université PARIS-SACLAY

NNT : 2019SACLV037

Thèse de doctorat

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Code optimization based on source to source transformations using profile guided metrics

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université de Versailles-Saint-Quentin-en-Yvelines

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: "Programmation : modèles, algorithmes, langages, architecture"

Thèse présentée et soutenue à Versailles, le 3 Juillet 2019, par

YOUENN LEBRAS

Composition du Jury :

Anthony Scemama IR CNRS, HdR, Université de Toulouse	Rapporteur
Angelo Steffenel) McF HdR, Université de Reims Champagne-Ardenne	Rapporteur
Michel Masella CEA DRF, HdR, Saclay	Examinateur
Denis Barthou PR. Université de Bordeaux	Président
Sophie Robert McF, Université d'Orleans	Examinateur
William Jalby PR, Univesité de Versailles Saint-Quentin	Directeur de thèse
Andres S. Charif-Rubial PhD., PeXL	Co-encadrant
Romain Dolbeau PhD ATOS BULL	Invité

Declaration of Authorship

I, Youenn LEBRAS, declare that this thesis titled, “Code optimization based on source to source transformations using profile guided metrics” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Youenn LEBRAS

Date: 07/03/2019

UVSQ

Abstract

Paris-Saclay
UVSQ - Li-Parad

PhD. Student in Computer Science

Code optimization based on source to source transformations using profile guided metrics

by Youenn Lebras

Modern high performance processor architectures tackle performance issues by heavily relying on increased vector lengths and advanced memory hierarchies to deliver high performance. Manual optimization of production HPC codes became a difficult task when having to manage multiple architecture dependent transformations. Developers usually trust compilers to automatically address these performance issues, but they deploy static performance models and heuristics which, must remain conservative or even fail in the worst case. Compiler optimization choices can be refines by using feedback data from dynamic profiling. But, it does not always consider some metrics and is rarely aggressive enough regarding metric data collection. On the other hand, performance analysis tools are pretty good at detecting specific performance issues, but only return observations on the quality and on the execution of the code. Our goal is to develop a framework will allow to perform of source code transformations based on performance analysis tools metrics. This framework will be integrated into the MAQAO tool suite. We present an FDO tool with a set of source-to-source transformations guided by metrics coming from the various MAQAO tools and open to user advices. This framework can also be used to simplify the development by automatically performing some simple, but time-consuming and error-prone transformations (e.g. loop/function specialization).

Acknowledgements

Avant toute chose, je voudrais remercier M. William Jalby, pour m'avoir acceuilli au sein du laboratoire et pour m'avoir suivi, orienté et soutenu durant cette thèse. Je voudrais aussi remercier Andrès S. Charif-Rubial pour son encadrement.

Je suis reconnaissant envers mes rapporteurs, Anthony Scemama et Angelo Steffenel pour avoir eu la patience de lire ce manuscrit et m'avoir fait des retours pour l'améliorer et sur mes travaux.

Mes années de thèses n'ont pas été de tout repos, mais grâce aux membres de ce laboratoire, elles m'ont paru plus facile à vivre. C'est pourquoi j'adresse un grand merci chacun d'entre eux, pour m'avoir aidé et soutenu que ce soit au travail ou en dehors via les team building au Montbauront et autres occasions. Cédric, pour avoir le courage d'agréger nos MPR pas toujours palpitant chaque mois, pour les soirées graphes avant les reviews et pour les tournées payées. Emmanuel, pour ces longues discussions durant nos trajets pour Teratec. Hugo, mon coéquipier breton de l'équipe, pour nos discussions au montbauront. Jasper, pour tes médecines douces et techniques de relaxation. Mathieu, pour avoir eu la patience (oui, la patience) de m'avoir eu en voisin de bureau et de m'aider à régler de nombreux problèmes d'interface chaise-clavier (que ce soit avec maqao ou pas) ainsi que pour tes citations toujours très inspirées et ton humour noir. A tous, un grand merci !

Je remercie bien évidemment tous les autres membres du laboratoire avec qui j'ai passé moins de temps, mais avec qui j'ai tout autant apprécié travailler/discuter/etc. Kevin, notre stagiaire qui un jour écrira son rapport ; Pablo pour m'avoir aidé dans de nombreuses démarches et avec qui j'ai apprécié travailler et donner des cours ; Yohan, doctorant et compagnon de galère de thèse (adum et formations), Nathalie et sylvain sans qui mon parcours n'aurait peut-être pas été le même, Clément, Marie, Sebastien, pour les différentes discussions et moments qui ont fait que ces années de thèse sont passées si vite.

Evidemment, merci à mes parents et ma soeur, pour leur soutien et pour leur patience lorsqu'ils tentaient de comprendre quel était exactement mon sujet de thèse et en quoi consistait mes travaux.

Un merci plus personnel et tout particulier à Chloé qui m'a soutenue autant (et aussi longtemps) que possible durant cette thèse et ce malgré ce qu'elle vivait. J'espère t'avoir autant soutenu et rendu le bonheur que tu as pu m'apporter. Pour tout ça, je lui dédicace cette thèse.

Un autre remerciement personnel, à mes amis qui sont là pour moi depuis de nombreuses années, Romain et MP, Julien, Jaunathan, Gaetan et Pierre (surtout pour les bières après le travail), Xavier, Jeremy, Zac, Anthony, Cassandre, etc. Il me faudrait encore quelques pages pour citer tout le monde donc je m'arrête là, mais vous pouvez toujours mettre votre nom dans la partie laissée en blanc qui suit.
Merci, !

And finally, after all these thanks, I would like to thank you random citizen! Yes you, who gonna read this thesis (or at least a brief part). Thank you for thinking that there is something interesting in this manuscript.

Contents

List of Figures

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
AVX	**A**dvanced **V**ector **E**xtensions
DDR	**D**ouble **D**ata **R**ate
FDO	**F**eedback **D**ata **O**ptimization
GCC	**G**NU **C**ompiler **C**ollection
GPL	**G**eneral **P**ublic **L**icense
GWP	**G**oogle-**W**ide **P**rofiler
HBM	**H**igh **B**andwidth **M**emory
HPC	**H**igh **P**performance **C**omputation
IR	**I**ntermediate **R**epresentation
LCT	**L**oop **C**ount **T**ransformation
LGPL	**L**esser **G**eneral **P**ublic **L**icense
LLVM	**L**ow **L**evel **V**irtual **M**achine
MAQAO	**M**odular **A**ssembly **Q**uality **A**nalyzer and **O**ptimizer
MSR	**M**odel **S**pecific **R**egister
PGO	**P**rofile **G**uided **O**ptimization
RAT	**R**egister **A**alias **T**able
ROB	**R**e-**O**rdering **B**uffer
SIMD	**S**ingle **I**nstruction on **M**multiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
SVT	**S**hort **V**ectorization **T**ransformation

À Chloé.

Chapter 1

Background

1.1 Evolution of HPC Processor Architectures

This section presents the evolution of High Performance Computing (HPC) processors architecture. To follow this evolution we focus on four micro-architectures of the HPC leader, Intel. In 2007 Intel adopted the "tick-tock" model where a new micro-architecture ("tock") is followed by a die shrink ("tick") and, sometimes, new instructions or features are introduced.

For consistency and stability, this survey only focuses on new micro-architectures designated during the "tock" cycles. First, we present Nehalem, an Intel micro-architecture released in 2008 and successor of the Intel Core 2.

Nehalem-based processors use a 45nm engraving process and allow hyper-threading. This micro-architecture comes with an L2 cache smaller than its predecessors but it embeds a very large L3 cache shared among all cores. The Nehalem "tick" is named Westmere and precedes the Sandy Bridge micro-architecture.

Sandy Bridge, was released in 2011 as the "second-generation core" and was considered as the successor of Nehalem. Sandy Bridge uses a 32nm process and offers a new set of vector instructions (AVX) while retaining most of Nehalem core features. The Sandy Bridge "tick" is named Ivy Bridge and is a 22nm die shrink of the original. The successor of Sandy Bridge is named Haswell.

The Haswell micro-architecture was released in 2013 as the "fourth-generation core", it uses a 22nm process and was specifically designed for power optimization. This micro-architecture was deployed in a wide range of low-power processors for ultrabook computers and saw an upgrade in the vector instruction set (AVX2). Intel also increased the number of some registers and enlarged multiple memory buffer sizes. The "tick" of this micro-architecture is labeled Broadwell and is made using a 14nm process.

Finally, Skylake will be the last micro-architecture we present. It was released in 2015 as the "sixth-generation core" and uses the same 14nm manufacturing process as Broadwell. With Skylake, Intel gave up the "tick-tock" model and offered several alternative versions

System	Year	# Cores	Frequency (Ghz)	Turbo (Ghz)	Lithography Process (nm)
Nehalem	2008	1 - 8	1.86 - 2.53	1.86 - 3.33	45
Sandy Bridge	2011	2 - 8	1.8 - 3.5	1.8 - 4.0	32
haswell	2013	2 - 18	1.9 - 4.0	2.7 - 4.4	22
Skylake	2015	4 - 56	2.0 - 3.6	2.3 - 3.8	14

Table 1.1: Release year, range of number of cores and range of frequencies with and without Turbo Boost for each micro-architecture. Source www.wikipedia.org (only server information has been selected), https://en.wikipedia.org/wiki/Transistor_count.

Processor Name	# Cores	Year	Transistor Count	Area (mm^2)	L3 size (MiB)	Transistor count Normalized	Area (mm^2) Normalized	L3 size (MiB) Normalized
Xeon Nehalem-EX	8	2010	2.3×10^9	684	18 - 24	0.288×10^9	85.5	2.25 - 3
Core i7 Sandy Bridge-E	6	2011	2.27×10^9	434	12 - 15	0.378×10^9	72.33	2 - 2.25
Core i7 Haswell-E	8	2014	2.6×10^9	355	15 - 20	0.325×10^9	44.5	1.9 - 2.5
Core i7 Skylake K	4	2015	1.75×10^9	122	6 - 15	0.437×10^9	30.5	1.5 - 3.75

Table 1.2: Number of transistors per processor and their area with normalization per 1 core. Source https://en.wikipedia.org/wiki/Transistor_count.

instead: Kaby Lake, Coffee Lake, Cannon Lake and Cascade Lake. Skylake introduces larger vector registers and a new set of vector instructions (AVX512), a deeper out-of-order buffer, more execution units, more load/store bandwidth as well as improvements to Hyper-Threading technology.

The Intel Xeon Phi (MIC: KNF, KNC, KNL) family of processors will not be covered in this work because of its demonstrated low performance, complex requirements and its lack of a developer friendly environment.

1.1.1 Uni-core Design Details

Over the last decade, CPU architectures evolved dramatically in terms of performance; from the Pentium family to the last generation of Intel chips (Skylake-X), it is obvious that the

performance model of the market shifted from high frequency single core processors to multi-tasking high-core-count (or Manycore) parallel architectures. This allowed CPU manufacturers to lower the power consumption (at the cost of latency sometimes) and increase memory bandwidth and instruction throughput. This shift of performance model introduced additional optimization challenges related to parallelism (task and data), compilation and code generation. In such environment, the optimization process is key to maintain a reasonable performance level on modern micro-processor architectures [35]. Jalby et al.[52] presented the progress of performance correlated to the Intel architecture evolution for the last twenty years; and especially how the top 500 most powerful systems struggle to reach peak computational performance on real world applications even with a continuous increase in flops. This evolution brings to light new optimization issues and challenges (i.e. managing larger vector lengths, automatic vectorization, ...) beyond previously encountered performance limiting factors (i.e. pipeline stages, memory hierarchy, ...).

Table 1.1 and 1.2 show the evolution of the technology using the core count, the frequency and the lithography process as performance markers for the presented micro-architectures. CPU performance is not only centered around core count and frequency but can also be affected by the memory hierarchy (cache levels, number of memory channels, ...) and its ability to process instructions and data in fewer cycles, or manipulate more data at once (vectorization). The last two axes are of utmost importance for a micro-architecture to deliver considerable performance. Table 1.3 shows Intel's endeavors to deal with both these axes with micro-architectural upgrades.

"Out-of-Order" execution is an important improvement in micro-architectures design. Basically, on "in-order" processors, if one or more operands are unavailable during the current clock cycle, the processor stalls until they become available. The Out-of-Order process avoids this stall by processing instructions based on their readiness rather than order; each instruction with a ready operand is added into a buffer and dispatched to free units; instruction 2 can be executed before instruction 1 has been completed if all operands to perform instruction 2 are completed. Once the execution finishes, the processor ensures that all instructions are qualified "in-order" to preserve the correct program semantic. This process has been used in microprocessors since 1990; the concept does not change but the size of all buffers or registers has significantly increased, as we can see in table 1.3. The "Out-of-Order" buffer has practically doubled in size from Nehalem to Skylake. The same evolution applies to Stores, Loads and the scheduler entries. The largest increase was for the number of instructions decoding queue which triples with Skylake. The number of execution ports has not increased a lot but ports are now allowed to perform more different kinds of operations; from Haswell, Intel added specialized ports which allow Fused-Multiply-Add (FMA) instructions, Advanced Encryption Standard, Carry-Less Multiplication, ... to be performed.

	Nehalem	Sandy Bridge	Haswell	Skylake
Out-of-Order buffer	128	168	192	224
ROB entries	128	168	192	224
In-flight Stores	32	36	42	56
In-flight Loads	48	64	72	72
Scheduler Entries	36	54	60	97
Integer Register File	0	144	168	168
FP Register File	0	144	168	168
Line fill buffer	10 entries	10 entries	10 entries	10 entries
Instruction Decode Queue	28/thread	28/thread	56/thread	64/thread
# Execution Ports	6	6	8	8
# ALU Ports	3	3	4	4
# Read Ports	1	2	2	3
# Write Ports	1	1	2	2
# Address Calculation	1	2	3	3
# FMA Ports	0	0	2	3
# Specialized Port	0	0	3	4

Table 1.3: Evolution of key parameters on recent architectures. Specialized ports gather: AES encryption, vector permutation, SADBW, PCLMUL, jump and branch, branch. Source: www.anandtech.com & www.agner.org/optimize/.

An FMA is a floating-point multiply-add operation performed in one step."That is, where an unfused multiply-add would compute the product $b * c$, round it to N significant bits, add the result to a, and round back to N significant bits, a fused multiply-add would compute the entire expression $a + b * c$ to its full precision before rounding the final result down to N significant bits"[1]. The set of FMA instructions allow faster and more accurate specialized operations and are closely related to vector instruction sets described in the next section.

