential false-sharing situation has increased, and threads share a cache line with another thread.

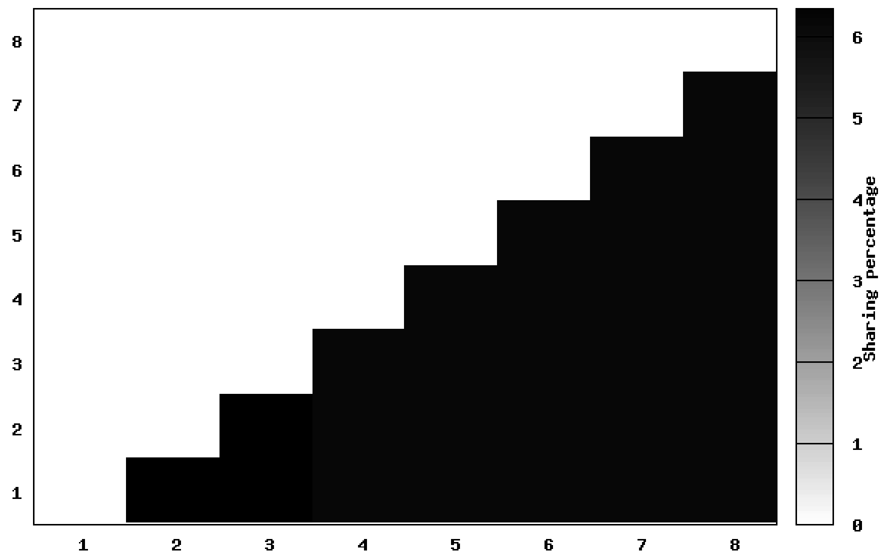
For 8 threads and DYNAMIC schedule, a padding would help to solve this issue, or a fixed chunk size, multiple of cache line. But this is clearly no longer possible for GUIDED and is not a scalable solution as the number of threads increases. Clearly, this situation suggests for the STATIC scheduling.

5.6.4.2 SPEC OMP 2001 224.apsi_m

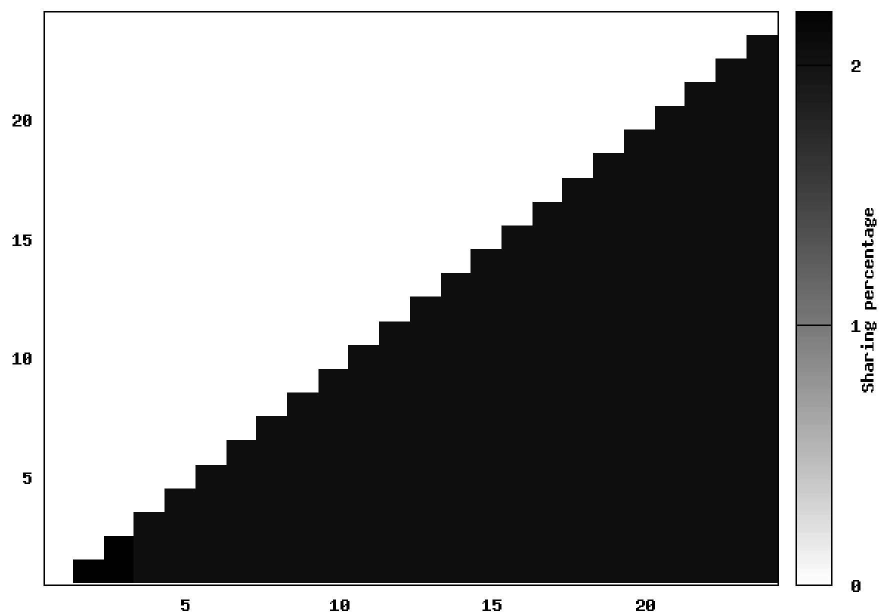
Considering the plots of Figure 5.29, the plot Figure 5.29.a shows a pattern representative of a STATIC distribution of the iterations among the threads. Each thread is given a contiguous chunk of iterations. Shared data are in fact shared cache lines, but there is no real common data between the threads. The percentage is very low and represents the percentage of accesses occurring at the extremities of a chunk. Figure 5.29.b and 5.29.c shows an unbalanced pattern between threads. Figure 5.29.d reveals different levels of sharing (here writes) on a 24-cores machine. Such differences are due to type of scheduling used, $DYNAMIC$.

5.7 Data Reshaping

We previously discussed the major role of the selection of data structures and the way they are traversed. In this section we will focus on OpenMP codes and detail



(a)



(b)

```
!$OMP DO
  DO LM = 1, K
  A(1:K,LM) = A(1:K,LM) - MATMUL( HtTim(1:K,1:K), Poj3(1:K,LM) )
  END DO
!$OMP END DO NOWAIT
```

(c)

Figure 5.27: Shared data plots for galgel benchmark, syshtn.F loop at line 98. The lower diagonal pattern indicates read-only shared data among all threads. In X and Y axis, the ids of the threads. (a) For the 8-core Nehalem, each thread has 6% of its accesses on shared data with any other thread (b) For the 96-core Dunnington (only 24 threads are used). Each thread has 2% of its accesses that are shared with any given thread.

our approach on how to find out inefficient data layouts. This approach handles

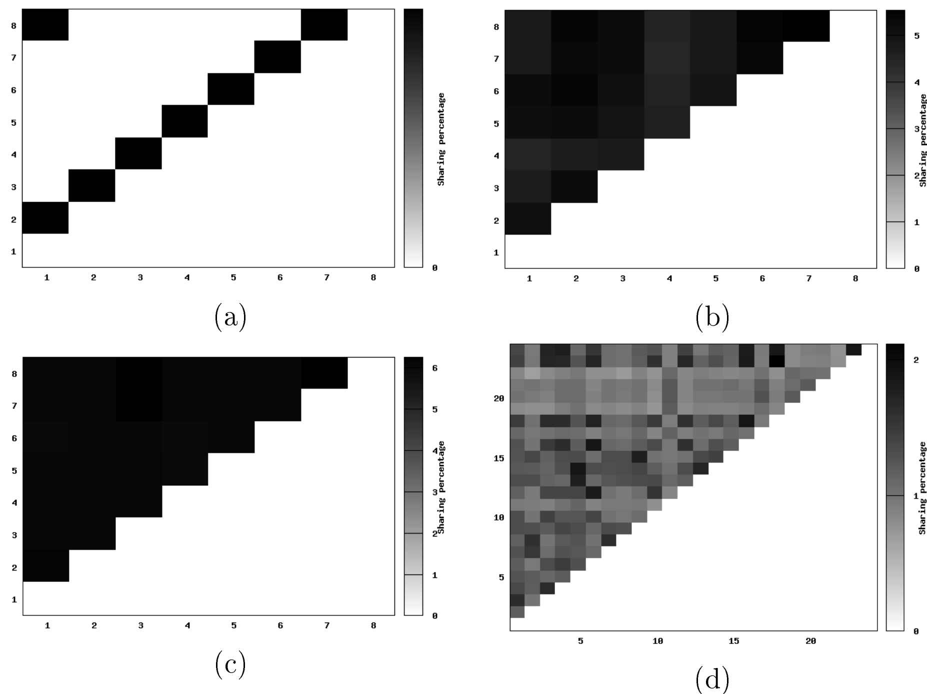
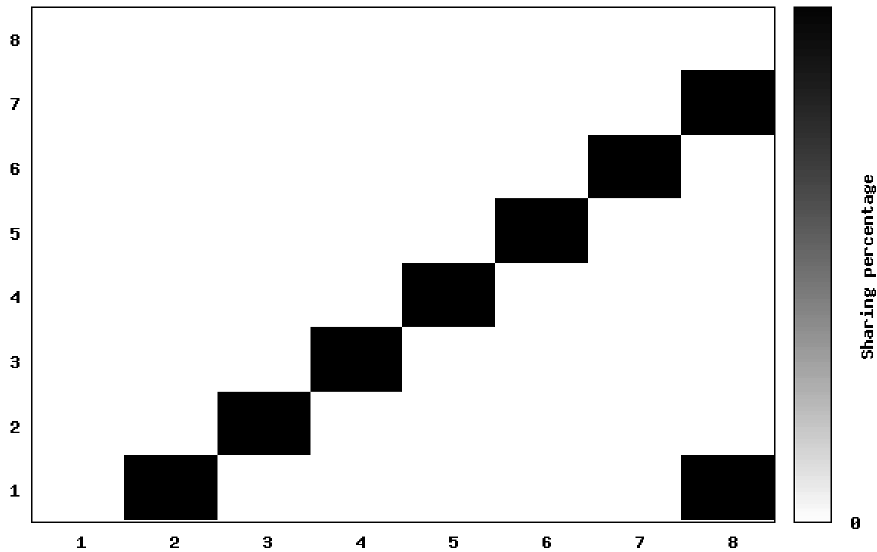


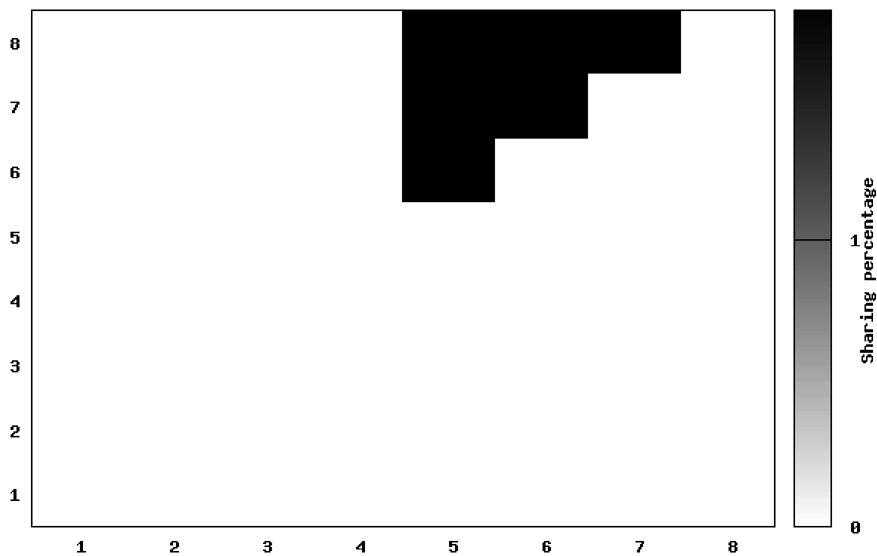
Figure 5.28: Shared data plots for galgel benchmark and loop at line 68 showing false-sharing among threads, according to different scheduling strategies and architectures: (a) STATIC, on 8-core Nehalem (b) GUIDED, on 8-core Nehalem (c) DYNAMIC, on 8-core Nehalem (d) GUIDED, on 96-core Dunnington, with only 24 cores used.

multidimensional arrays, generalizing structure splitting, and takes into account possible interactions between threads. Spatial locality is then enhanced and the working set for each thread is reduced.

Data structure reshaping and splitting is an approach to reduce the number of memory transactions by enhancing spatial locality. The idea of these techniques is to put in adjacent memory locations data accessed consecutively. At compile time locality analysis generally requires inter-procedural alias-analysis [123] and is hampered by pointers, indirections or complex control flow and may result approximate results. Moreover for parallel codes, the analysis is made more complex due to the number of interactions among threads, some of them depending on runtime decisions. In OpenMP programs, the number of threads, the scheduling strategy and the mapping of threads on cores can be chosen dynamically for instance. At runtime, locality analysis relies on execution traces capturing the flow of memory references. It is not hampered by program constructions or runtime decisions. However it has to cope with two issues: Parallel executions may generate huge traces, even using compression techniques; The locality analysis is based on data collected in a few runs, meaning that data layout transformations proposed to the user have to rely on a generalization of the results of the analysis. These two issues become crucial when dealing with large multidimensional data structures, widely used in scientific programs. Few works have focused on structure reshaping based on parallel trace analysis, but to the best of our knowledge, no work has tackled a more general problem for multidimensional data structures: regrouping elements of such data structure requires to reorganize the multidimensional layout as a whole, needing an



(a)



(b)

Figure 5.29: Shared data plots for apsi benchmark. (a) This lower diagonal pattern corresponds to a STATIC distribution of loop iterations, in large chunks. A cache line is shared only at the border of these chunks (b) An example of unbalanced shared data, here between scalars.

appropriate representation for such data structure.

Most OpenMP applications mainly use the data parallel work-sharing construct (OpenMP 2.5). Tasks have been introduced recently and programmers seldom use them. At assembly level, arrays are loaded by a single or multiple instructions and each instruction is executed by all the threads. Since our traces are per thread and per instruction, we can detect an accessed region by merging the traces of the threads for each instruction.

To illustrate our algorithm, we will be using the 312.swim benchmark from SPEC OMP 2001 Medium (Loop at line 119).

5.7.1 Related work

There have been many studies on program data locality analysis. We present in this section an overview of these related works.

Many of the prior works focus on single threaded codes. Among them, one of the approaches studied relies on safe data transformations for general purpose programs: Zhao *et al.* [123] propose in Forma a framework for array structure reshaping, reorganizing arrays to improve data locality. They split large data types into smaller ones to improve cache locality and hardware prefetching. Finding all accesses to the same fields and same structure of an array is first obtained through an inter-procedural field-sensitive alias-analysis. Structure fields are then grouped according to the result of a runtime analysis. Fields can be grouped when they are accessed to similar frequency (hot fields are then put in a structure different from cold fields), when they are accessed contemporaneously, or they are always split into different structures. Curial *et al.* [23] propose a Memory-Pooling-Assisted data Splitting (MPADS), a framework performing data structure splitting with memory pooling. The user is assumed to allocate data of different type in different pools and a pointer-analysis ensures that splitting is valid. In a similar way, Lattner an Adve [64] controls the layout of heap-based data structures in pointer intensive codes through different memory pools. These methods focus on structure splitting. The general case of array splitting (where multiple array elements are split from the other elements of the array) is not considered.

Asher and Rotem[10] describe a method changing data structures, based on memory profiling. Memory accesses are traced and abstracted by an address interval and a single stride per instruction. By analyzing these abstractions, the data layout is transformed in order to reduce bank conflicts, assuming that memory addresses are the same from one run to the other. Zhong *et al.* in [124, 125] use trace-level reference affinity and profiling-based method for array regrouping and structure splitting. These works do not capture multiple strides that can appear in multidimensional arrays and are also for sequential codes.

Lin and Yew [67] propose a framework for structure layout optimization and array flattening. Through alias analysis, they detect and transform multidimensional arrays defined through multiple-level pointer dereference into single-level dereference, similarly to array regrouping techniques. Linearized multidimensional arrays are not studied however.

5.7.2 Overview

The main algorithm is sketched in Algorithm 2.

```
trace ← ReadApplicationTrace()
lmt ← MergeThreads(trace)
lmi ← MergeInstructions(lmt)
oa ← FindOverlappingAreas(lmi)
DetectInefficientAccessPatterns(lmi,oa)
```

Algorithm 2: MAIN

At the beginning, the program trace is read and organized into threads and then instructions. The first step, depicted in Figureokula 3, consists in finding similarities

between the memory areas traversed by all the threads. Then, if applicable, these resulting regions are merged between instructions. The following step finds the dependencies between regions. With all this information at hand, we can finally detect issues.

5.7.3 Reconstructing shared data structures

In order to find out the shared data structures by all threads we need to merge their traces into one region, when it is possible. To achieve this goal, we first need to merge the traces between threads and then between instructions (loading a data structure may require multiple instructions).