Modern high-performance processor architectures heavily rely on increased vector lengths and advanced memory hierarchies to deliver high-performance. This stresses the importance of data access optimization and efficient usage of the underlying available vector capabilities detailed in section 1.2.1 and 1.2.2.

1.1.2 Memory Technology

Until recently, the memory was divided into two large families: first, persistent memory (e.g. Hardrive, flash drive); and second, volatile memory (i.e. DRAM, SRAM). For a myriad of reasons, memory technology hasn't evolved as fast and as much as CPU technology. But, the computer bus and interfaces have seen massive improvement in bandwidth and transfer speed over the years. In 2000, the first Double Data Rate (DDR) memory based on the synchronous dynamic random-access memory (SDRAM or DDR1), was released. This technology has better bandwidth transfer rates; the interface allows to transfer data on both the rising and the falling edges of the clock signal and thus doubles the transfer rate. Figure 1.1 presents the evolution of key parameters of the different DDR.

Memory Bus and Interfaces The main factor behind this evolution is bus and interfaces improvement, as we can see on table 1.4 and 1.6. The first DDR was clocked at 100Mhz with an IO bus clock between 100 and 200 Mhz. The DDR2 bus came up even faster and allowed to operate external data twice as fast as its predecessor. All other key parameters have also been improved and even doubled. Prefetcher buffer width moved from 2 bits in DDR1 to 4 bits in DDR2; both the I/O bus clock and the memory clock frequencies were increased to provide a better transfer rate. Compared to DDR2, DDR3 reduces power consumption by 40% by using a lower voltage, 1.5 volts against 1.8 for DDR2 and 2.5 for DDR1. DDR3 introduced features allowing to control the refresh rate according to the Automatic Self Refresh and the Self-Refresh Temperature variation. DDR4, which is supported by Skylake, has a reduced operating voltage of 1.2 volts with an increased transfer rate of 3.2 GT/s. Only

[1]Source: https://en.wikipedia.org/wiki/Multiply-accumulate_operation

	DDR1	DDR2	DDR3	DDR4
Year	2000	2003	2007	2014
Memory clock (Mhz)	100	200	200	400
I/O bus clock (Mhz)	100 - 200	200 - 533	400 - 1067	800 - 1600
Transfer rate (Gigatransfers/s)	0.2	0.8	1.6	3.2
Cycle time (ns)	5 - 6	1.8 - 5	1.5 - 0.6	1.2 - 0.4
Prefetcher buffer width	2 bits	4 bits	8 bits	8 bits
Energy consumption (Volts)	2.5	1.8	1.5	1.2

Table 1.4: Double Data Rate key parameters. "Gigatransfers per second refer to the number of operations transferring data that occur in each second". Cycle Time represent time between two clock cycles in nanosecond. Source: https://en.wikipedia.org/wiki/DDR_SDRAM, https://www.memoireonline.com/01/12/5117/m_volution-sur-la-memoire-vive7.html, https://en.wikipedia.org/wiki/Double_data_rate & https://www.transcend-info.com/Support/FAQ-296

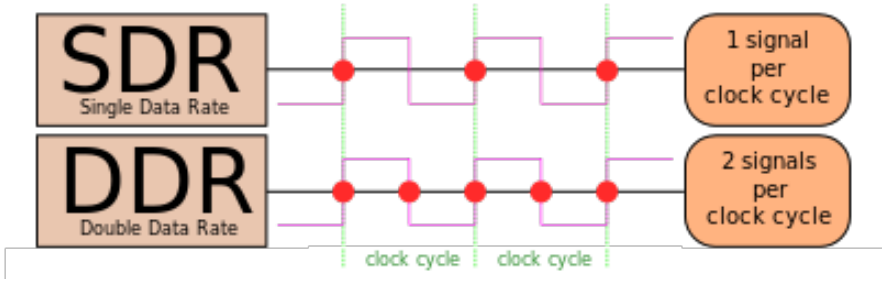


Figure 1.1: DDR1 allows to transfer data on both, the rising and falling edges of the clock signal. Source https://en.wikipedia.org/wiki/Double_data_rate

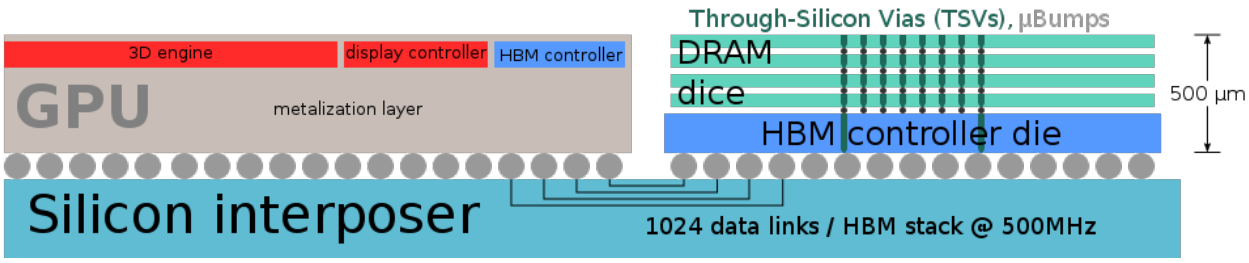


Figure 1.2: Sub-part of a cut through a graphics card that uses High Bandwidth Memory. Grey dots represent: PCI express / Electrical current / Display connectors. Source: https://en.wikipedia.org/wiki/High_Bandwidth_Memory

the prefetcher buffer width remained unchanged from DDR3. DDR4 introduces features to improve the stability of data transmission and memory signal integrity. DDR5, planned for 2020, is intended to reduce power consumption while doubling the capacity of the current DDR4.

Other technologies have also been developed; for example, High Bandwidth Memory (HBM), mainly used in GPUs, is designed to stack silicon dies and get a 3D structure for DRAM (2011), as well as a high-performance RAM interface for a 3D-stacked DRAM (2013). Figure 1.2 presents a use of the HBM in a GPU. The HBM memory bus is wider than DRAM bus (i.e. DDR4), it can support up to 4GB per package and has two 128-bits channels per die, for an HBM stack of four DRAM dies, for example. The second generation of HBM, released in 2016, allows up to 8 GB per stack and is able to reach 256 GB/s memory bandwidth per package. A third generation is planned for 2020, and even a fourth between 2022 and 2024, with the objective of reaching the Exascale mark.

Even with improvements to the memory, on Skylake mainly, it remains a complex component with many strong dependencies between caches. Intel has recently introduced the Optane technology, a new class of configurable memory that can function in both, persistent and volatile modes. This adds a new level of complexity above the already complex hierarchy. Intel Optane memory is a smart memory technology that accelerates responsiveness[2] and is named 3D XPoint due to its functioning system; the bit storage is based on a change of bulk resistance, in conjunction with a stackable cross-grided data access array to further boost density[3]. With its new 3D structure with perpendicular wires connecting submicroscopic columns, individual memory cells can be addressed by selecting its top and bottom wire[3].

1.2 With Great Evolution Comes Great Difficulties

Modern high-performance processor architectures tackle performance issues by heavily relying on increased vector lengths and advanced memory hierarchies in order to deliver high-performance. This stresses the importance of data access optimization and efficient usage of the underlying hardware.

1.2.1 Vectorization Evolution

Vector instructions consists of applying the same operation on packed data. Specialized instructions are used for performing vector operations, referred to as SIMD (Single Instruction Multiple Data). Data is loaded in a vector register, and SIMD instructions are used before storing the results in a vector register. In figure 1.3, the scalar operation performs four additions serially, while the SIMD instruction performs only one addition on the four elements.

On Intel microprocessors, the SIMD set of instructions has evolved from what Intel labeled MMX, (extensions to the x86 architecture added in 1996 with Pentium) to AVX-512 (Advanced Vector Extensions), which expands the vector length to 512-bit. AVX-512 was first supported by Intel's Knights Landing processor.

Lately, vectorization has reemerged as a key element for performance. The continuous increase in vector length creates more opportunities to boost the performance of HPC applications. Vectorization is key in the optimization process allowing high-performance and

[2]Source: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html

[3]Source: https://www.anandtech.com/

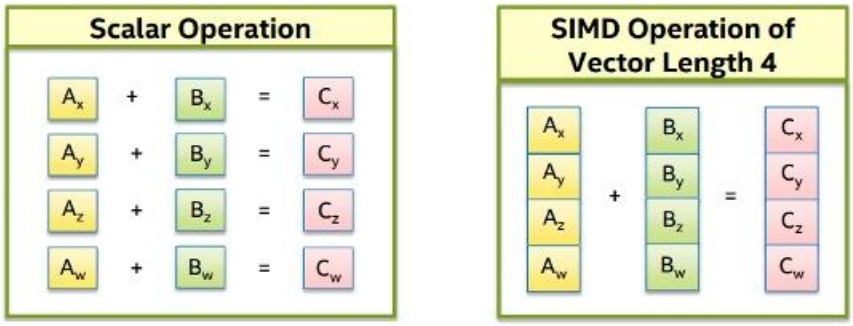


Figure 1.3: Difference of execution between a sequential addition and a vector operation of addition with vectors of 4 elements. Source : https://www.slideshare.net/IntelSoftware/the-next-leap-in-javascript-performance

Name	Vector size	Architecture
MMX	64b	Pentium 2 (1997)
SSE	128b	Pentium 3 (1999)
SSE2	128b	Pentium 4 (2000)
SSE3	128b	Pentium 4 (2004)
SSSE3	128b	Merom (2006)
SSE4.1	128b	Penryn (2007)
SSE4.2	128b	Nehalem (2008)
AVX	256b	Sandy Bridge (2011)
AVX2	256b	Haswell (2013)
AVX512	512b	XeonPhi (2013) & Skylake-X (2017)

Table 1.5: Evolution of Intel Vector instruction set.

tipping the balance between memory accesses and computation.

However, the advantages are not obvious given that the task of vectorizing and choosing the most appropriate/efficient instructions is left to compilers. Moreover, there's great difficulty in determining whether a loop will be more efficient in sequential or in vectorized form. The compiler uses a static cost model to guide its choice but, without knowledge of loops bounds, it cannot make an optimal choice every time. On Intel architectures, vector instructions are very sensitive to memory alignment and thus exhibit different performances depending on whether allocated memory is aligned to a power of two (or cache line) boundary or not. Although aligned accesses are more efficient, in most cases, it is extremely difficult to generate SIMD code for aligned accesses.

1.2.2 Memory Organization

Memory hierarchy has evolved slowly during the last decade, even after the introduction of the memory wall [121] problem. Table 1.6 presents the evolution of different caches, the translation lookaside buffer (TLB) and latency from Nehalem to latest Skylakes. The low latency values in table 1.6 are explained by the fact that the last level cache is sliced into multiple L3 connected caches. This configuration affects latency but improves the bandwidth. This table shows that the cache and TLB have not evolved much before Skylake which comes with upgraded L2 and L3 caches. With the previous generations of Intel microprocessors, each core had an inclusive private L1 and L2 caches and the last level cache was also inclusive and covered all cores through a bi-directional interconnect. With Skylake-X™, the L3 cache (last level cache) becomes non-inclusive.

Figure 1.4 presents the two possible use cases of the new Intel Optane technology, which can be used either as a second storage or cache space like an SSD drive, or as extended memory where it will be complementary to the RAM. As shown on figure 1.4 it can be used as complements to the RAM rather than as a complete replacement of it, thus creating a biggest memory pool and a new level in the hierarchy.

Caches Hierarchy As mentioned previously, cache hierarchies have seen little advancements and the improvement of Skylake with larger buffers and increased capacity is not an evolution. However, the new hard drive technology is a revolution in the domain of memory and the Intel Optane technology brings a new approach and is complementary to both RAM and disk level I/O. This technology is too recent and hasn't shown its potential on current HPC workloads. Also, it is unclear what kind of challenges the HPC developers will have to face when this technology becomes more available. What must be kept in mind is that this

	Nehalem	Sandy Bridge	Haswell	Skylake
Cache line size	64-bytes	64-bytes	64-bytes	64-bytes
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way	32K, 8-way
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way	1M, 16-way
L3 Data Cache	8M, 16-way	8M, 12-way	8M, 12-16 way	2M/core, 11-16 way
L1 Instruction TLB	4K(page): 128(entries),4-way 2/4M(page): 7/thread	4K(page): 128(entries),4-way 2/4M: 8/thread	4K(page): 128(entries),4-way 2/4M: 8/thread	4K(page): 128(entries), 4-way
L1 Data TLB	4K(page): 64(entries), 4-way 2/4M(page): 32(entries), 4-way 1G(page): fractured	4K(page): 64(entries), 4-way 2/4M(page): 32(entries), 4-way 1G(page): 4(entries),4-way	4K(page) 64(entries), 4-way 2/4M(page): 32(entries), 4-way 1G(page): 4(entries),4-way	4K(page): 64(entries), 4-way
L2 Unified TLB	4K(page): 512(entries), 4-way	4K(page): 512(entries), 4-way	4K+2M(page)shared: 1024(entries), 8-way	4K(page): 1536(entries), 12-way
L1 Latency	4 cycles	4-5 cycles	4-5 cycles	4-5 cycles
L2 Latency	10 cycles	11 cycles	12 cycles	12 cycles
L3 Latency	depend # of cores	30 cycles	34-65 cycles	34-45 cycles
Line fill buffer	10 entries	10 entries	10 entries	10 entries

Table 1.6: Memory hierarchy key parameters
L3 latencies fluctuates depending of the number of cores, the more L3 slices, the more latency goes up. Cache TLB are presented as follow: "page size : # entries, associativity"
Source: https://www.anandtech.com and https://en.wikichip.org.

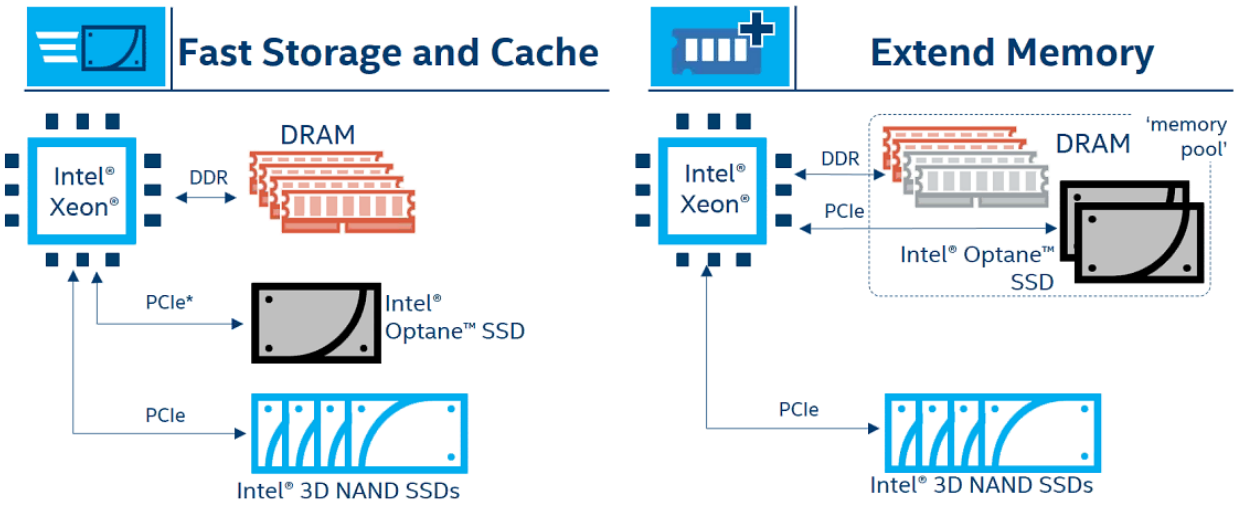


Figure 1.4: The two possible Intel Optane use cases. Source : Intel(R)-Optane(TM)-Technology-Workshop-Analyst-and-Press-Slides-322.pdf

technological breakthrough shows that memory can - indeed - be improved and that it will be a key element in the future when dealing with optimization.

Currently, most memory issues are related to data accesses and alignment, whereas most optimizations focus on how to better manage computation or parallelism, communications and synchronization. With new technologies such as Intel Optane and advancements in bus and other memory interfaces, the paradigm will certainly shift to current memory problems. The future of the evolution will be at the DRAM level with the possibility of the disappearance of hard drives and even SSDs.

1.3 Compilers

1.3.1 Introduction

Compilers are the starting point for developers to start optimizing an application. A compiler processes statements written in a human-readable language and turns them into machine language or "code" that a microprocessor chip executes. This process consists of three main steps. First, the frontend, during which the compiler checks the grammar and syntax of the code. Next, the code is transformed into an Abstract Syntax Tree (AST) with an Intermediate Representation (IR) where it applies major optimizations. Finally, the IR is

transformed into assembly code for a target micro-architecture. During the optimization process, every part of the code is statically analyzed and tries to apply as many optimizations as possible. Compilers have a plethora of available optimizations such as loop optimizations (i.e. unroll, split, interchange), vectorization, data-flow optimization (constant propagation, sub-expression elimination, induction variable recognition, and elimination) and code-block reordering. Compilers deploy the latest optimization techniques that offer many benefits to embedded systems developers. With challenging real-time performance goals, cost constraints, developers increasingly rely on the compiler's intimate knowledge of a processor instruction set and performance patterns to produce optimal code.

To help choose which optimizations to apply, static performance models and heuristics are used to avoid degrading the performances. In addition, compilers provide multiple ways to add information to help making choices. For example, developers can give hints about what to expect from some part of their codes by using directives. These hints can be information about the expected trip count for a loop, whether a loop can be vectorized, or even force the vectorization, enable/disable unrolling.

Compilers also provide different flags to trigger optimizations, adding debug information in the binary or control flow. Flags have to be specified during the compilation pass, and can be applied on a unique file or a whole project to maximize their impact. Some compilers also have an integrated Feedback Data Optimization (FDO) tool to help them in their optimization choices, i.e. Profile Guided Optimization (PGO). PGO is a dynamic analysis of a program that runs a dataset to extract specific information and applies other transformations. The three steps of PGO consist of producing an instrumented binary, generating profiling data by running the instrumented binary, and using the collected data to produce a new optimized binary.

However, even with all available optimization possibilities, compilers suffer from limitations where every compiler decision is not obviously optimal. In Chapter 4, we will discuss compiler issues and expected improvements. Usually, due to the cost in time of the research, flag optimizations are applied after the developer has performed source optimizations.

1.3.2 Limitations

Most of the time, developers trust compilers to automatically address performance issues but, sometimes, due to the static performance cost model and heuristics, compilers remain conservative when applying certain optimizations, or fail in the worst case. These static

analyses and strict performance cost models are the main limitation behind compiler optmization failures. Even though developers can provide hints to the compiler using directives, their number is limited and they are compiler dependent. Also, some compiler directives can be hazardous. For example, if a loop is forcefully vectorized, the resulting code can be very slow or worst, and can introduce bugs. The same can be said for flags. Limited in number and scope. In addition, flags are applied on a whole file or project and cannot be applied on a specific part of the code. Another limitation of flags is that they do not always allow performance gain so they have to be used partially and tested carefully. Even if they can be combined, the number of flag combinations which can be applied can increase very quickly. Assuming K compiler flags available, properly setting a string of N flags leads to exploring K^N combinations [52]. Research about flag combination is expensive because the code must be executed several times to obtain the most optimal combination. The use of the PGO to overcome the issue of the static analysis is limited. PGO lacks information on what is done and in the current implementation, the amount of information gathered at runtime is limited. Moreover, the available transformations space is still fairly small. Both of these limitations have a strong negative impact on the efficiency of the applied transformations. Another limitation is that mapping may miss opportunities for performance enhancements in exchange for correctness or portability.

1.4 Performance Analysis Tools

In the HPC industry, optimization is the key to reach peak application performance. Tools, such as profilers and analyzers can be extremely helpful in locating performance bottlenecks and can also help application developers optimize their programs and clear up these performance issues. This section presents different profilers and analyzers which can be classified into two types: static and dynamic.

1.4.1 Static Analysis

Static analysis is a method which consists of examining a code without executing it. Static analysis allows an understanding of the code structure and can predict some issues using metrics. The principal advantage of this method is that without executing the program it provides the first estimation of different issues of the code, as well as a code quality control.

IACA (Intel Architecture Code Analyzer) [48] is a static analysis tool made by Intel to statically analyze the scheduling of instructions when executed by modern Intel processors. On the one hand, it allows performing static analysis of kernel throughput and latency under

the ideal front-end, Out-of-Order engine and memory hierarchy conditions. On the other hand, it identifies the binding of the kernel instructions to the processor ports and the kernel critical path[4]. IACA enables a first-order estimate of relative kernel performance on different micro-architectures, but does not provide absolute performance numbers.

Kerncraft [45] is a tool which analyzes loop kernels using the Execution Cache Memory model, the Roofline model and actual benchmarks. It investigates the data reuse and cache requirements by static code analysis[5]. When combined with Intel IACA, kerncraft can give an overview of both in-core and memory bottlenecks and performance models can be applied on that data.

llvm-mca [16] is a performance analysis tool that uses information available in LLVM to statically measure the performance of machine code in a specific CPU. Performance is measured in terms of throughput as well as processor resource consumption[6]. It starts by parsing an assembly code, executing a module to simulate the execution of a machine and analyzing the output to generate performance reports. For example, in its report, llvm-mca estimates the Instructions Per Cycle (IPC), as well as hardware resource pressure or provides a timeline view which shows each instruction state transitions through an instruction pipeline. The main goal of this tool is to predict the performance. It also helps with diagnosing potential performance issues. The analysis and reporting were inspired by the IACA tool from Intel.

MIRA [53] is a static analyzer using Rose to perform its analysis of binaries and their associated source code. MIRA generates one AST from the source code and one from the compiled binary code, then tries to match them and uses information retrieved from these trees to improve the accuracy of the generated model. Their analysis focuses on loops and uses the polyhedral model to formalize loops in their general model.

CQA [23] (Code Quality Analyzer) is a static analysis module of MAQAO [12] that aims at tackling core level issues by modeling the processor pipeline, performing a simulation of the relevant stages of the hot-spots in a program, and providing numerous metrics that characterize these hot-spots behavior. CQA gathers multiple metrics such as an estimation of data dependencies, vectorization ratio as well as the cycle cost - from the evaluation of

[4]Source: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer
[5]Source: https://github.com/RRZE-HPC/kerncraft
[6]Source: https://llvm.org/docs/CommandGuide/llvm-mca.html

the cost of a group of instructions to know the execution latency and throughput of every instruction variant on a target architecture, execution units and decoding time of instructions.

UFS [85] (Uop Flow Simulation) is an approach combining static analysis and cycle-accurate simulation to very quickly estimate a loop execution time while accounting for out-of-order limitations in modern CPUs. UFS allows to model the impact of varying latency: this can be done uniformly on loads and stores or individually. This allows to understand the potential performance gain of: better blocking (blocking for L2 instead of L3), better prefetching (add extra prefetch instructions on targeted loads), using on-die DRAM versus external DRAM (cf. KNL).[7] In addition, it can correlate code with buffer usage and provide a detailed insight on this buffer usage. UFS allows to characterize the latency impact on loop performance with massive parameter explorations.

1.4.2 Dynamic Analysis

In contrast to static analysis which only requires source code and/or binary, dynamic analysis needs to execute the program to gather information. The objective is to find what happened during the execution of a program rather than by repeatedly examining the code offline. The dynamic analysis can be split into sampling and tracing and each has advantages and drawbacks. Both are made to reveal performance issues and to identify which parts of a program the process of improvement will be the most time consuming. Efforts will be focused on investigating and improving them. Sampling is statistical profiling, when the program is repeatedly interrupted as it runs, with a fixed interval between interruptions. The purpose of each interruption is to take a sample. This is done by visiting each running thread, and then examining the stack to discover which functions are running[8]. On the other hand, tracing uses an event logged in the program. This log has to be detailed enough to report execution of function calls, returns, and other statements. Tracing may require the program to be instrumented by inserting instrumentation directly in the source code or dynamically into the machine code.

Using Sampling

Sampling is a statistical profiling process or technique that takes samples. This method does not require any modification of the binary and can use hardware counters to get information about what happens during the execution of the program. Sampling is often sufficient to

[7] Source: https://dyninst.github.io/scalable_tools_workshop/petascale2017/assets/slides/SWT2017_WJA_data_latency_access_V9.pdf

[8] Source: https://www.jwhitham.org/2016/02/profiling-versus-tracing.html

pinpoint load imbalance due to problem decomposition and/or identify the origin of excessive communication time[9]. It is fastest, cheaper than tracing and can be useful as the first step to categorize some issues and to localize performance problems but it can lack accuracy.

Scalasca [39] is a software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks, in particular, those concerning communication and synchronization, and offers guidance in exploring their causes[10]. Scalasca focuses its profiling on parallel issues, such as communications and synchronization inefficiency but nothing on computation issues. The Scalasca analysis can be visualized in other tools, such as cube [29], Paraver [66] or Vampir [112].

VTune [4] is a commercial application for software performance analysis of Intel architectures. It provides 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 subroutine 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 also be used to analyze thread and storage performance[11].

PerfExpert 4.0 [89, 20] is a tool that diagnoses performance bottleneck at the function and loop level for any core/socket/node level. It is able to perform source transformation based on specific patterns on identified bottlenecks. PerfExpert also provides an analysis report with hints on how to remove bottlenecks if optimization cannot be done automatically using their source transformation. The report gives an overview of multiple elements of the program. For example, it gives details on the data accesses (hits and misses of all cache levels) and instruction accesses by counting the LPCI arising from accesses to memory: memory accesses for variables, floating point instruction by counting the LPCI from executing floating point instructions, gflops peak. The number of transformations are pattern based and limited to loop interchange, tiling and fission. Moreover, PerfExpert uses HPCToolkit for the analysis and the Rose compiler, for source-to-source, to perform transformations[12].

[9] Source: http://ipm-hpc.sourceforge.net/profilingvstracing.html
[10] Source http://www.scalasca.org/
[11] Source: https://en.wikipedia.org/wiki/VTune
[12] https://github.com/TACC/perfexpert/blob/master/doc/user_manual/pdf/user_manual.pdf

MAQAO LProf (Lightweight Profiler) is a MAQAO module for the purposes of quickly profiling an application at runtime by characterizing the loops and/or the functions. It is used as an entry point of the MAQAO tool-chain as it allows to quickly target the most relevant part of the code. In order to perform its profiling, LProf uses sampling. It stops the processor at regular intervals and captures the current value of the instruction pointer and some of elements that are useful to complete the characterization (i.e stack state). This profiling technique is almost non intrusive and offers a good accuracy, while keeping the overhead very low. The sampling method relies on the Linux perf event open syscall which sets up the Performance Monitoring Unit (PMU) and on Ptrace to track threads and/or processes and configure the PMU for each of them. LProf can provide as many classifications as hardware counter events available on the target processor.

Using Tracing

In contrast to sampling, the trace is a log of events within the program allowing it to be more accurate than sampling. Tracing requires program instrumentation where probes are inserted either in the source code or in the binary. The amount of data in the trace increases with the runtime. As such in order to bind the memory usage by the tracing one must periodically write the data out to a disk or a network. Tracing is useful for detailed examination of timing issues occurring within a code.

Score-P [55] is an instrumentation framework, that allows users to insert instrumentation probes into their codes to collect data (e.g. times, communications, hardware counters, etc.). It is a system which instruments and analyzes codes for numerous other external tools and provides a number of components that interact with each other, such as like an online interface, cube and supports external tools, such as TAU, Scalasca, Vampir.

Callgrind [21] is a profiling tool which uses runtime instrumentation via the Valgrind framework for its cache simulation and call-graph generation. This way even shared libraries and dynamically opened plugins can be profiled[13]. Callgrind generates files and performance results can be visualized with KCachegrind. It uses the processor emulation of Valgrind to run the executable and catches all memory accesses for the trace. The user program does not need to be recompiled; it can use shared libraries and plugins, and the profile measuring does not influence the trace results. The trace includes the number of instruction/data memory accesses and 1st/2nd level cache misses. It relates them to source lines and functions of the run program.

[13]Source: http://kcachegrind.sourceforge.net/html/Home.html

MAQAO VProf [47] (Value Profiler) is also a MAQAO module which is able to find various characteristics of a given subset of loops and/or functions. It uses the patching capabilities of MAQAO to instrument target functions and loops. Using MAQAO core analysis and to the intermediate representation of the instructions, assembly code can be injected at any location in the binary (including between loops iterations) without altering the program original task, and thus obtain a functional program that can run any additional profiling code. VProf includes multiple probes and instrumentation algorithms which allow numerous measurement types. If requested, these probes can operate with thread local storage (TLS) variables and work in a multi-threaded environment. The VProf-driven instrumentation is able to provide, among other metrics, the number of cycles spent in a given iteration of a loop, the parameters of a function, the number of iterations of a loop or the number of instances of a function.