5.7.3.1 Merging regions accessed by each thread

For each instruction, each thread may have several polyhedrons describing the traversed memory region. Before comparing the traces of each thread, we need to sort the polyhedrons according to their starting value since the merged region is built by adding a new thread each time (*NewThreadTrace*. When comparing multiple threads, some of these may have additional polyhedrons not found in the others. In order to avoid these artifacts from stalling our detection process, we use a position algorithm similar to the $diff$ algorithm, thus aligning compatible polyhedrons. Then, if the polyhedrons have the same form, we add them to a merge object. By form, we mean structure (depth and pattern). For instance, Figure 5.31.a shows eight compatible polyhedrons.

Figure 5.30 presents the typical access patterns when multiple threads share the same data structures. We can distinguish two different types of accesses

- Accesses are alternated, either overlapping or consecutive (e.g. P1 and P2)
- Block fashion, either consecutive or spaced by a constant value (e.g. P3 and P4)

We use a variable that keeps track of the type of access to be able to merge, at the end of the process, the polyhedrons present in the merge object. The are three possible types :

- follows means that consecutive polyhedrons do not overlap. Each new thread trace starts where the previous finishes.
- overlap indicates an interleaved polyhedrons.
- distinct means that the polyhedrons have no common memory references.

Finally, polyhedrons stored in the *merged_region* object are merged according to the merge type explained above.

Figure 5.31.b shows an example of such a merged area. It is actually an array divided between all the threads (across iteration space).

As noted previously, a data structure may need multiple instructions to be loaded. According to that, we need a further step to merge,if applicable, previously discovered regions between instructions.

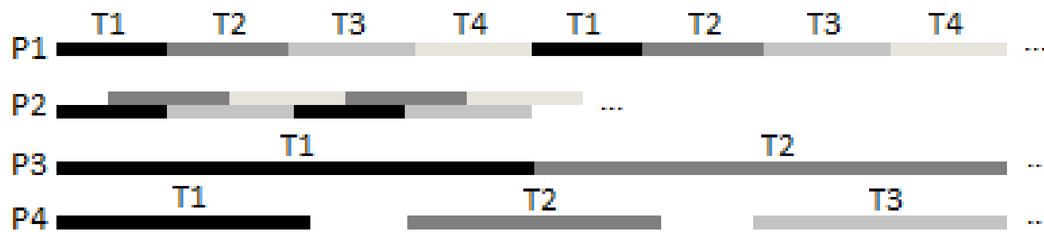


Figure 5.30: Typical access patterns of arrays shared by multiple threads. Each thread T1 to Tn reads a subset of the same array. P1 to P4 represent common access patterns : interleaved, interleaved with overlap, consecutive and separate (structure fields)

```
input  : A list of thread traces each containing polyhedrons
output: Merged region

SortThreadTracesByInitialValues();
foreach element in the list of polyhedrons do
    FindSameStartingPolyhedron() ;
    merged_region ← NewThreadTrace();
    tnext ← NewThreadTrace();
    while New thread trace available do
        if HaveCompatibleForm(merged_region,tnext) then
            if Overlaps(merged_region,tnext) then
                if Follows(merged_region,tnext) then
                    merge_type ← follows;
                else
                    merge_type ← overlap;
                end
            else
                merge_type ← distinct;
            end
            merged_region ← merged_region ∪ tnext;
        else                                             /* stop merging */
            merged_region ← ∅;
            merge_type ← none;
        end
        tnext ← NewThreadTrace();
    end
end
if merged_region ≠ ∅ then
    Merge polyhedrons of merged_region according to merge_type;
end
```

Function MergeThreads

5.7.3.2 Merging regions between instructions

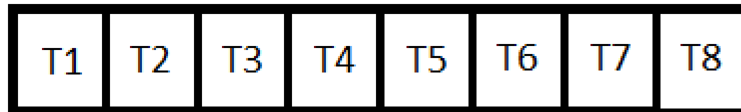
The *MergeInstructions* function is actually quite similar to *MergeThreads*. There are only few differences when

```
Instruction 31
Thread1
for i0 = 0 to 799
 for i1 = 0 to 166
  for i2 = 0 to 165
   val 354271968 + 30416*i1 + 64*i2
Thread2
for i0 = 0 to 799
 for i1 = 0 to 166
  for i2 = 0 to 165
   val 359351440 + 30416*i1 + 64*i2
thread3
for i0 = 0 to 799
 for i1 = 0 to 166
  for i2 = 0 to 165
   val 364430912 + 30416*i1 + 64*i2
Thread4
for i0 = 0 to 799
 for i1 = 0 to 166
  for i2 = 0 to 165
   val 369510384 + 30416*i1 + 64*i2
Thread5
for i0 = 0 to 799
 for i1 = 0 to 166
  for i2 = 0 to 165
   val 374589856 + 30416*i1 + 64*i2
Thread 6
for i0 = 0 to 799
 for i1 = 0 to 166
  for i2 = 0 to 165
   val 379669328 + 30416*i1 + 64*i2
Thread 7
for i0 = 0 to 799
 for i1 = 0 to 165
  for i2 = 0 to 165
   val 384748800 + 30416*i1 + 64*i2
Thread 8
for i0 = 0 to 799
 for i1 = 0 to 165
  for i2 = 0 to 165
   val 389797856 + 30416*i1 + 64*i2
```

(a)

```
Instruction 31
for t = 0 to 7
  for i0 = 0 to 799
    for i1 = 0 to 165
      for i2 = 0 to 165
         val 354271968 + 5079472*t + 30416*i1 + 64*i2
```

(b)



(c)

Figure 5.31: Trace of each thread for one instruction (a). Resulting polytope describing the shared region (b). Representation of the shared region (c).

- merging is done between instructions
- only instructions of the same type can be merged (either load or store)
- when failing at combining two instructions, the process continues (step by step) until all instructions are processed.

Loop unrolling is a common example of regions spread among multiple instructions. After sorting instructions by offset, the clustering process extracts a first candidate group for merging (Figure 5.32.a).

Figure 5.32.b illustrates the resulting polyhedron according to the previous algorithm.

```
Insruction 31
for i0 = 0 to 799
  for i1 = 0 to 166
    for i2 = 0 to 165
       val 354271968 + 30416*i1 + 64*i2
Insruction 34
for i0 = 0 to 799
  for i1 = 0 to 166
    for i2 = 0 to 165
       val 354271984 + 30416*i1 + 64*i2
Insruction 37
for i0 = 0 to 799
  for i1 = 0 to 166
    for i2 = 0 to 165
       val 354272000 + 30416*i1 + 64*i2
Insruction 40
for i0 = 0 to 799
  for i1 = 0 to 166
    for i2 = 0 to 165
       val 354272016 + 30416*i1 + 64*i2
```

(a)

```
for t = 0 to 7
  for i0 = 0 to 799
    for i1 = 0 to 166
      for i2 = 0 to 165
        val 354271968 + 16*t + 30416*i1 + 16*i2
```

(b)

Figure 5.32: Candidate group of instructions for merging (a). Merged region (b)

5.7.4 Analysis and hints generation

Once we have been able to recognize some data structures, we can start the inspection process in order to detect inefficient access patterns leading to poor locality. Data prefetching is a key factor to achieve performance. In out-of-order processors, this task is handled by the hardware prefetchers that are very sensitive to strides (regular accesses). When exceeding some architectural limits, one can face a dramatic performance slow down. For instance, Intel processors have the ability to recognize strides within +/-2 KB. Moreover, when exceeding the size of memory pages, you have to pay for an additional penalty due to TLB misses.

5.7.4.1 Changing traversal order

A wrong traversal order on a data structure is detected when access patterns appear reversed on a particular region. By reversed we mean a decreasing access pattern, the biggest stride being on the innermost dimension. From this point, we can tell the user to change his data structure in order to achieve a proper traversal. However, we must first ensure that we are not introducing a performance slowdown. Indeed, if the same array is used elsewhere in another fashion (not the same traversal order), we

must evaluate if it is worth the change. In order to make this decision, we use a cost function that evaluate the cost of each usage of this array. It is based on the number of memory references and the implied working set (cache lines involved). The cost function takes as input a polyhedron (access pattern and number of accesses) and the associated instruction type. Based on that we determine how many elements a cache line will contain for this instruction and consequently how many cache lines will be needed. The cost is thereby represented by the number of cache lines. If the cost function validates the transformation, we can provide the user with hint telling him to proceed to a data structure transformation on the candidate region.

5.7.4.2 Splitting

Affinity splitting Grouping certain (hot) fields used together within a loop improves data locality. We inform the user about access patterns that only accesses parts of a region. This grouping may concern either a few fields or whole vectors. In practice, the user is provided with a pattern that help him reshape his data structures. For instance, if we have the following pattern :

```
for i0 = 0,i0max
  for i1 = 0,i1max
    for i2 = 0,i2max
      val + a1*i0 + a2*i1 + a3*i2
```

then we will propose another pattern exposing the structure to adopt in order to perform the splitting.

```
a2 = a3 * (1+i3max), a1 = a2 * (1+i2max)
```

False sharing False sharing occurs when multiple threads access non overlapping regions of the same cachelines with at least one of the threads writing to it. Behind the scene, this means sharing cache lines and invalidating each others data since cache coherence comes into play. Regions built by merging polyhedrons separated by only a few cache lines (short distance between starting offsets) will experience false sharing. The user is told about this issue and we propose, either adding padding in his data structure or splitting the involved data structure in order to isolate conflicting accesses.
All of these hints provides the programmer with high level information on how to apply transformations on his code in order to improve locality, thus enhancing performance.

5.8 Conclusion

In this chapter we proposed a new approach and analyses to characterize the memory behavior of multi-threaded OpenMP applications. This approach relies on memory access traces, compacted on-the-fly per thread and per instruction into a union of polytopes. Our analyses can be achieved for each source loop and each function (considering the availability of debug information). We use this polyhedral representation in conjunction with a cache simulator to identify multiple issues. All thread accesses are mapped into an associative cache and interactions between threads are extracted, identifying potential false-sharing situations, workload imbalance. We

have also proposed a 2D graphical representation of pairwise thread interactions through shared data, revealing some interactions due to scheduling strategies. In particular, in the case of an array written to by all threads, DYNAMIC scheduling strategy entail false-sharing situation between all threads. We also detailed how access patterns can lead to a poor locality. Issues involving addresses, mainly related to cache structures, such as alignment, bank conflicts, set associativity, were discussed. We finally have focused on multidimensional data structure reshaping in order to enhance spatial locality. We have shown how to use such memory traces to generate reshaping hints improving locality.

Chapter 6

Conclusion

This dissertation presents the work accomplished during my thesis. Chapter 2 provides an overview of the growing complexity of modern architectures. There is clearly a need for parallelism in order to harness the horsepower provided by current clusters (of machines). An overview of performance analysis tools is also provided. Each addresses multiple issues at different levels of granularity. In Chapter 3 we present MAQAO tool and its top-bottom performance analysis approach, i.e. coarse grain to fine grain. Static and Dynmaic approaches are coupled to provided a better understanding of multifaceted problems usually met in scientific applications. Chapter 4 presents a domain specific instrumentation language that takes advantage of this coupling. Its goal is to provide a mean to easily build performance evaluation tools based on a simple but rich scripting language (static analysis through MAQAO API). Dynamic analysis is performed by the probes inserted in the instrumentation file. At fine grain, memory related issues remains the most important factor of performance degradation. In the sequel, we decided to focus on the characterization of the memory behavior of multithreaded applications, presented in Chapter 5.

The objectives set in the introduction have been reached. All the contributions of this thesis have been implemented in our MAQAO tool, thus providing new valuable features.

The remaining of the conclusion of this dissertation is organized as follows. First, contributions are presented. Then, we give a brief description of the necessity of having a methodology when using performance evaluation tools. Finally, extensions of the current modules and research lead are proposed.

6.1 Contributions

There are two main contributions detailed in this dissertation along with two other contributions, which are more technical and only briefly described in a part of Chapter 3. Note that the contributions do not appear in chronological order

The first main contribution, MTL, provides a mean to characterize the memory behavior of multi-threaded applications. In particular, we have focused on a shared memory programming model, OpenMP. Four analysis (each providing metrics) are presented. The first one deals with the analysis of memory access patterns of instructions at the thread level. It makes it possible to find out bad access patterns that will lead to slowdowns and can presents opportunities for software prefetching. The three others provide a mean to understand the interactions between threads. the data sharing analysis shows the amount and type of information exchanged between threads. The workload balancing analysis reveals any potential imbalance between threads due to a bad scheduling scheme. Finally, the thread affinity analysis allows to update Data sharing and Workload balance analyses by changing the pinning of threads and without replaying all the traces (one trace per thread).

This is made possible because all the important data is stored in the internal cache structure. Another important feature in the ability to project performances on hypothetical architectures. MTL actually uses a machine file description to project its results. Thus, it is possible to virtually increase the number of threads, and observe the behavior of the application. Our experiments show how MTL helped in detecting and fixing memory access issues.

The second main contribution, MIL, is a domain specific instrumentation language that simplifies the creation of performance evaluation tools. The basic idea is to provide a mean to easily build new performance tools that suit specific needs, with the lowest possible overhead. The major achievement related to MIL was the integration of the MAQAO tool in the TAU Parallel Performance System toolkit as part of the Program Database Toolkit (PDT). More precisely, a new script using MIL was introduced. Our experimentations revealed that MIL outperformed its state-of-the-art competitor, Dyninst.

The third contribution, MAQAO Profiler, is a profiling tool, based on MIL, that provides (gprof-like) function and loop level timing. It also integrates a beta feature handling OpenMP parallel regions (GCC and ICC OpenMP runtimes). Compared to the closest competitor, Intel compilers' profiler, we are able to use lighter probes and handle OpenMP applications (interleaved functions produced by ICC).

The last contribution is a modular scripting infrastructure that allows an easier access to the MAQAO low-level API and the addition of new plugins. This higher level API combined with the flexibility of the scripting language (LUA) enables end-user developers to easily achieve a better productivity when developing a plugin. MTL and MIL, among others, are based on this scripting infrastructure.

All the contributions of this thesis have been implemented in our MAQAO tool, thus providing new valuable features.

6.2 Performance evaluation tools and methodology

I personally don't believe in push button approaches, that would automatically detect and fix issues. Performance tuning involves two major parts, namely, finding issues and solving them. Many tools only provide a mean to pinpoint issues but not to solve them. Sometimes a feedback or hints are returned but it is actually a set of generic advices not very easily applicable. Having a set of tools is a strong requirement when going through the process of performance tuning. But I believe that a methodology must articulate these tools. In chapter 3 we described a top-bottom methodology which starts from coarse grain analyses down to fine grain analyses. In our case the transition between both levels was directed by the characterization of the main issues, namely, memory or compute bound. From a practical point of view, the goal is to guide end-user developers through specific tools that should address the observed issues. Depending upon the estimated remaining potential gain, an end-user developer can choose whether or not to keep continuing.

6.3 Future work and research leads

Future objectives will be directed towards a deeper understanding of the current problems (decribed in this thesis) and new research leads that may anticipate issues that will probably occur with the advent of the Exascale Era.

First, possible extensions to current MIL and MTL contributions are presented. Then research leads are proposed.