MAQAO DECAN [57, 15, 58] (DECremental ANalysis) is a module slightly different from the other tools. It runs a differential analysis (extension of the decremental analysis) on loops or basic blocks, to obtain a concise idea of the nature of the application bottleneck(s), its potential future bottlenecks, and the potential gain if those bottlenecks are cleared. DECAN uses the patcher to create multiple versions of the profiled piece of code and measures the cost of every variant. Each version removes or modifies a certain class of instructions (except those involved in the control flow); among all variants, one is the DL1 (Data in L1) variant which modifies all memory accesses to force their data to be in the L1 cache.

Once the measures are done, DECAN performs a differential analysis by comparing the cost of each variant with the cost of the original code. This results in invaluable information such as saturation ratios which represent how many memory accesses or floating operations are limiting the performance, and give a deep insight into the potential gain once the application is optimized.

1.4.3 Global view using both

In this section, we consider tools that combines both, static and dynamic analysis to provide a more accurate view of the performance of a program.

HPCToolkit [3] is an integrated suite of tools for measurement and analysis of program performance. By using statistical sampling of timers and hardware performance counters, it collects accurate measurements of a program work, resource consumption, and inefficiency and attributes them to the full calling context in which they occur[14]. HPCToolkit provides

[14]Source: http://hpctoolkit.org/

a sequence of analysis which consists of : 1) measuring execution costs (hardware consumption cost); 2) analyzing source code structure; 3) attributing measured costs to source code structure combining the dynamic profile with the source code analysis to attribute measured costs incurred during the execution of the program to meaningful source code constructs. It also provides a visualizer to see attributed costs in source code or timeline views.

Tau [103] (Tuning and Analysis Utilities) is a software capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling[15]. TAU provides tools that support sophisticated views of a program structure. Currently, the code analysis systems have been used to analyze C++ source to automatically generate TAU profiling instrumentation. It also provides Paraprof, a graphical interface to visualize all results in an aggregate and per node/context/thread form. The user can quickly identify sources of performance bottlenecks in the application using this graphical interface. In addition, event traces can be used by other visualizers such as Vampir, Paraver or JumpShot.

Intel APS (Application Performance Snapshot) [9, 119] is the Intel toolbox that gathers in one view: Intel Trace Analyzer and Collector, Intel MPI Tuner, Intel VTune and Intel Advisor, to take a quick look at the program performance issues at multi levels: Cluster with MPI imbalance and bounds and thus see detailed rank-to-rank communications; Node, to see CPU/Memory bound and thread scalability issues, disk IO, memory access, etc; and finally Core, with the analysis of the vector efficiency (i.e. FMA, FPU). The goals of APS is to identify optimization areas with detailed reports while having a low overhead and a high scalability all in a global and simple view which gathers all details issues and bottlenecks.

MAQAO ONE-View ONE-View is a MAQAO module that drives the execution of other MAQAO modules (LPROF, VPROF, CQA and DECAN) in order to produce synthetic reports. Static and dynamic analysis are used according to the analysis description. Experiments are configured using a file or some command line parameters. Produced reports can be formatted as HTML web pages, XLSX spreadsheets or simple text and gather all metrics of executed modules. ONE-View defined several built-in reports:

- one - The fastest report. It combines LPROF profiling with CQA static analysis. It needs only one run of the application.
- two - It contains all data from "one" with some metrics produced by VPROF and DECAN. It needs four or five runs of the application according to the configuration.

[15]Source: https://www.cs.uoregon.edu/research/tau/home.php

- three - It contains all reports from "two" with a lot of DECAN metrics. It needs about twenty runs of the application and is the longest report.

In addition, users can define their own reports by filling in a report description file to filter which module to execute and metrics to compute.

1.5 Optimization Tools & Techniques

All optimization techniques are different. For example, specific directives can transform a sequential code into a parallel one (i.e. OpenMP). Another technique consists in finding the best compiler flags combination or provide a domain specific language which facilitates the writing of complex operation. The remainder of this section presents different optimization tools and techniques aiming at providing assistance to optimize developers codes.

1.5.1 Compiler Optimization Techniques

Compilers provide a wide range of optimization techniques, which are listed below.

- Data-flow analysis: gathers information about the possible set of values calculated at various points in a computer program. A program control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate[16].
- Partial evaluation, dead code elimination and common sub-expression elimination, to reduce code size (i.e. "a*1" reduce to "a").
- Inline expansion: replaces a function call site with the body of the called function. Inlining will improve speed at very minor cost of space, but excess inlining will hurt speed, due to inlined code consuming too much of the instruction cache, and also cost significant space[17].

[16] Source: https://en.wikipedia.org/wiki/Data-flow_analysis
[17] Source: https://en.wikipedia.org/wiki/Inline_expansion

- Instruction scheduling: improves instruction-level parallelism by avoiding pipeline stalls by rearranging the order of instructions, and avoiding illegal or semantically ambiguous operations (typically involving subtle instruction pipeline timing issues or non-interlocked resources). The pipeline stalls can be caused by structural hazards (processor resource limit), data hazards (output of one instruction needed by another instruction) and control hazards (branching)[18].
- Common loop optimizations include interchange, splitting, unrolling, etc.
- Automatic parallelization by converting sequential code into multi-threaded or vectorized code in order to utilize multiple processors simultaneously[19].
- strength reduction: expensive operations are replaced with equivalent but less expensive operations[20] (i.e. replacing a multiplication by an addition with a loop, or an exponentiation by a multiplication with a loop).

This non-exhaustive list presents most common optimization techniques in most compilers and automatically applied during the compilation process. Other optimizations and compilation passes can be triggered or configured by the user with flags.

1.5.2 Flag Research

Compiler flags allow to easily obtain good performance gain [91] and can be combined for even better gain. Many researches have been done on the different techniques to obtain the best combination flags [60, 92, 90]. Flags (including PGO) can help to obtain good performance for example, over the full SPECFP2006 database, flags provides an average 5% gain of performance and reach a peak at 60%, but the research of the right combination of flags if very expensive [52]. Tens of flags exists which perform different kinds of operations, like adding control flow protection, adding debugging information. Some of them are even predefined combination of flags, for example, -O2 and -O3 are respectively a combination of 45 and 60 other flags. PGO flag is a specific flag which allows overcoming the static limitation; it requires to execute the program on a representative dataset and is expensive.

1.5.3 Profile Guided Optimization (PGO)

A typical PGO process encompasses three steps:

[18] Source: https://en.wikipedia.org/wiki/Instruction_scheduling
[19] Source: https://en.wikipedia.org/wiki/Automatic_parallelization
[20] Source: https://en.wikipedia.org/wiki/Strength_reduction

- Producing an instrumented binary using a special compiler flag or multiple flags;
- Executing the resulting binary in order to obtain a profile (feedback data);
- Using the obtained feedback data during the compilation process to produce a new version supposed to be more efficient.

The last step enables some specific optimizations and can also modify the behavior of other optimizations. For example, for Intel compiler PGO, among other optimizations, it allows the following:

- Use feedback data on function entry counts. Function grouping is done to put hot/cold functions adjacent to one another;
- Profile value of indirect and virtual function calls in order to specialize the indirect function call for a common target;
- An intermediate language annotated with the edge frequencies and block counts. They are then used to guide a lot of the optimization decisions made by other passes of the compiler, such as the in-liner and partial in-liner, the basic block layout, the conversion from switch tables to if statements, loop transformations like unrolling, etc.

PGOs overcome the static limitation but is a black box where the user cannot add any information and does not really know what is done after the process. In addition, the research space as the associate transformation is limited.

1.5.4 Libraries

Specialized libraries allow achieving good performances to perform specific operations because generally, they parameterize the algorithm implementations based on hardware characterization to generate multiple code variants from automated code transformations and data size selection. Two well-known examples are ATLAS [100] and PHiPAC [17], two self-tuning solver libraries for the basic linear algebra subroutine (BLAS [33]). Similarly, the Optimized Sparse Kernel Interface (OSKI) library, which is a collection of low-level C primitives that provide automatically tuned computational kernels on sparse matrices. It targets sparse BLAS and its indirect, irregular memory access and low computational intensity. Another well-known library is FFTW [36] which is autotuned to perform the fast Fourier transform

algorithm. SPIRAL [97] solves linear digital signals, it is generated and tuned for specific problem case and target architecture. Each high-performance library only provides a small number of functions to solve a specific problem, without letting the user interact during the process. Libraries are tuned for target architecture so they have to be installed on all computer where the program will be run.

1.5.5 Directives

Directives are annotations in source code intended for a compiler or a tool that will perform an operation according to that directive. It is the lightest and simplest method for developers, they just have to insert a line of code to trigger a transformation. As such, the application source code is kept portable and the original version can be built using a traditional compiler. Probably the best known one using this technique is OpenMP [79] which transforms block of sequential code into parallel one. HMPP [31] and OpenACC [77] also aim at transforming sequential code into parallel one only using directives. They are compatible with OpenMP and MPI but also designed to handle hardware accelerators such as GPUs. Compilers already provide directives, however, existing one cannot always satisfy users' requirements. In some cases, customizing the compiler directives for application-specific code transformations is required in order to achieve a high-performance as well as high-performance portability by using different performance optimizations for individual systems. Xiao et al. [122] propose a mechanism that allows programmers to customize existing compiler directives. Orio [46] is an autotuning framework performing source-to-source transformations; the tool automatically tuning the performance of codes written in different source and target languages, including transformations from a number of simple languages (e.g., a restricted subset of C) to C, Fortran, CUDA, and OpenCL targets. The tool generates many tuned versions of the same operation using different optimization parameters and performs an empirical search for selecting the best variants among all multiple optimized code. Directives are simple to use but are limited in their transformation space and are compiler/tool dependent even if we find an equivalent of most popular directives in each compiler.

1.5.6 Domain Specific Language

Domain Specific Languages (DSLs) are computer languages specialized to a particular application domain to solve particular problems in a particular domain and is not intended to be able to solve problems outside it (although that may be technically possible). Plenty of Domain Specific Languages are available for performing source-to-source transformations, such as DMS [114], XLanguage [32], rascal [54],Stratego [69], TXL [27], SmaCC [19], Locus [107]

or POET [123]. These tools allow the user to define their own source program analysis and modifications of source code via a new language especially designed for that. Some DSLs are also designed to write operations that can be complex in another language more easily, like CIL [75] which transforms CIL program to C ANSI, Microsoft C or GNU C. Among these DSLs, some are designed to implement parallel transformations: Paraformance [88], PPCG [113], etc. DSLs are more powerful than directives because developers can exactly express what they want. However, it is dangerous to assume they will be willing to invest time and resources to write their own transformations, even if the interface is based on a well-known language such as Fortran.

1.5.7 Source-to-Source

Source-to-source transformation is another level of optimization. Generating source code is the more portable way to optimize a code, in addition, the output code can be built with any compiler. A lot of existing tools allow to perform source-to-source transformation, most of them focus on optimizing loops using the polyhedral model. The polyhedral method treats each loop iteration within nested loops as lattice points inside mathematical objects called the polyhedral. It performs affine transformations or more general non-affine transformations such as tiling on the polytopes, and then converts the transformed polytopes into equivalent, but optimized, loop nests through polyhedral scanning. The polyhedral model allows to achieve good performance on loops that can be handled and is especially used for the parallelism, but it can be applied on a small set of loops. PoCC [73] is an example of source-to-source compiler embedding multiple tools based on the polyhedral model, such as the "Legal transformation Space explorator" (letSee) [94]. For iterative compilation using affine multidimensional schedules; PLuTo [18] is an automatic parallelizer and locality optimizer for multi-cores, used for powerful optimization with tiling and parallelism in the polyhedral model; and the Chunky Loop Generator (CLoog) [13] is used to generate syntactic code from the polyhedral representation. Other tools and frameworks that do not use the polyhedral model are specialized for one type of transformation. For example, the "Tile Loop generator" (TLoG) [5] focuses on generating a tiled version of a loop, or Scout [59] which transforms a simple C or C++ loop into a vectorized one using SIMD instruction via a graphical interface. All are not as specific, the other tools are more oriented towards parallelism, whether for GPU or CPU [106, 44, 37, 50]. Usually, for source-to-source transformation users have to develop they own transformations or trigger these transformations themselves on the statement they want to transform. However, tools exist that do not require any assistance from users. These tools are auto-tuning tools. The goal of these tools is to test several transformations on a statement, keep the best and start again somewhere else.

1.5.8 Auto-tuning

Auto-tuning is an optimization method which aims at improving performances of a program based on an automated procedure, using the empirical method or a model-driven performance optimization, while maximizing productivity without sacrificing portability. Basically, auto-tuning creates multiple variants of a code and analyses which one delivers the best performance. Auto-tuning tools differ on how they generate their versions and which metrics they use to choose the best version. Autotune tools work on changing compilers setting like modifying flags as described in section1.5.2, using alternative algorithms or applying code transformations. Due to performance space that can potentially be large and complex, tools have to limit their research procedure to partial evaluation only. Active Harmony [1] illustrates what auto-tuning tools can do. It applies search strategies within a space of possible combinations to support application-level tuning by simultaneously evaluating user-defined tunable parameters. Multiple transformations are tested (i.e. loop unrolling, blocking and scheduling). Most of auto-tuning tools use FDO techniques to collect information and drive their choices. For example, CHiLL [24, 108], Aestimo [83] or AutoFDO [43, 25] use FDO techniques to refine their code generation strategy during empirical iterations.

1.6 Conclusion

In this chapter we showed how the increasing complexity of the micro-architectures and some key parameters (e.g. increased vector lengths and advanced memory hierarchy) are heavily relied to performance issues.
Even compilers are not able to provide an optimal code each time, despite all available optimization possibilities. They suffer from limitations related to their static analysis, heuristics and a strict performance cost model. Moreover, even if compilers can be guided by user directives and flags; these ones are limited in number and can be ignored. Compilers can also help with a dynamic analysis using their PGO mode. It lacks information on what is done and in the current implementation, the amount of information gathered at runtime is limited. Moreover, the available transformations space is still fairly small.

Developers must turn to performance analysis tools to optimize their codes. However, even if these tools are pretty good at identifying specific issues, none of these tools automatically optimize the application. They only provide information on the application and what happened during the execution of a program; in the best case they can return hints about how to improve the code. However, most of the time it requires the need of an expert to understand all of the data and correctly optimize the code.