6.3.1 Extending MIL

The first valuable extension of MIL would be to natively support all OpenMP and MPI events. Also, an iterative instrumentation approach in order to automatically take into account previous instrumentation sessions, would be of a great help. It may be a key feature in designing systematic performance evaluation tools, refining at each step the identification of performance issues.

The language itself would need some extensions:

- Object oriented style: The language could be further simplified by adopting an object oriented style, which is more effective than the current tables definitions. Also, the current implementation only considers the default hierarchical event evaluation process. An object oriented style would allow the specification of the relations existing between events.
- Conditional probes: Currently, conditional probes are used internally by MIL. Exposing conditional probes would add a mean to control the execution of probes. A practical usage would be sampling.
- Hardware performance counters: A native support for performance hardware counters collection would avoid understanding and using external libraries.
- Timers: Add built-in function that would provide a mean to get timing information between two events.
- User-defined events: To permit the extension of the language in order to take into account events that are not natively available through the language, a user-defined event feature should be added.

6.3.2 Extending MTL

The current cache simulator used in MTL is a simplified cache simulator. It has no replacement policy and is mainly used to detect the interactions between threads. The cache is filled sequentially with each thread trace. An extension would be to add a replacement policy and replay each thread in parallel.

The second extension relates to data reshaping that needs additional testing and developments. An interesting approach would be to give the ability to the end-user developer to register his data structures. Based on that information, we could then have a more global view of data structures and provide more accurate hints based on the access patterns analysis. If no user information is available, it would be suitable to develop a method based on debug (dwarf) information in order to establish a link between assembly instructions and variables defined in the source code.

MTL has been tested on a 96-core machine. It would be interesting to test it on the new Intel MIC architectures.

6.3.3 Research aspects

MTL does not cover all the memory issues on all know architectures. As a consequence, many aspects need to be investigated. For instance, it could be interesting to examine the possible extensions to distributed (e.g. MPI) or PGAS (e.g. UPC) memory models.

An import aspect to investigate would be the connection between MTL and runtimes. For instance being able to retrieve scheduling and chunk size information from an OpenMP runtime would enable MTL to understand how data structures are split. Thus, being able to provide more accurate hints. The other way is also interesting to consider because it could pass information collected by MTL to the runtime.

MTL lacks temporal locality information because it does not store time information. Keeping precise timestamps information make the compaction of generated data impossible. An interesting idea would be to design a trace format that would provide a tradeoff between the lossless compression used in MTL and precise time information. As a consequence, a temporal dimension could be added to our memory related analyses.

Additional work on access patterns could allow to determine accessed regions of memory for a portion of code. Data prefetching could then be performed for those regions. This is very useful when considering local memories. More precisely, if data transfers have a huge cost compared to computations, then prefetching memory regions is a way to cover that cost.

We previously evoked MPI when considering memory issues. It could be interesting to check if our method could be applicable to MPI communications. Our algorithms may not be adapted but the same idea would remain: detecting patterns and compressing information.

As a final research lead, we could think about an energy-oriented MAQAO. While studying MTL, we saw that some applications could not scale to the available nodes. Reducing the number of cores required to perform the same amount of work is one example of energy efficiency. Beyond this example, we could consider a static model including an energy cost when analyzing the quality of code. A dynamic model could also be added to refine and confirm the static model.

Appendix A

Appendix A: MAQAO scripting examples

This appendix presents a few examples showing how to use the MAQAO Lua API.

A.1 Example 1 : objdump-like binary desassembler

This example prints assembly code of a given application like objdump.

```
p = project.new("objdump");
if (arg[1]== nil) then
   print("Usage: objdump <binary-file>");
else
   a = p:load(arg[1],0);
   for fct in a:functions() do
      print(fct:get_name()..":");

      for b in fct:blocks() do
         for ins in b:instructions() do
            print(string.format("%x",ins:get_address()).."\t"
                ..string.format("%8s",ins:get_coding()).."\t"
                ..ins:tostring());
         end
      end
   end
end
```

A.2 Example 2 : Printing function names of a binary

This example prints the name of all functions of a binary

```
p = project.new("objdump");
if (arg[1]== nil) then
   print("Usage: objdump <binary-file>");
else
  for f in bin:functions() do
    print (f:get_name());
  end
end
```

A.3 Example 3 : Maximum number of instructions in innermost loops

This example shows how to find the innermost loop that has the higher number of instructions. It prints the MAQAO id of the “winner” loop along with its beginning source line.

```
p = project.new("objdump");
local max_nb_insns = 0;
local loopptr;

if (arg[1]== nil) then
   print("Usage: objdump <binary-file>");
else
   local a = p:load(arg[1],0);
   for fct in a:functions() do
      for l in fct:innermost_loops() do
local nb_insns = 0;
for b in l:blocks() do
  for ins in b:instructions() do
    nb_insns = nb_insns + 1;
  end
end

if(nb_insns > max_nb_insns) then
  max_nb_insns = nb_insns;
  loopptr = l;
end
      end
   end
end

print("Loop id "..loopptr:get_id().." (starting at source line "
      ..loopptr:get_src_line()..") has "..max_nb_insns.." instructions");
```

Appendix B

Appendix B: STAN dynamic extension

In this appendix, we will describe the Dynamic extension tool of the STAN module. It is used to compare statically predicted results by STAN and the measured results. We will consider a binary file containing one loop in the main function.

The following MIL script is used to obtain the number of cycles needed to execute the loop along with the number of instances.

```
mil.data.loop_meta_info = Table:new();
mil.data.instru_loop_counter = 0;

function udf_exec_at_instru_exit()
  mil.data.loop_meta_info:save_output("loop_meta_info.cycles.lua",
                                      "loop_meta_info_cycles");
end

function udp_get_loop_newunique_id(loop)
  local fct_name       = loop:get_function():get_name();
  local maqao_loop_id  = loop:get_id();
  local lfm            = mil.data.loop_meta_info;
  local new_instru_lid ;

  if(lfm[fct_name] == nil) then
    lfm[fct_name] = Table:new();
  end
  new_instru_lid = mil.data.instru_loop_counter;
  if(lfm[fct_name][new_instru_lid] == nil) then
    lfm[fct_name][new_instru_lid] = maqao_loop_id;
  end
  mil.data.instru_loop_counter = mil.data.instru_loop_counter + 1;

  return new_instru_lid;
end

function udp_get_loop_currunique_id(loop)
  return mil.data.instru_loop_counter - 1;
end

function udf_isinner_oneblockloop(loop,gvars)
  if(loop:is_innermost() and loop:get_nblocks() == 1) then
    --print("Loop "..loop:get_id().." is innermost and one-block");
    return true;
```

```
  else
    --print("Loop "..loop:get_id().." isn't innermost and one-block
            ("..loop:get_nblocks().." blocks)");
    return false;
  end
end

local mil_out_path = io.popen("pwd"):read("*l").."/";

events = {
  run_dir = mil_out_path,
  functions_global_blacklist = {
    {subtype = "stringlist",value = {"call_gmon_start","_init","_fini",
                                     "_start","frame_dummy"}},
    {subtype = "regexplist",value = {"^_*irc_*.*","^__do_global_.*",
                "^__libc_csu_.*","^_*intel_.*","^__pthread_.*"}}
  },
  at_entry={
    {
      name = "instru_load",
      lib = "libinstru.so",
      params = { {type = "macro",value = "instru_launch_params"} }
    },
    {
      name = "instru_set_result_file",
      lib = "libinstru.so",
      params = { {type = "string",value = "my_div_baseline_cycles_rslt"} }
    }
  },
  main_bin = {
    properties={
      enable_function_instrumentation = true,
      enable_loop_instrumentation = true,
      --generate_metafile = true,
    },
    path = mil_out_path.."my_div_baseline",
    output_suffix = "_inst_cycles",
    functions={{
      loops = {{
filters = {{
    type = "user",
    filter = udf_isinner_oneblockloop
}},

entries={{
  name = "instru_loop_tstart",
  lib = "libinstru.so",
  params = {{type = "function",value = udp_get_loop_newunique_id} }
}},
```

```
exits={{
  name = "instru_loop_tstop",
  lib = "libinstru.so",
  params = {{type = "function",value = udp_get_loop_currunique_id} }
}},
      }}
    }}
  }
};
at_instru_exit = udf_exec_at_instru_exit;
```

The following MIL script is used to obtain the number of iterations of the loop. The script is actually quite the same. Instead of using $before$ and $after$ loop events, we use the *backedge* event.

```
mil.data.loop_meta_info = Table:new();
mil.data.instru_loop_counter = 0;

function udf_exec_at_instru_exit()
  mil.data.loop_meta_info:save_output("loop_meta_info.iters.lua",
                                      "loop_meta_info_iters");
end

function udp_get_loop_newunique_id(loop)
  local fct_name       = loop:get_function():get_name();
  local maqao_loop_id  = loop:get_id();
  local lfm            = mil.data.loop_meta_info;
  local new_instru_lid ;

  if(lfm[fct_name] == nil) then
    lfm[fct_name] = Table:new();
  end
  new_instru_lid = mil.data.instru_loop_counter;
  if(lfm[fct_name][new_instru_lid] == nil) then
    lfm[fct_name][new_instru_lid] = maqao_loop_id;
  end
  mil.data.instru_loop_counter = mil.data.instru_loop_counter + 1;

  return new_instru_lid;
end

function udp_get_loop_currunique_id(loop)
  return mil.data.instru_loop_counter - 1;
end

function udf_isinner_oneblockloop(loop,gvars)
  if(loop:is_innermost() and loop:get_nblocks() == 1) then
    --print("Loop "..loop:get_id().." is innermost and one-block");
    return true;
```

```
  else
    --print("Loop "..loop:get_id().." isn't innermost and one-block
                            ("..loop:get_nblocks().." blocks)");
    return false;
  end
end

local mil_out_path = io.popen("pwd"):read("*l").."/";

events = {
  run_dir = mil_out_path,
  functions_global_blacklist = {
    {subtype = "stringlist",value = {"call_gmon_start","_init","_fini",
                                              "_start","frame_dummy"}},
    {subtype = "regexplist",value = {"^_*irc_*.*","^__do_global_.*",
                        "^__libc_csu_.*","^_*intel_.*","^__pthread_.*"}}
  },
  at_entry={
    {
      name = "instru_load",
      lib = "libinstru.so",
      params = { {type = "macro",value = "instru_launch_params"} }
    },
    {
      name = "instru_set_result_file",
      lib = "libinstru.so",
      params = { {type = "string",value = "my_div_baseline_iters_rslt"} }
    }
  },
  main_bin = {
    properties={
      enable_function_instrumentation = true,
      enable_loop_instrumentation = true,
      --generate_metafile = true,
    },
    path = mil_out_path.."my_div_baseline",
    output_suffix = "_inst_iters",
    functions={{
      loops = {{
filters = {{
    type = "user",
    filter = udf_isinner_oneblockloop
}},
backedges={{
  name = "instru_loop_backedge_count",
  lib = "libinstru.so",
  params = {{type = "function",value = udp_get_loop_newunique_id} }
}},
      }}
```

```
    }}
  }
};
at_instru_exit = udf_exec_at_instru_exit;
```

This following MAQAO script prints the number of estimated cycles (by static analysis) of all innermost loops for a given binary, function and micro-architecture. In our case, it will print the results for the previously introduced loop.

```
dofile("loop_meta_info.iters.lua");
dofile("my_div_baseline_iters_rslt");
dofile("loop_meta_info.cycles.lua");
dofile("my_div_baseline_cycles_rslt");

--Table:new(loop_meta_info_iters):tostring();
--Table:new(my_div_baseline_iters_rslt):tostring();
--Table:new(loop_meta_info_cycles):tostring();
--Table:new(my_div_baseline_cycles_rslt):tostring();

local function print_usage ()
   print ("Usage: maqao print_estimated_cycles.lua
           uarch=<micro-architecture>
           bin=<path to binary> fct=<function name>");
end

-- Parses command line arguments
args = server:get_args (arg);

-- Checks command line arguments
if (args.uarch == nil or args.bin == nil) then
   print (args.uarch, args.bin, args.fct)
   print_usage ();
   os.exit (-1);
end

-- Gets and checks the micro-architecture
local uarch;
if     (args.uarch == "CORE2_65"    ) then
  uarch = server.consts.UARCH_CORE2_65
elseif (args.uarch == "CORE2_45"    ) then
  uarch = server.consts.UARCH_CORE2_45
elseif (args.uarch == "NEHALEM"     ) then
  uarch = server.consts.UARCH_NEHALEM
elseif (args.uarch == "SANDY_BRIDGE") then
  uarch = server.consts.UARCH_SANDY_BRIDGE
else
   print (args.uarch .. " is not a valid micro-architecture");
   print ("Valid micro architectures: " ..
         table.concat (avail_uarch_list, " "));
```

```
   os.exit (-1);
end

-- Creates an empty MAQAO project
local proj = project.new ("function_names");
if (proj == nil) then
   print ("Cannot create a MAQAO project named
          \"print_estimated_cycles\"");
   os.exit (-1);
end

-- Loads the binary
local bin = proj:load (args.bin, uarch);
if (bin == nil) then
   print ("Cannot load (disassemble, analyze DDG...)
           the binary "..args.bin);
   os.exit (-1);
end

local fcts = loop_meta_info_cycles

for f in bin:functions() do
   if (fcts[f:get_name()] ~= nil) then
      for l in f:innermost_loops () do
 local lid = l:get_id();
 local instru_lid = fcts[f:get_name()]:get_index_valueof(lid);

 if(fcts[f:get_name()][instru_lid] ~= nil) then
   local stan_results = stan:get_static_analysis_results (l);

   if (stan_results ["can be analyzed"]) then
      local nb_cycles = my_div_baseline_cycles_rslt["threads"][0]
                                ["loops"][instru_lid]["elasped_cycles"];
              local nb_iters = my_div_baseline_iters_rslt["threads"][0]
                                ["loops"][instru_lid]["iters"];
      local nb_instances = my_div_baseline_cycles_rslt["threads"][0]
                                ["loops"][instru_lid]["instances"];
      local measured = nb_cycles / nb_iters;

      print (string.format ("Loop #%d:", l:get_id()));
      print (string.format ("Avg iteration nb: %d",
                                    nb_iters / nb_instances));
      print (string.format ("Estimated: %.2f cycles per binary loop
                                    iteration", stan_results ["cycles L1"]));
      print (string.format ("Measured : %.2f cycles per binary loop
                                    iteration", measured));
   end
 else
   print("Loop "..lid.." was not executed");
```

```
 end
      end
   end
end
```

Figure B.1 presents the script that instruments the original binary containing the target loop. Each MIL script produces an instrumented binary. After that, the comparing script is executed. Results are shown in Figure B.2.

```
echo "###################################################"
echo "## Dynamic instrumentation for MAQAO STAN module ##"
echo "###################################################"

maqao module=mil input=mil_get_loop_iters.lua &> log
maqao module=mil input=mil_get_loop_cycles.lua &>> log
./my_div_baseline_inst_iters 100000 2000 &>> log
./my_div_baseline_inst_cycles 100000 2000 &>> log
maqao print_estimated_cycles.lua uarch=NEHALEM bin=my_div_baseline
```

Figure B.1: Shell script to execute the differents steps

```
###################################################
## Dynamic instrumentation for MAQAO STAN module ##
###################################################
Loop #1:
Avg iteration nb: 2000
Estimated: 14.00 cycles per binary loop iteration
Measured : 29.09 cycles per binary loop iteration
```

Figure B.2: MIL script to get instances and cycles information

Bibliography

[1] A. AB. Acumem SlowSpotter and Acumem ThreadSpotter, 2009. http://www.acumem.com/content/view/133/182/.