Numerous tools and techniques exist to improve a code from using directives/flags to drive compiler choices to rewrite the code using a DSL to adapt the language to the need, but the source-to-source remains a good balance between expressiveness, optimization possibilities and time spent to optimize the code. This allows to no longer be dependent on the compiler without rewrite the whole project into a new language while providing good results.

Chapter 2

ASSIST

This chapter presents the characteristics of ASSIST (Automatic Source-to-Source assISTant). It is a semi automatic source-to-source framework to optimize source codes using performance evaluation tools which can also be fully or partially guided by the user. This framework aims at optimizing industrial source codes, focusing on loops and functions. Transformations can be triggered by the user who previously inserted directives in the code, or according to the results of performance evaluation tools (e.g. MAQAO modules). The originality of the approach lies in the combination of both source-to-source transformations using annotations and FDO approaches. More precisely feedback data drive source-to-source transformations to achieve both productivity and performance. In this chapter we also present the state-of-the art of existing tools able to perform source-to-source transformation and the specific tool we chose as a foundation for ASSIST according to the constraints specific to our problematic.

2.1 Background

2.1.1 Specifications / Context

The previous chapter presented the evolution of computer architectures that quickly evolved during the last decade. This evolution introduced new difficulties which prevent to use the whole capacities of HPC computers. Due to these evolution and difficulties, it becomes harder to manually optimize and maintain codes. Performance evaluation tools emerged to help developers to overcome these problems, but most of them require either to be an expert, or at least to get the help of an expert, to understand all metrics and exactly know how to optimize a code.

We want to propose a new framework to help developers to both optimize their codes according to the requirements of a targeted architecture as well as keep these codes maintainable, by using the results from evaluation tools. This new framework must be a semi-automatic source-to-source framework. It must be able to provide transformations of an input source code according to the metrics issued by the evaluating tools and following the directives inserted by the developer in this code.

Nowadays, there are several compilers which can manage source-to-source transformations and there are also frameworks using compiler frontend to produce these transformations. Our purpose is to search for the compiler that will best answer the following constraints :

- No intermediate representation : the compiler has to generate an AST and to allow its manipulation before any optimization is performed. Moreover, we want an output code that remains at source level and not in a compiler-specific intermediate representation (IR).
- Input Languages : We give priority to scientific applications (HPC field), hence selecting C, C++ and Fortran languages.
- Framework license : It has to be free but not GPL (General Public License) or at least LGPL(Lesser General Public License) to be integrated into MAQAO.

2.1.2 Existing Tools for Source-to-Source Transformation

There are many available compiler infrastructures and specialized source-to-source frameworks, but only very few can satisfy our requirements. Most of these tools only perform a very restrictive set of transformations or only on a subset of a language. In this section, we present and compare a non-exhaustive list of the main compilers or tools allowing to perform source-to-source. Our presentation is limited to the main tools that could be used as a basis for ASSIST.

Cetus

Cetus [30] is a compiler specialized in source-to-source transformations. It supports C ANSI and has a fairly complete and documented API (Application Programming Interface) oriented to generate parallel codes. Its license is under the OSI (Open Source Initiative). Cetus cannot be chosen due to its usage of a single language (C ANSI) which limits the number of industrial applications; a second limitation is that it would be complicated to integrate it into MAQAO due to its Java implementation.

Par4All & PIPS

Par4All [6] is an automatic source-to-source program transformations for GPU-like. It is based on PIPS [50], a compiler allowing to transform a sequential code into a parallel one with OpenMP, Cuda or OpenCL, by transforming a sequential loop into a parallel section. It handles C and Fortran and is under MIT License, a license of free software and open source, no copyleft. Par4All and PIPS only allow to transform programs into parallel programs, the number of functions available is limited and it simply performs automatic transformations without letting the user operate. Furthermore C++ is not handled.

OpenC++

OpenC++ [78] is an open-source (BSD license) C++ frontend and refactoring library. It enables the development of C++ language tools, extensions, domain specific compiler optimizations and runtime meta-object protocols. It is maintained by a group of volunteers but has not been updated since 2004. OpenC++ was implemented to assist other programs to easily analyze C++ codes or to perform source-to-source transformations. The programmer who wants to use OpenC++ writes a meta-program in which specifies how to translate or analyze a C++ program; this plug-in is then compiled by the OpenC++ compiler and linked to it as a plug-in.

It was designed to enable the users to develop those tools without being concerned by tedious parts of the development such as the parser and the type system. As described, it only handles C++ and is dependant of the OpenC++ compiler.

DMS Software Re-engineering Toolkit

Developed by "Semantic Designs", DMS (Design Maintenance System) [114] is a commercial compiler. It can be used to construct analyzers that generate reports or to find and fix coding, using previously mentioned analyzer outputs to locate issues and choose/apply transformations to resolve them. DMS is a compiler which has : a parser for different languages (Java, Cobol, C, Fortran, Ada, etc.); a set of semantic analyzers (including a variety of pattern matching engines); a set of compiler data structure modification engines (including source-to-source program transformation engine) and final output formatting components (converting compiler data structures back to valid source code rather than binary code).

The developer can apply source-to-source pattern transformations or write procedural transformations, and then regenerate compilable source text corresponding to the transformed program. DMS software is a reference in the domain of source-to-source transformation because it can translate many languages into others, or optimize a source code by

applying multiple transformations. Due to its commercial environment we cannot base ASSIST on DMS.

LLVM (Clang & Flang)

Clang and Flang are two compiler frontends for C family and Fortran language. They use LLVM (Low level Virtual Machine) as their middle and back end. They are designed to offer a complete replacement to the GNU compiler Collection. Their source codes are available under the University of Illinois/ NCSA License. LLVM [67, 115] has an entire API to perform AST analyze and manipulation but even if theoretically possible with the existing rewrite system, it is not intended to provide a real source-to-source system.

Scout

Scout [59] is a configurable source-to-source transformation tool designed to automatically vectorize C source code. Scout tries to cover a wide range of loop constructs and is capable of targeting various modern SIMD architectures. It provides the means to vectorize loops using SIMD intrinsic instructions sets like SSE or AVX at source level. To vectorize a loop with intrinsics, it only requires to insert a simple directive above the intended loop and Scout replaces the current loop body by a vectorized one.

Scout only has a C frontend and a function to perform vectorization transformation. Moreover, it is not designed to be integrated into an other framework.

Orio

Orio [46] is a Python framework, under the MIT license, for the transformation and the automatic performance tuning of codes written in different source and target languages, including transformations from a number of simple languages (e.g., a restricted subset of C) to C, Fortran, CUDA, and OpenCL targets. The tool generates many tuned versions of the same operation using different optimization parameters, and performs an empirical search for selecting the best optimized code variants.

ROSE

ROSE [99] is a compiler specialized into source-to-source transformations. It processes C, C++ and Fortran among other languages. It provides a wide API to analyze and modify AST and a whole rewriting system to generate code after modification. Moreover, ROSE is under BSD (Berkeley Software Distribution) license.

ROSE is the only compiler corresponding to all our constraints and able to easily perform real source-to-source through a large API. That is why we chose ROSE as the base for ASSIST. ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools. It is developed at the Lawrence Livermore National Laboratory (LLNL). The goal is to provide the required support to easily build tools that operate on source code (analyzing or optimizing). ROSE is a library that makes it easy to build a wide range of tools from optimizing source-to-source compilers to special purpose analysis tools. ROSE supports: Fortran (66,77/95/2003), C89, C99, C++, OpenMP applications. ROSE uses Open Fortran Parser (OFP) to parse Fortran codes and Edison Design Group (EDG) to parse C/C++ codes. These frontends produce a ROSE intermediate representation that is converted into an AST by ROSE. ROSE provides a large API to analyze and transform the AST, making it the ideal tool for our use.

ROSE uses and develops SAGE III originally known as SAGE++, developed at the university of Indiana before the ROSE team took over the API. This API is the core of the AST with each class representing each node.

As previously described, ROSE can handle C and C++ codes with the EDG frontend, but it is a black box that cannot be modified, so any problem encountered with C or C++ codes cannot be solved by modifying the way the frontend parses the file and no rules can be added. Contrary to the Fortran parser (Open Fortran Parser (OFP)) this frontend can parse Fortran from 77 to 2003 and is open source. We had to modify it to better manage comments and directives which were not natively handled.

Even if ROSE can handle C, C++ and Fortran and seems robust at first sight on these languages, it was particularly suffering of a bad management of Fortran. We had to fix some part of the ROSE code to better handle some Fortran aspects required by our industrial applications. Among all these changes, we had to modify the generated output (e.g. missing space, indentation, keywords not handled, missing element in "include" list, revert list, etc); we also had to upgrade the management of comments and directives and add the management of missing keywords.

Summary

Table in figure 2.1 presents a summary of previous existing tools and their abilities to manage our constraints. We can see that ROSE is the only one which respects all these constraints and seems robust enough to handle HPC language and thus serve as a basis for ASSIST. In addition, it is an open source framework allowing us to modify what could hamper us to

	License	C	C++	Fortran	Source-to-Source	Documentation	Weakness
Cetus	OSI	✔	x	x	✔	✔	Only handle C codes
Par4All	MIT	✔	x	✔	✔	✔	Only working on polyhedral for Parallelism
OpenC++	BSD	x	✔	x	✔	~	Used only as IDE plug-in
DMS	Commercial Product	✔	✔	✔	✔	✔	Commercial product
LLVM	BSD	✔	✔	✔	~	✔	Not intented to perform Source to source
Scout	BSD	✔	✔	x	✔	~	Only perform vectorization
Orio	MIT	~	x	x	~	x	Only handle a subset of C, Intented to translate A language to another
ROSE	BSD	✔	✔	✔	✔	✔	EDG frontend for C/C++ Some fix to do to handle fortan

✔	Requirement OK
~	Theoretically Possible / weak
x	Requirement NOT OK

Figure 2.1: Summary of existing tools performing source-to-source.

manage certain industrial applications. All the others are either commercial or do not handle the three required languages or are only designed to focus on polyhedral loops.

2.1.3 MAQAO

Figure 2.2 presents a general view on MAQAO[12, 109, 111] architecture. Three main blocks can be identified, namely: Modules, APIs and MAQAO core. At the top of MAQAO, we have modules. Among these modules, there are analyzers and profilers which return their analysis to users in the form of HTML report or raw data. All modules used by ASSIST have been presented in Section 1.4. These modules are performance analysis software which exploit the data structures and APIs offered by the lower parts of MAQAO. The APIs allow to manipulate an intermediate representation of loops, functions and basic blocks, defined

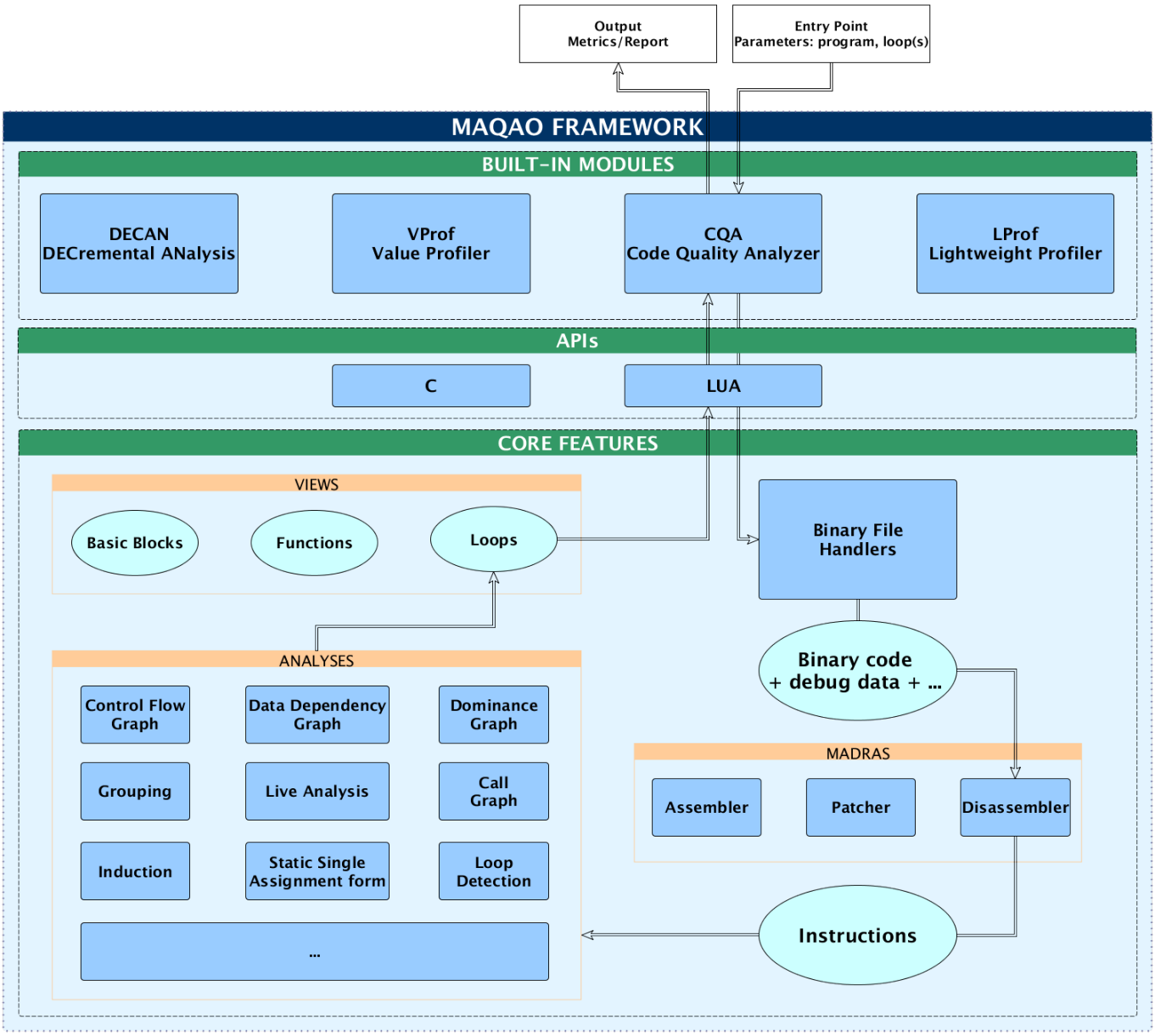


Figure 2.2: Process of a profiling with MAQAO overview.

at the core level. The core level allows to disassemble a binary into instructions that will be sent for analysis before to be built as IR. The following analysis are performed to construct the IR:

- The control flow analysis: it takes the instruction stream returned by MADRAS and, by following the branch instructions, constructs the control flow graph (CFG) for each function.
- The loop detection analysis: it gets the CFG as an entry and performs a transversal depth-first search within it to construct the loop hierarchy.
- The advanced static analysis: it consists of several static analyses with the aim of polishing the CFG and Call Graph (CG), such as indirect branches resolution, static single assignment (SSA) and groups detection. Several useful data structures result from this module. These include: the CFG, the Loop Hierarchy Graph (LHG) and the CG. A rich API is also exposed for the extension modules to exploit these structures.