[2] 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. Technical Report TR08-06, Rice University, 2008. 42

[3] N. Aeronautics and S. Administration. Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html. 79, 88

[4] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization Made Easier with Intel Performance-Tuning Utility, 2007. http://www.intel.com/technology/itj/2007/v11i4/. 96

[5] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. *SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance*, volume 2104, pages 1–10. Springer-Verlag, 2001. 79

[6] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in openmp? In *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, WOMPAT'03, pages 147–159, Berlin, Heidelberg, 2003. Springer-Verlag. 116

[7] E. Ayguadé, M. Brorsson, H. Brunst, H. C. Hoppe, S. Karlsson, X. Martorell, W. E. Nagel, F. Schlimbach, G. Utrera, and M. Winkler. OpenMP Performance Analysis Approach in the INTONE Project. In *Workshop on OpenMP*, 2001. 72

[8] K. C. Barr and K. Asanovic. Branch trace compression for snapshot-based simulation. In *In International Symposium on Performance Analysis of Systems and Software*, 2006.

[9] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi. Performance tuning of x86 openmp codes with maqao. In M. S. Muller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113. Springer Berlin Heidelberg, 2010. 87

[10] Y. Ben-Asher and N. Rotem. Automatic memory partitioning: increasing memory parallelism via data structure partitioning. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 155–162, New York, NY, USA, 2010. ACM. 123

[11] T. O. A. R. Board. Openmp : The openmp api specification for parallel programming. http://openmp.org/wp/. 25

[12] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, pages 67–77, 2011. 2

[13] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[14] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Intl. Journal on High Performance Computing Applications*, 14:317–329, November 2000. 30, 40, 71, 72, 76, 89

[15] M. Burtscher. Tcgen 2.0: a tool to automatically generate lossless trace compressors. *SIGARCH Comput. Archit. News*, 34(3):1–8, June 2006. 98

[16] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *IEEE Trans. Comput.*, 54(11):1329–1344, Nov. 2005. 98

[17] 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 *ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. 47

[18] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[19] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Workshop on OpenMP applications and tools*, July 2001. 72

[20] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM. 2

[21] T. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Atlanta, May 1999. ACM Press.

[22] I. Corporation. IntelÂ® VTune Performance Analyzer 9.1 with IntelÂ® Thread Profiler, 2009. http://software.intel.com/en-us/intel-vtune/. 39, 96

[23] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, and R. Archambault. Mpdas: memory-pooling-assisted data splitting. In *ISMM*, pages 101—110, Tucson, Arizona, June 2008. ACM Press. 123

[24] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46—55, 1998. 95

[25] A. M. Dani, K. Varadarajan, B. Amrutur, and Y. N. Srikant. Accelerating multi-core simulators. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2377–2382, New York, NY, USA, 2010. ACM Press.

[26] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. Sigma: a simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 99

[27] L. DeRose, T. Hoover, and J. Hollingsworth. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. In *IEEE Intl. Parallel and Distributed Processing Symposium*, 2001. 30, 72, 78

[28] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. Exploring Application Performance: a New Tool For a Static/-Dynamic Approach. In *Los Alamos Computer Science Institute Symp.*, Santa Fe, NM, Oct. 2005.

[29] D.L.Bruening. *Efficient,transparent,and comprehensive runtimecode manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

[30] J. J. Dongarra, S. W. Otto, and M. Snir. An introduction to the mpi standard. Technical report, 1995. 26

[31] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM. 2

[32] H. Dybdahl and P. Stenstrom. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE*, pages 2–12. IEEE Computer Society, 2007. 2

[33] D. Eklov, D. Black-Schaffer, and E. Hagersten. Statcc: a statistical cache contention model. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 551–552, New York, NY, USA, 2010. ACM. 27

[34] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *in Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2010*, 2010. 27

[35] E. N. Elnozahy. Address trace compression through loop detection and reduction. *SIGMETRICS Perform. Eval. Rev.*, 27(1):214–215, 1999. 99

[36] Fujitsu. White paper fujitsu primergy servers memory performance of xeon e7-8800/4800/2800 (westmere-ex) based systems. http://globalsp.ts.fujitsu.com/dmsp/Publications/public/wp-westmere-ex-memory-performance-ww-en.pdf. 11

[37] R. Harkness. Experiences with enzo on the intel many integrated core (intel mic) architecture. http://www.tacc.utexas.edu/documents/13601/ac58a8be-19f8-42a1-afb5-2e345875a782. 2

[38] J. K. Hollingsworth, O. Niam, B. P. Miller, Z. Xu, M. J. R. Goncalves, and L. Zheng. Mdl: A language and compiler for dynamic program instrumentation. *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, 1525:201–212, 1997. 71, 73

[39] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao - an extensible micro-architectural optimizer. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 1–10. IEEE Computer Society, 2011.

[40] T. K. Institute. Open|speedsop. http://www.openspeedshop.org. 42

[41] Intel. Architecture code analyzer. http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/. 27

[42] Intel. Architecture code analyzer user manual. http://software.intel.com/file/43956.

[43] Intel. Array building blocks : Sophisticated library for vector parallelism. http://software.intel.com/en-us/articles/intel-array-building-blocks/. 26

[44] Intel. Cilk plus. http://software.intel.com/en-us/articles/intel-cilk-plus/. 25

[45] Intel. Compilers with integrated profilers. http://software.intel.com/en-us/articles/intel-parallel-studio-home/.

[46] Intel. Concurrent collections for c++. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/. 26

[47] Intel. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. http://download.intel.com/embedded/software/IA/324264.pdf. 92

[48] Intel. Intel 64 and ia32 optimization reference manual. 26, 51, 107

[49] Intel. Threading building blocks. http://threadingbuildingblocks.org/. 25

[50] Intel. Single-chip cloud computing, 2010. http://techresearch.intel.com/articles/Tera-Scale/1826.htm. 2

[51] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *Proc. Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, Beijing, China, 2008. ACM. 38, 96, 97

[52] Y. Jiang, E. Zhang, X. Shen, Y. Gao, and R. Archambault. Array regrouping on cmp with non-uniform cache sharing. In *Intl. Workshop on Languages and Compilers for Parallel Computing*, 2010.

[53] E. E. Johnson. Pdats ii: Improved compression of address traces, 1999. 98

[54] E. E. Johnson and J. Ha. Pdats: Lossless address space compression for reducing file size and access time. 1994. 98

[55] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved. In *Proceedings of the First International Symposium on Memory Management, ACM.* Press, 1998. 26

[56] A. joint project of LBNL and U. Berkeley. Berkeley upc - unified parallel c. http://upc.lbl.gov/. 26

[57] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *In International Parallel & Distributed Processing Symposium (IPDPS)*, 2010. 2

[58] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhnag. Cache topology aware computation mapping for multicores. *SIGPLAN Not.*, 45(6):74–85, 2010. 97

[59] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 94–103, New York, NY, USA, 2008. ACM Press. 97, 100

[60] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. M. D. Quang, and W. Jalby. A balanced approach to application performance tuning. In *Intl. Workshop on Languages and Compilers for Parallel Computing*, Oct. 2009. 57

[61] R. Kufrin. Perfsuite: An accessible, open source performance analysis environment for linux. In *In Proc. of the Linux Cluster Conference, Chapel*, 2005.

[62] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM. 98

[63] J. R. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1995.