After IR is constructed, modules use APIs to manipulate loops, functions and basic blocks and thus apply the different analysis before returning a report or computed metrics either to the users or to ASSIST with the chosen output.

2.2 Design & Implementation

2.2.1 Overview

ASSIST is an open source FDO tool and a framework based on the ROSE compiler infrastructure and integrated into the MAQAO tool-set. Among existing analysis tools, MAQAO has been chosen because it offers multiple modules allowing to analyze both statically and dynamically HPC codes. It provides a plethora of metrics to guide our optimizations with tools such as: CQA which can statically analyze a code and, among other metrics, has the vectorization ratio of a loop; VProf which can provide the number of iterations of a loop; or DECAN which analyses and modifies the code to test if it is profitable to fit data in L1, for example. Figure 2.3 presents an overview of the steps involved in the tool operation. The following section will provide more details on transformations and examples illustrating this process.

The user is at the center of the process, he can drive each step. ASSIST provides users with a simple yet flexible interface that offers multiple alternative approaches to transform a source code:

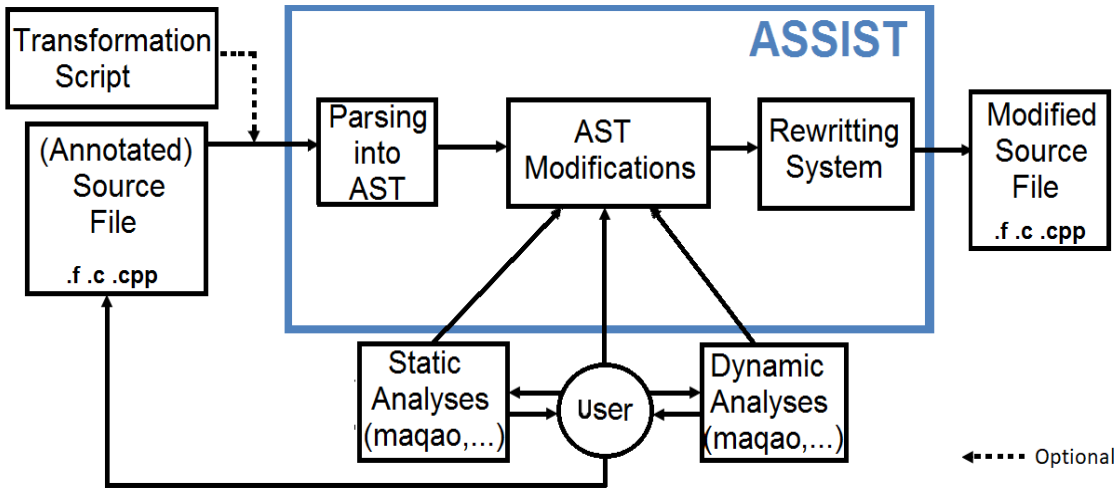


Figure 2.3: Overview of tool usage. The user decides what static and dynamic analyses have to be performed. Transformation script is a Lua script where the user specifies transformations to be applied, avoiding to directly modify source code. Alternatively the user can let ASSIST directly use profile to perform transformations.

- The first one makes use of directives that the user can add above a loop or a function to trigger a transformation. For example, the directive !DIR$ MAQAO UNROLL=4 above a loop triggers the unroll (factor of 4) of the loop, if applicable and by running the following command: "maqao s2s -option="apply-directives" -src=<source file>" the transformed code can still be compiled and even reviewed by the programmer if necessary.

- With the second approach, ASSIST transformations can also be guided by giving information through a Lua script instead of annotating the source file with directives. A detailed template is available in appendix A.6.

- With the third, when feedback data from MAQAO are available, it is also possible to leverage optimization opportunities. ASSIST is able to use MAQAO API to search for information about loops and files into a source file or binary and read MAQAO tools results to apply transformations.

Available analyses are based on MAQAO CQA (code quality), VPROF (value profiling) and DECAN (binary modification and comparator). The user has to select MAQAO modules and metrics that will automatically trigger ASSIST transformations. Feedback data can

be mixed with user's directives to guide transformations. During the modification process the source code is parsed and transformed into an abstract syntax tree (AST) that ASSIST will transform accordingly to user's directives and/or MAQAO results. As a semi-automatic framework, ASSIST can interact with the user during the AST modifications, asking him information on potential issues with the transformation. The user can then share his knowledge and decide whether or not to apply this transformation. At the end of the process, the modified AST is parsed to generate a modified source file as output. We decide to implement a semi-automatic tool because the automatic approach is limited by its cost and by some issues of transformation legality. Our system first alerts the user with an estimate of the costs induced by the exploration of the potential of a given transformation. He can then decide to select only a few transformations. Then our system also let the user force a transformation because he knows that the transformation is legal.

2.2.2 ASSIST Principle

As all MAQAO modules, ASSIST has a Lua wrapper to handle all input information that a user can give : input source file, output name, MAQAO results, and the features he wants to use. In addition, the processing of some of the information management is done at this level : the creation of a "rmod" file which corresponds to the header of Fortran module required by ROSE, the management of MAQAO results and information, etc. Then all information are sent to another part of ASSIST, written in C++. This part takes care of applying transformations on the AST nodes based on information from the Lua part. At first, it will browse all statements and create a sub-tree of functions and loops. Then it will try to match all these information and find loops and functions to transform, including those with directives above them. Finally, it will apply transformations, update information in the sub-tree and mark the statement to not apply another transformation that would be incompatible with the first one. In the end, it uses ROSE back-end and its rewriting system to create the new output. We disabled the part of the ROSE back-end which compiles the new output with GCC-4.4 because it does not obviously works with industrial applications which require a more recent version of GCC or another compiler (i.e. ICC).

2.2.3 Integration Into MAQAO

ASSIST is a MAQAO module. That means that it has access to MAQAO core (binary and analysis layers) and can also communicate with other MAQAO tools through an internal API. MAQAO tools deal with binary function and loop objects. Since ASSIST manipulates source code, it must perform a mapping between real source lines and those provided by the compiler through debug information. That way, ASSIST can establish a link between

source and binary functions/loops. In order to make loops match between source and binary, ASSIST starts by gathering information from MAQAO that will give the first and the last loops lines to be managed according to debug information provided by the compiler, which can be more or less accurate. Then ASSIST create a sub-tree only composed of loops of the file with all their information: what kind of loop (e.g. "for", "while", "do", etc.); line start and end of the loop; a pointer on the loop; if we are already making match with a binary loop; and if they are other loops inside the body of the loop. Then ASSIST browses this sub-tree trying to match lines start and stop of binary and source, first at the precise line then, if the loop has not been found, by looking around with up to three lines before and after where it should have been found. A delta of three lines is well balanced because it decreases the number of missing loops without increases the number wrong match between binary and source loops. If binary information have not matched with source code a warning is raised at the end of the process. The current implementation of ASSIST uses four MAQAO modules: LPROF for profiling (hotspots); CQA for code quality metrics (i.e. vectorization efficiency); VPROF for function and loop value profiling; and DECAN to check if it is interesting to block a loop. For this last part, DECAN generates and measures special loop variants where all operands are accessed from L1. In this paper only features used by ASSIST are mentioned.

2.2.4 Interaction With The User

The automatic approach is limited by its cost and by some issues of transformation legality. Also, users often know their code better than what we can detect with any tools. To overcome this limitation, as we can see on figure 2.3, we decide to let our system open to users and to add interactions with them by providing: first, an estimate of the costs induced in the exploration of the potential of a given transformation. Which means that users have to choose which are the most adapted profilers and metrics in terms of costs and of analysis performed. For example if a user knows that any of its loop can be, or has to be tiled, he will disable the DECAN profiling which only triggers the tiling transformation. Then, after analysis, ASSIST can present the user with the different possible transformations according to the previously returned metrics. Then, the user has to select which transformations to perform. Moreover, our system also lets the user force a transformation when he knows that the transformation is legal.

2.3 Explicit Supported Transformations

In this section, all transformations that ASSIST explicitly performs are described; all these transformations are well known. They are shortly detailed below, from common ones like

loop unrolling to less common ones like loop and function specialization, to set a common dictionary. Moreover, the specialization transformations have been specifically designed to be combined with other available onces. Most of these transformations can be done by inserting directives to the compiler. The main drawback in using directives is that the compiler can ignore them.

2.3.1 Unroll

The unrolling transformation duplicates N times the loop body and adapts the iterator with the right value at the right iteration at each copy. The goal of loop unrolling is to increase a program speed by reducing or eliminating instructions that control the loop and by facilitating vectorization.
The directive for this transformation is: !DIR$ MAQAO UNROLL=value.
Example below illustrates the unroll transformation applied on the inner loop in the nest :

```
do j = 1, x
 !DIR$ MAQAO UNROLL=4
 do i = 1, N, 1
  A[j] = b[i] + c[j]
 end do
end do
```

(a) Before the unroll transformation

```
do j = 1, x
 do i =1, N-3, 4
  A[j] = b[i] + c[j]
  A[j] = b[i+1] + c[j]
  A[j] = b[i+2] + c[j]
  A[j] = b[i+3] + c[j]
 end do
end do
```

(b) After the unroll transformation

2.3.2 Full Unroll

The transformation of full unrolling is similar to the unroll but the loop is replaced by the unrolled body. It goes further than casual unrolling by removing all instructions that control the loop.
The directive for this transformation is: !DIR$ MAQAO FULLUNROLL[=N]. An unroll factor can be defined if bounds have not been fixed.
The example below illustrates the full unroll applied on the inner loop of the nest, the loop is removed and replaced by its unrolled body :

```
do j = 1, x
 !DIR$ MAQAO FULLUNROLL
 do i = 1, 7
  A[j] = b[i] + c[j]
 end do
end do
```

(a) Before the full unroll transformation

```
do j = 1, x
  A[j] = b[1] + c[j]
  A[j] = b[2] + c[j]
  A[j] = b[3] + c[j]
  A[j] = b[4] + c[j]
  A[j] = b[5] + c[j]
  A[j] = b[6] + c[j]
  A[j] = b[7] + c[j]
end do
```

(b) After the full unroll transformation

2.3.3 Tile

Tiling (or Blocking) consists in dividing an iteration space into tiles and in transforming the loop nest to iterate over them. Loop tiling in a multi-level loop nest is done to change the spacial and temporal locality of data in the arrays, improving data reuse for some computation patterns. It can also be used to statically fix the size of the inner most loop and help the compiler to do a better optimizing job. With Fortran, we use the standard function "MIN" as shown in the example.
The directive for this transformation is: !DIR$ MAQAO TILE=N, when n is the value of the tile.
The example below illustrates the tile transformation of a whole loop nest :

```
!DIR$ MAQAO TILE=8
do x = 1, N, 1
 do y = 1, M, 1
  <loop body>
 end do
end do
```

(a) Before tile transformation

```
do lt_var_x = 1, N, 8
 do x = lt_var_x, min(N,lt_var_x+7), 1
  do lt_var_y = 1, M, 8
   do y = lt_var_y, min(M,lt_var_y+7), 1
    <loop body>
   end do
  end do
 end do
end do
```

(b) After tiling transformation

2.3.4 Strip Mine

A variant of the tiling is the strip mining. It is a method to adjust the granularity of an operation. The strip mining transforms a singly nested loop (the inner) into a doubly nested one.
The directive for this transformation is: !DIR$ MAQAO TILE_INNER=N, when n is the value of the strip. The example below illustrates the strip mine transformation of a whole loop nest :

```
!DIR$ MAQAO TILE_INNER=8
do t = 1, N, 1
 do k = 1, M, 1
  <loop body>
 end do
end do
```

(a) Before Strip Mine

```
lt_bound_M = (M /8)*8
do lt_var_k = 1, lt_boutnd_M, 8
 do t = 1, N, 1
  do y = lt_var_k, lt_var_k+7, 1
   <loop body>
  end do
 end do
end do
if (lt_bound_M < M) then
 do t = 1, N, 1
  do y = lt_bound_M+1, M, 1
   <loop body>
  end do
 end do
end if
```

(b) After Strip Mine

2.3.5 Interchange

Loop interchange permutes two loops. These loops can be consecutive or not, but they must not have any statement between them. Loop interchange is a code transformation which can induce a change in the spacial and temporal locality of memory elements by moving arrays column-major to raw-major or the opposite.
The directive for this transformation is: !DIR$ MAQAO INTERCHANGE[N,M], where N and M are the depth of loops to permute, 1 corresponding to the loop where the directive is attached.
The example below illustrates the interchange transformation of a whole loop nest :

```
!DIR$ MAQAO INTERCHANGE
do t = 1, N, 1
 do k = 1, M, 1
  <loop body>
 end do
end do
```

(a) Before interchange transformation

```
do k = 1, M, 1
 do t = 1, N, 1
  <loop body>
 end do
end do
```

(b) After interchange transformation

2.3.6 Loop Count Transformation (LCT)

Knowing the low bounds and more precisely the loop iteration counts enables to perform very efficient specialization based on that information. This can be exploited in many different ways including through compiler directives. Intel compilers offer the ability to specify a loop count (min, max, avg) directive, and can then make that information available to its optimization passes. By default the compiler will generally generate multiple variants (i.e. scalar, SSE, AVX, etc.) of the same source loop at binary-level. However, it will generate much fewer variants and will adapt its optimizations by considering loop count data. Helping the compiler in this way throughout the whole application can provide a significant performance gain (see section 4). This information is obtained with VPROF which returns the minimum, maximum and average number of iterations of a loop, then ASSIST uses this information either for specialization or for LCT.
The example below illustrates the loop count transformation applied on a single loop :

```
for (int i=0 ; i < x; i++ ) {
  ..
}
```

(a) Before Loop Count Transformation

```
#pragma loop_count min=100,max=100,avg=100
for (i = 0; i < x; i++) {
  ...
}
```

(b) After Loop Count Transformation

In the example, we detect that the loop only performs one hundred iterations; the loop count directive is inserted above the loop to indicate to the Intel compiler that it must adapt its choice according to that information. Generally, for a small number of iterations, it disables vectorization and unrolls the loop without peel/tails loops.

2.3.7 Short Vector Transformation (SVT)

We noticed that even when loop bounds were hard-coded the compiler would not vectorize that loop properly. We have been able to detect such cases using MAQAO CQA which analyses loops and computes vectorization efficiency metrics. For such cases, we developed a specific transformation (SVT) performing the following steps: force the compiler to vectorize the loop using the SIMD directive, prevent peeling code from being generated using the vector unaligned directive, and adapt the number of iterations to the vector length.
The directive for this transformation is: !DIR$ MAQAO SHORTVEC[=val], where "val" can be AVX2 (by default) or SSE.
The example below illustrates this transformation on a single loop :

```
double *a, *b;
...
#pragma MAQAO SHORTVEC=AVX2
for (int i=0 ; i < 7; i++ ) {
  a[i] += b[i]
}
```

(a) Before short vector optimization

```
double *a, *b;
...
#pragma simd
#pragma vector unaligned
for (i = 0; i < 4; i++) {
  a[i] += b[i]
}
#pragma simd
#pragma vector unaligned
for (i = 4; i < 6; i++) {
  a[i] += b[i]
}
a[6] += b[6]
```

(b) After short vector optimization

If the bounds are not known, a generic version can be used. It will create multiple versions of the SVT depending of the rest of the modulo of N.
The directive for this transformation is: !DIR$ MAQAO GENSHORTVEC[%N], where N is a small number (4 by default) which will be part of the modulo test. N must be small because it adds a lot of tests that can cost more than the gain obtained.
Example below illustrates the generic short vector optimization applied on a single loop :

```
double *a, *b;
...
#pragma MAQAO GENSHORTVEC%4
for (int i=0 ; i < N; i++ ) {
  a[i] += b[i]
}
```

(a) Before generic short vector optimization

```
double *a, *b;
...
if ((N%4) == 0) {
 #pragma simd
 #pragma vector unaligned
 for (i = 0; i < N; i++) {
  a[i] += b[i]
 }
} else {
 if ((N%4) == 1) {
  #pragma simd
  #pragma vector unaligned
  for (i = 0; i < N-1; i++) {
   a[i] += b[i]
  }
  a[N-1] += b[N-1]
 } else {
  if ((N%4) == 2) {
   ...
  } else {
   if ((N%4) == 3) {
    ...
   } else {
    for (int i=0 ; i < N; i++ ) {
     a[i] += b[i]
    }
   }
  }
 }
}
```

(b) After generic short vector optimization

Prefetcher	Description
L2 hardware prefetcher	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	Fetches the cache line that comprises a cache line pair (128 bytes)
DCU prefetcher	Fetches the next cache line into L1-D cache
DCU IP prefetcher	Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines

Table 2.1: The four types of hardware prefetchers for data prefetching. Source : https://software.intel.com

2.3.8 Prefetcher

By modifying the Model Specific Register (MSR), we can modify the prefetchers behavior. This register is available on every core controls but only available on the following micro-architecture: Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, and Broadwell. The impact of modifying prefetcher has already been discussed several times, for example, Lee [62] proposes a scheme for prefetching and also tries to determine if modifying prefetcher has an impact on PGO; Porterfield [93] evaluates several cache-line hardware prefetching schemes; Liaou et al. [64] proposed to improve modify prefetchers configuration. Most of researches on prefetchers have been done at application level to evaluate which prefetcher allows performance gains but too few have been undertaken at function level. However, prefetchers can be more or less efficient according to different loops parameters such as: iteration step, type of loop (i.e. computation loop, gather/scatter loop, ...), etc.

There are four types of hardware prefetchers for data prefetching, two associated with L1-data cache (DCU) and two associated with L2 cache (see table 2.1). These four hardware prefetchers can be controlled by calling the code in Appendix B.1 which will set the MSR to a value between 0 and 16 to enable/disable prefetchers. For this transformation, ASSIST inserts function calls at the beginning of functions to modify the prefetchers behavior for the whole function. The called function only takes an integer which represents one of the sixteen possible values as an input. However, we do not know how to choose the right value;

a possible way is to test every sixteen values and keep the best for each function but it is a long iterative process. This transformation has never been tested on real world application, only on NR.
Like other transformations, it is possible to trigger it using the following directive: !DIR$ MAQAO PREFETCH=<val>, where "val" is an hexadecimal number which represents the combination of prefetchers that have to be enabled or disabled. 0 means that all prefetchers are enabled.

2.3.9 Constant Propagation

Constant propagation is the process of substituting the values of known constants in expressions at compilation time. This transformation is used during the specialization transformation to set a variable to a specific value. Constant propagation can also cause conditional branches to be reduced to one or more unconditional statements, to determine the only possible outcome. This happens when the conditional expression is evaluated as true or false at compilation time.

2.3.10 Local Dead Code Elimination

Since we can apply multiple transformations, it might be necessary to clean up the transformed code in order to eliminate useless chunks generated by specialization (i.e. for example, conditionals). The local dead code elimination is a transformation that can be applied on any basic block. All statements of the sub-AST of the basic block are browsed to check all conditional branches. If a loop is detected with only one iteration, the loop is replaced by its body and the iteration variable replaced by its value in the whole body. ASSIST also checks the ”if” statements, by checking if the conditional expression is always true or false to replace the whole ”if” statement by its ”then” or by its ”else”. To check if a conditional expression is always true or false, the expression is statically evaluated. If it is composed of two integers, we compare them with the corresponding operator. If it implies a variable, ASSIST tries to trace it back, through previous assignment statements involving this variable; the aim is to check if it ends up as a constant and if this assignment is not the result of an ”if” condition or a loop. If all of the conditions are true, the variable will be considered as the right value and the test continues.

2.3.11 Specialization

Specialization is the process of creating particular versions of the same code by explicitly considering specific values of one or more variables. Specialization can be a very efficient transformation making it easy to trigger other transformations (e.g. a loop will be optimized differently depending on a high or low iteration count). Specialization is not obviously an end in itself, but can be a means to make optimizations applicable. It is used, when possible, to simplify in some way a portion of code based on the knowledge of one or multiple values and of their occurrences. As a consequence, the main drawback of specialization is that it can worsen performance if not used sparingly. To perform either loop or function value profiling, we rely on MAQAO VPROF. Our specialization transformations can be categorized into two classes: loop specialization and function specialization.

Loop

The loop specialization will duplicate the loop, propagate constants, apply the local dead code elimination on the specialized version and add an "if" statement to verify the value of specialized variables. The original loop is kept and used when the condition is wrong. The efficiency of the specialization has already been demonstrated several times [8, 74, 26]. Usually, specialization is mostly used to trigger other transformations for a particular value. The directive for this transformation is: !DIR$ MAQAO SPECIALIZATION (var [=|<|>|{N,M}] val [, ...]), where "var" is the name of the variable to specialize, an operand "=" to fix the value, "<" and ">" to indicate that the variable will never be over or under a value and "{N,M}" to indicate that "var" will still be between "N" and "M"; multiple variables can be defined.
The example below illustrates the specialization transformation of a single loop :

```
!DIR$ MAQAO SPECIALIZE(N=4)
do i = 1, N
 a[j] = b[i] + c[j]
end do
```

(a) Before the loop specialization transformation

```
if (N .eq. 4) then
 do i = 1, 4
  a[j] = b[i] + c[j]
 end do
else
 do i = 1, N
  a[j] = b[i] + c[j]
 end do
end if
```

(b) After the loop specialization transformation

Function

In the case of function specialization we will usually want to target specific value combinations. A new specialized function is created and the proper conditional statement is generated. To simplify the specialized code, an in-house version of the previously presented local dead code elimination pass is applied. It is possible to apply as many specialization directives as condition combinations. The current implementation specialization is limited to only integer variables. The three main issues of an efficient specialization is to know which variables are interesting to specialize, what are their values and how to minimize the number of variables to specialize. This allows to to be as generic as possible, while having the best results.

Like loop specialization, the directive is: "!DIR$ MAQAO SPECIALIZATION (var [=|<|>|={N,M}] val [, ...])", where "var" is the name of the variable to specialize, an operand "=" to fix the value, "<" and ">" to indicate that the variable will never be over or under a value and "{N,M}" to indicate that "var" will still be between "N" and "M"; multiple variables can be defined.
The next example illustrates the specialization transformation of a function.

```
#pragma MAQAO SPECIALIZATION(N=4, c={1,10}
void foo (int *a, int *b, int N, int c) {
 if (c == 0) {
   for (int i=0; i < N; i++) {
     a[i] = 0;
   }
 } else if ( c <= 10) {
   for (int i = 0; i < N; i++) {
     a[i] += b[i];
  }
 } else if (c > 10) {
    for (int i = 0; i < N; i++) {
      a[i] -= b[i];
    }
 }
}
```

(a) Before the function specialization transformation including constant propagation and local dead code elimination

```
void foo (int *a, int *b, int N, int c) {
 if ((N==4) && (c >= 1) && (c <=10)) {
    return foo_Ne4_cb1_10(a,b,c);
 } else {
  if (c == 0) {
   for (int i=0; i < N; i++) {
     a[i] = 0;
   }
  } else if ( c <= 10) {
   for (int i = 0; i < N; i++) {
     a[i] += b[i];
   }
  } else if (c > 10) {
    for (int i = 0; i < N; i++) {
      a[i] -= b[i];
    }
  }
 }
}
...
void foo_Ne4_c1_10 (int *a, int *b, int c) {
 for (int i = 0; i < 4; i++) {
   a[i] += b[i];
 }
}
```

(b) After the function specialization transformation including constant propagation and local dead code elimination

2.4 Assessing Transformation Verification

"Between what I think, what I want to say, what I think I say, what I say, what you want to hear, what you hear, what you think you understand, what you want to understand, what you really understand, there are nine possibilities to not understand each other but let's try anyway"
(B. Werber, The Encyclopedia of Absolute and Relative Knowledge).

This citation refers to the communication between humans, but it is the same between our transformations and the compiler. Between what we think could be done, what we can do, what the compiler understands and what it does with it, there are multiple chances that our optimizations have no effect or worse degrade performances. We saw how to combine analyzers and profilers to trigger ASSIST transformations but the process can be done the other way around to verify the legality of these transformation afterwards. We minimize this issue using the knowledge gathered in MAQAO over times. This allows us to know what transformation can be applied to increase performance speedup. But we want a way to prevent the possibility that our transformations could degrade performances. This problematic has been part of an internship performed by Claire Baskevicth under my supervision. The goal was to add a new automated part in ASSIST to gather information and metrics collected by different MAQAO modules, in order to have a screenshot before transformations. Then we have to execute ASSIST with the gathered metrics to apply transformations; finally, we have to execute again some MAQAO modules on one hand to have the information whether or not we degrade performances and on the other hand, to know if the issue is still remaining or if new issues have appeared for which we can apply a new transformation to gain even more in performances. This allows us to have an iterative optimizer where the user can interact and know what exactly is done at each step.

How It Works

Step 1

It starts by loading the configuration file (see Appendix A.2 for more information about configuration file), before to set up the environment. The source code root directory is renamed to have a current version (N) of source code and the next version (N+1) after ASSIST transformations. Names of those directories have this syntax : <src_dir_name>_Vn, where n is the version number. For the first run, the original version starts at 0. Then, the program executes ONEVIEW on the Nth version. If an error occurs during this phase, the N+1 version directory will not be created.

Step 2

During this phase, information about loops, such as source file name, id, start and end lines, provided by the analysis, are stored in a table. Then labels are added in the Nth version of source code to later recognize loops that will be compared. Subsequently, ASSIST performs transformations on the Nth version of the source code using a file generated by the ONEVIEW analysis. Transformed files are stored in the N+1 directory. Finally, files are compiled again and ONEVIEW is executed again on this N+1 version.

Step 3

Finally, information about loops from the N+1 version (CQA, DECAN and VPROF metrics) are used to fill in the structure and only loops selected in phase 2 are kept. Metrics before and after transformations are compared and printed. If a loop is not detected after the second ONEVIEW execution, then an error message is printed and only metrics of the loop before transformations are presented.

Compared Metrics

Multiple metrics are used to compare loops between two executions and are separated into two mains parts. First, the ONEVIEW global metrics gather information on the whole application, see Appendix A.3 for more information about each metric.

The second part of metrics comes from other MAQAO modules and especially from the static analyzer CQA. We try to gather all important metrics to help the user at most, but the CQA qualities are also its weakness. Due to its static aspect, CQA considers that all data fit in L1 and iteration loops are infinite. So some of the metrics can be misleading, we will see in the following "Use Case Example" section, that after specializing and tiling a loop, CQA indicates that on almost all metrics, the transformation degrades performances, only the flops per cycle is better.
CQA compared metrics principally concern static information about specific loops such as vectorization ratio, and floating operation per cycle, etc, see Appendix A.3 for more information.

A static analysis allows not to execute the program again but the user can decide to execute dynamic profilers to have more information. If VPROF is executed, we compare the number of iterations, minimum, maximum and average of loops. If DECAN is executed, we compare the minimum, maximum and mean ratio of L1 on Original.

```
        GLOBAL METRICS                          BEFORE TRANSFO      AFTER TRANSFO       IS BETTER ?

        Total Time (s) :                           2.34                 1
        Flow Complexity :                          1                    1               lower is better
        Array Access Efficiency (%) :              71.95                54.66           higher is better

        Speedup if clean :                         1                    1.04            lower is better
        Nb loops to get 80% if clean :             1                    1               lower is better

        Speedup if FP Vectorised :                 1                    1.47            lower is better
        Nb loops to get 80% if FP vectorized :     1                    1               lower is better

        Speedup if Fully Vectorised :              1.66                 1.87            lower is better
        Nb loops to get 80% if Fully vectorized :  3                    2               lower is better

        Speedup if data in L1 Cache :              1                    1.18            lower is better
        Nb loops to get 80% if data in L1 Cache :  0                    2               lower is better

        Compilation Options :                      OK                   OK
        ------------------------------------------------------------------------------

opernlb_ylm_pp.f90 :

        METRICS                                 BEFORE TRANSFO      AFTER TRANSFO       IS BETTER ?

        Loop lines :                            314 - 314               881-884

        Loop_id :                               74                      246

        Speedup if clean :                      1.00                    1.06            lower is better
        Speedup if fully vectorized :           1.62                    2.06            lower is better

        Bottlenecks :                           p5                      P0,01
        Unroll confidence level :               max                     max
        Cycles L1 if clean :                    13                      8.00            higher is better
        Cycles L1 if fully vec :                8                       2.06            higher is better
        Vector-efficiency ratio all :           85.71                   42.00           higher is better
        Vectorization ratio all :               67.86                   68.00           higher is better
        FP op per cycle L1 :                    2.46,                   3.76            higher is better

        ------------------------------------------------------------------------------
```

Figure 2.4: Example of comparison before and after transformations using ABINIT with the test case Ti-256.