[64] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Chigago, Illinois, June 2005. 123

[65] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, pages 175–183, 2010. 71, 72

[66] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. *SIGARCH Comput. Archit. News*, 38(3):270–279, 2010. 97

[67] J. Lin and P.-C. Yew. A compiler framework for general memory layout optimizations targeting structures. In *Intl. Workshop on Iteraction between Compilers and Computer Architectures*, Pittsburg, Mar. 2010. ACM Press. 123

[68] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Redd, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005. 71

[69] MAQAO. Modular assembly quality analyser and optimizer. http://www.maqao.org. 3, 71, 75, 77, 78, 94, 97, 101

[70] J. Marathe and F. Mueller. Source-code-correlated cache coherence characterization of openmp benchmarks. *IEEE Trans. on Parallel and Distributed Systems*, 18(6):818–834, 2007. 96, 97

[71] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting. *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, 0:289, 2003. 96, 99

[72] A. Mazouz, S.-A.-A. Touati, and D. Barthou. Study of variations of native program execution times on multi-core architectures. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '10, pages 919–924, Washington, DC, USA, 2010. IEEE Computer Society. 26

[73] S. L. G. P. B. K. M. K. McKusick. gprof: a call graph execution profiler. http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf. 31, 34

[74] A. Milenkovic and M. Milenkovic. Stream-based trace compression. *IEEE Comput. Archit. Lett.*, 1(1):9–12, Jan. 2002. 99

[75] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. Bruce, I. Karen, L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 1995. 30

[76] B. Mohr, A. D. Malony, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for openmp. In *Workshop on OpenMP*, 2002.

[77] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Towards a performance tool interface for openmp: An approach based on directive rewriting. In *Workshop on OpenMP*, 2001. 31, 72, 78, 89

[78] T. Moseley, D. Grunwald, and R. Peri. Seekable compressed traces. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, IISWC '07, pages 129–138, Washington, DC, USA, 2007. IEEE Computer Society. 99

[79] 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. 32

[80] J. Mußler, D. Lorenz, and F. Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Euro-Par Conference*, volume 6852 of *Lect. Notes in Computer Science*, pages 65–76. Springer-Verlag, Sept. 2011. 71, 73, 77

[81] S. Nanda, W. Li, L. chung Lam, , and T. cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, Mar. 2006.

[82] N. Nethercote. Cachegrind. http://valgrind.org/info/tools.html. 27, 96

[83] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007. ACM Press. 27, 31, 71, 96

[84] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artif. Int. Res.*, 7(1):67–82, Sept. 1997. 98

[85] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. 25

[86] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998. 26

[87] D. of Computer Science University of Illinois. Charm++. http://charm.cs.uiuc.edu/. 26

[88] P. C. U. of Rio de Janeiro in Brazil. Lua is a powerful, fast, lightweight, embeddable scripting language. http://www.lua.org. 60, 74

[89] C. O'Hanlon. A conversation with john hennessy and david patterson. *Queue*, 4(10):14–22, Dec. 2006. 2

[90] M. Pall. Luajit. http://luajit.org. 76

[91] S. Potluri, K. Tomkoy, D. Bureddy, and D. K. Panda. Intra-mic mpi communication using mvapich2: Early experience. http://www.tacc.utexas.edu/documents/13601/7f745047-5b63-44ac-aa7b-fb32cf0c4c05. 2

[92] G. Project. gprof. http://sourceware.org/binutils/docs/gprof/. 28, 31, 34

[93] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society. 2

[94] E. Raman, R. Hundt, and S. Mannarswamy. Structure layout optimization for multithreaded programs. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation.* IEEE Computer Society, 2007.

[95] A. Rane, J. Browne, and L. Koesterke. Perfexpert and macpo: Which code segments should (not) be ported to mic. http://www.tacc.utexas.edu/documents/13601/b9cc6eaa-d79e-4d78-80ef-cf1bec05085a. 2

[96] J. Rattner. On the decidability of phase ordering problem in optimizing compilation. In *The 14th International Symposium on High-Performance Computer Architecture*, HPCA '08. IEEE, 2008. 2

[97] S. P. Reiss and M. Renieris. Languages for dynamic instrumentation. In *Workshop on Dynamic Analysis*, pages 6–9, 2003.

[98] W. Richter, E. Taralova, and K. Naden. libperf. https://github.com/theonewolf/libperf. 86

[99] B. Risio, A. Berreth, S. Zuckerman, S. Koliai, M. Ivascot, W. Jalby, B. Krammer, B. Mohr, and T. William. How to Accelerate an Application: a Practical Case Study in Combustion Modelling. In *Proc. of ParCo*, Lyon, France, 2009. 110

[100] L. D. Rose, B. Mohr, and S. R. Seelam. Profiling and tracing openmp applications with pomp based monitoring libraries. In *Euro-Par Conference*, Lect. Notes in Computer Science, pages 39–46. Springer-Verlag, 2004. 72

[101] A. D. Samples. Mache: No-loss tract compaction. Technical report, Berkeley, CA, USA, 1988. 98

[102] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, Apr. 2008. 72

[103] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby. Qmc==chem : Quantum monte carlo for large chemical systems: Implementing efficient strategies for petascale platforms and beyond. Preprint under finalization, 2012. 90

[104] C. A. Schaefer, V. Pankratius, and W. F. Tichy. Atune-IL: An instrumentation language for auto-tuning parallel applications. In *Euro-Par Conference*, Lect. Notes in Computer Science, pages 9–20. Springer-Verlag, 2009. 71, 73

[105] K. W. Schulzy, R. Ulerich, N. Malaya, P. T. Bauman, R. Stogner, and C. Simmons. Early experiences porting scientic applications to the many integrated core (mic) platform. http://www.tacc.utexas.edu/documents/13601/aa8e98bc-3544-4136-81aa-3920ae882a65. 2

[106] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[107] SGI. Sgi numalink interconnect fabric. http://www.sgi.com/products/servers/numalink.html. 11

[108] SGI. Sgi uv 1000. http://www.sgi.com/products/servers/uv/models.html. 11

[109] S. S. Shende and A. D. Malony. The tau parallel performance system. *Intl. Journal on High Performance Computing Applications*, 20:287–331, 2006. ix, 41, 42, 84

[110] S. SMP. Verdatile smp : Server agregation. http://www.scalemp.com. 12

[111] S. Srinivasan, L. Zhao, B. Ganesh, B. Jacob, M. Espig, and R. Iyer. Cmp memory modeling: How much does accuracy matter?

[112] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. Hpctoolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 125(1):012088, 2008.

[113] J. Tao. Comprehensive cache performance tuning with a toolset. *Future Generation Computer Systems*, 26(1):167 – 174, 2010. 96

[114] S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM. 24

[115] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010. 33

[116] T. A. University. Stapl : Standard template adaptive parallel library. https://parasol.tamu.edu/stapl/. 26

[117] C. Valensi and D. Barthou. MADRAS: Multi-Architecture Disassembler and Reassembler, 2009. http://maqao.prism.uvsq.fr/wiki/wiki/MadrasDownload. 100

[118] S. Wallace and K. Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 209–217, San Jose, CA, March 2007. 30, 97

[119] J. Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 93–113. Springer Berlin Heidelberg, 2008. 10.1007/978-3-540-68564-7_7. 37

[120] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *In Proceedings of International Conference on Computational Science*, pages 440–447. Springer, 2004. 37

[121] J. Westbrook. Randomized algorithms for multiprocessor page migration. In *SIAM Journal on Computing*, pages 135–149. 26

[122] B. Ylvisaker and S. Hauck. Probabilistic auto-tuning for architectures with complex constraints. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 22–33, New York, NY, USA, 2011. ACM. 2

[123] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. Forma: A framework for safe automatic array reshaping. *ACM Trans. on Programming Languages and Systems*, 30(2), 2007. 121, 123

[124] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Washington DC, June 2004. ACM Press. 123

[125] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Trans. on Programming Languages and Systems*, 31(6), 2009. 123