Use Case Example

Figure 2.4 presents an example of comparison before and after the specialization and the tiling of the test case Ti-256 from ABinit. In this example we can see that our transformations allow to gain 1,34 seconds, however, almost all CQA metrics say that transformations degrade the generated code except the metric flops per cycle which passed from 2.46 floating point operation per cycle to 3.76. This example perfectly highlights CQA weaknesses, all metrics are not useful for any transformation. This implies that we have to know what to look for each transformation. It remains a static analysis so it cannot be perfectly accurate about the fact that the tile transformation can gain or not, however, some metrics are good clues to guide us without executing the whole program.

Limitations

As we just saw, trying to compare a code before and after transformation can have some limitation. It can be caused by a missing loop after a transformation, by full unrolling for example; here after the loop will not be found during the last step of the process and no comparison can be made. It will be impossible to know if the transformation had a positive impact or not. ONEVIEW only analyses loops, and moreover innermost ones, so if a loop disappeared after transformations then the second execution of ONEVIEW will not find it. A solution was found, however, there are some disadvantage making it impossible to setup for the moment. The idea is to detect, after transformations, if a loop is full unrolled. If it is the case, the loop just above in the loop nest is taken. However, comparing two different loops will give biased results. The solution should be to execute again the first run of ONEVIEW and select the loop above of the unrolled one. Nevertheless, two problems occur : the first one is that ONEVIEW only analyses inner loops and the second is that the time of execution will increase considerably. ONEVIEW based its analysis on debug information given by the compiler and can lack of accuracy (loop line start/stop); if the accuracy exceeds more or less three lines, the loop will not be found.

The second limitation is that to really know if a transformation obtained a performance gain, we need to execute it. The static analysis can only give hints but it will never be as accurate as the dynamic analysis and it is impossible to quantify the gain after transformation without testing it.

2.5 Conclusion

In this chapter we introduce ASSIST, a semi-automatic tool which can perform source-to-source transformations guided by performance analysis metrics and open to user advices. ASSIST is based on the Rose compiler and has been integrated into the MAQAO tool suite. It is an FDO tool that use MAQAO performance analysis tools to drive a set a well-known transformation. Analysis gather static and dynamic analysis tools such as CQA, VPROF and DECAN to produce quality low-level measurements and insights. CQA gives a first view of the quality of the code in term of vectorization. VPROF provides the number of iteration of each loop. DECAN pinpoints precisely the instruction responsible for a performance anomaly. How transformations are triggered using metrics from these tools is detailed in chapter 3 and results obtained with this method are presented in chapter 4. ASSIST also provides with an assessing transformation verification system that tries to determine if a transformation has been counterproductive. This method has its flaws and it is up to the user to check if the transformation must be kept.

Chapter 3

What Triggers Transformations and How

3.1 Introduction

In this chapter, we highlight what triggers transformations and how. As a reference, we present two well-known and widely used FDO tools, Intel compiler PGO and AutoFDO. They have been developed for many years and offer good performance gains while having two different approaches. We compare them with our tool.

FDO tools follow the same process but they have multiple ways to perform each step, especially during the optimization. First, transformations can be explicitly applied in the source code. The main drawback of this solution is that we cannot be sure that the compiler will not add any optimization or will understand what is expecting to be done. Second, directives can be inserted in the source code to drive compiler optimization choices. Most of Intel directives to perform optimization are listed in Appendix B.2. Directives are limited in number and often related to only one compiler. But even this compiler can ignore them. A last possibility is to communicate with the compiler using a plug-in or an interface. This solution is compiler dependent and does not guarantee that the compiler will not modify the code afterward.

3.2 Collected Data and Triggered Transformations

3.2.1 Compilers PGOs

The PGO has been implemented in the three best known compilers, LLVM, GNU and Intel. It shows that this optimization method has the potential to significantly improve performances. Among these compilers PGOs, the Intel one is the most advanced. We have too few information about the GNU PGO and it does not seem to be a priority for GNU developpers. The LLVM PGO is too recent and even if a lot of efforts have been done, more are still

required to reach the level of Intel.

Intel PGO is already included in the Intel Compiler, it uses all resources provided by the compiler and overwrites the static cost model by the dynamic analysis to drive the transformations. One of the main drawback is that the instrumentation step must be reiterated each time the source code is modified. Another one is that it is a full blackbox. It is really difficult to know what the PGO sends to the compiler and what the compiler performs after receiving these data. According to Intel [82], their compiler PGO can perform multiple transformations, such as: re-ordering code layout to reduce instruction-cache problems, shrinking code size, and reducing branch mispredictions. For the re-ordering code layout, for example, the Intel PGO analyzes the control flow and sorts the paths by hot and cold blocks before grouping them together. This is an example of hot/cold path management by the PGO.

```
Example:
  ...
  for (int i = 0; i < 1000; i++)
    for (int j = 0; j < 10; i++)
        foo (j);
  ...

  foo(int x) {
    if(x < 10)
      hot();
    else
      cold();
}
```

During the analysis step, the Intel PGO has detected that the function "foo" is widely called with a value smaller than ten. The whole function is considered as a hot block when the parameter "x" is smaller than ten and cold for other cases. According to these results, the compiler separates the blocks into two groups, respectively hot and cold.

The triggered transformations, thanks to the PGO profiling, provide the following benefits: the spill code location is optimized using the profile information to modify the register allocations; by identifying targets, PGO can improve the branch prediction of indirect function calls; it disables the vectorization of any loop that only executes a small number of iterations.

All FDO tools try to minimize the overhead implied by the profiling step. This minimization forces them to limit the quantity of data collected. Intel tries to overcome this limitation in the last version of their PGO. According to [116], the Intel PGO version 18 collects the hardware-based event sampling data, thus avoiding instrumentation which caused the overhead and an extra need of memory. With this feature the Intel compiler will be able to collect more data and perform more efficient transformations.

According to GCC optimization and instrumentation options [81, 80], the GNU Compiler PGO profiles arcs frequency and expression values. However, due to a lack of information about the GNU PGO, we do not know how it really works, if arcs frequency and expressions values are the only two things collected, and finally, what kind of transformations are really performed after the profiling.

According to LLVM [95], LLVM PGO can perform, among others transformations: block layout, spill placement, inlining heuristics, hot/cold partitioning, etc. Most of these transformations have already been included in the Intel PGO (i.e. hot/cold partitioning, etc). Moreover, part of Google compiler developers have stopped to work on LLVM PGO and they are now developing AutoFDO described in the next section.

3.2.2 AutoFDO

AutoFDO is a framework which collects feedback from production workload and applies it at compilation time. It samples hardware performance counters and uses data to create profiles (described in the next paragraph). Then, these profiles are used to compile the next release of the program. During this release, the compiler uses the profiles to annotate the compiler IR. Thus, these feedback will drive compiler optimizations. Generated profiles can be used with LLVM (version 3.5 and after) and a clone of GCC 4.8 (available on their website). This GCC clone has been modified by Google to support AutoFDO. To summarize, AutoFDO can be used as an alternative to traditional compiler PGOs profilers.

AutoFDO uses sampling to profile applications with less than 1% overhead while achieving 85% of the gains of traditional FDO. The binary profile has two maps. The first one is a map connecting binary instructions addresses to their frequency. The second one is a map connecting branches to their frequency. Both can be obtained using the CPU performance monitoring unit (PMU). The binary profile is then converted into a source profile with more accurate information about the location and allows to extend these information. For example, it can extend the frequency data collected for a statement to all other statements from the

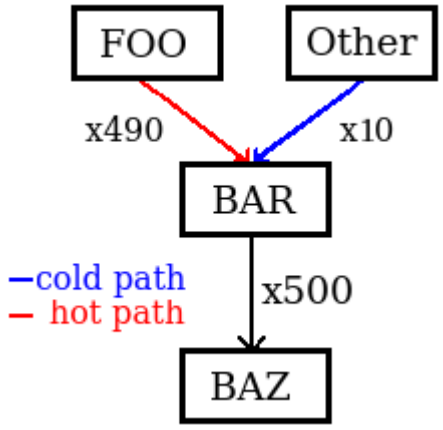


Figure 3.1: AutoFDO call graph to detect hot paths throught function calls.

same basic block. Finally, the source profile is used to annotate the IR. When the IR is annotated, it not only marks the edge frequency, but it also adds the context. For example, AutoFDO can say that a function "baz" is called 500 times by "bar" but it also precises that "bar" mainly calls "baz" when called by the function "foo". The edge of frequency is hot when "foo" call "bar" which call "baz". With this analysis, "baz" will be inlined through "bar" into "foo" but not into other functions. Figure 3.1 represents this example. AutoFDO forwards information to compilers (GCC or LLVM) which then has to perform the optimizations according to these data. For LLVM, AutoFDO annotates the IR and thus triggers the PGO transformations as if the PGO itself had profiled the program. For the GNU compiler, AutoFDO has to use the "gcov" format [38] to provide its information and trigger GNU PGO transformations as if the GNU PGO itself had profiled the program.

3.3 ASSIST Transformations to Trigger

Most optimization tools are data driven. They have a lot of data and then, they search the right transformation to apply according to these data. On the opposite, ASSIST is transformation driven. It means that we have several transformations available in ASSIST and we want to find out what information could trigger them. This section presents how we can trigger ASSIST transformations, described in section 2.3, using MAQAO performance analysis tools.

3.3.1 Loop count

The loop count transformation is a little bit different from the other transformations. It does not modify directly the source code but inserts directives that will drive compiler choices by giving him hints about loops trip count. Each directive has three parameters representing the minimum, maximum and average expected numbers of iterations. We trigger this transformation when any other transformation has been performed and if the loop trip count information is available. This information is obtained by using the MAQAO value profiler, VProf. It profiles each loop and records among other things, their minimum, maximum and average numbers of iterations.

The loop count transformation is an example of what we can obtain by using compiler directives. As said at the beginning of this chapter, there are multiple other directives to perform different optimizations. These directives are presented in Appendix B.2. Currently, even if ASSIST only handles the loop count directive nowadays, it could also manage the other directives in the future. Moreover, some of ASSIST transformations could be performed by using these directives. Among these directives, some could be easily triggered using MAQAO modules metrics. The directive "FMA / NOFMA" could be triggered when CQA detects that FMA instructions are not optimally used. ASSIST could disable the compiler unrolling using the directive "NOUNROLL" when CQA highlights unrolling and/or peels/tails issues. Vectorization of loops could be forced by inserting "SIMD" or "VECTOR" directives. These directives could be triggered when CQA detects a bad vectorization. And "NOVECTOR" could be used on loops fully vectorized while having too few iterations to have optimal performance. "PREFETCH / NOPREFETCH" has nothing to do with our prefetch transformation. The directive aims at giving the compiler a hint to prefetch data from memory. The directive could be inserted when DECAN shows there would be a potential speedup if data were in L1 cache. But the speedup would not ne sufficient to trigger the tiling transformation.

3.3.2 Unroll & fullunroll

The unroll transformation as well as the fullunroll one allow to help the compiler to vectorize a loop. CQA can analyze whether a loop has already been unrolled or not and determine its unroll factor. If the loop has not been unrolled, CQA can compute the potential gain of a possible unroll. However, the biggest challenge in this transformation is the difficulty to know the right unroll factor. For this, we currently have no other solutions that to provide multiple versions of the code with different unroll factors: 2, 4, 8, 16 and 32. Then, we still have to choose the most efficient version.

3.3.3 Interchange

The interchange transformation allows improving arrays accesses. Currently, we do not have any metric that could trigger this transformation. A potential way to do so would be, for example, to check how many are not stride-one arrays in a loop nest. We could detect that either at source-level or at assembly-level. Indeed, we could first, check arrays accesses and loops iterators. Then, we could compare how many stride-one accesses are currently done and how many will be, if we interchange the loops order. It is easy to verify on simple cases but it can become harder on a complex loop nest; we can also miss information if the loop to handle is in a inlined function.

3.3.4 Tile & strip mine

The tile and the strip mine transformations allow to improve the data locality. These transformations not only require to know how many cache misses are done in a loop but also the cost of a cache miss and where data are before the loop starts. One variant of the MAQAO module DECAN is named DL1. It simulates that all data access are made from L1 cache. It then compares both executions to know the performance gain potential for a specific dataset and thus, determine the performance improvement, had the data be in L1 cache. Tiling and strip mining are interesting transformations in term of performance gain. However, even if we know when to trigger these transformations thanks to the DL1 metric, a metric is missing to determine the tile size. Another point is when DECAN detects a good potential speedup for a loop containing multiple arrays. In that case, it could be interesting to verify if the speedup comes from one of the arrays and if it can be restructured to improve performance speedup.

3.3.5 Prefetcher

As reminder of the section 2.3, the activity prefetcher aims at being controlled. There are four prefetchers which can be separately turned on or off. The main difficulty is to find the right combination. There is no existing metrics that can drive us to choose which behavior to apply and when. The current solution is to generate sixteen versions of the code for each possible behavior, analyze which behavior has the best results for each function and then, modify the prefetcher behavior of each function with the best version. This transformation could only be applied on functions with a high coverage ratio. For functions with a small coverage, changing the prefetcher behavior will not have a sufficient impact. The main drawback is that we must test all behavior combinations to determine which one allows to obtain the best results on which functions. Modify prefetchers behavior at function level can also

cause possible side effects on rest of the code.

When the prefetcher transformation is applied, the prefetchers configuration is modified at function-level. ASSIST inserts a call at the beginning of a target function. The code performed by the inserted call is detailed at Appendix B.1. This code is provided by MSR-Tools [72]. It is an open-source project with Intel as major contributor. The code allows to read and write MSR from/to any CPU or all CPUs by modifying a value in a file located at "/dev/cpu/<#cpu>/msr", where "<#cpu>" refers to a cpu id.

A drawback appeared when the MSR is modified on the fly at function level. The execution time is multiplied by ten and several system calls have been added representing more than 80% of this time. This drawback can come from memory accesses. The MSRs have to be modified at the same time on all CPUs and it is a locked memory address, so all accesses are concurent and most of the time is spent in function like "_spin_lock", "sysret_check", "generic_exec_single", etc. More results are available in chapter 4. These results show the impact of turning on or off the prefetchers on multiple real-world applications at function level. However, for these results, the prefetcher behavior has been changed before to execute each application. Second drawback is that the register to modify prefetchers behavior is only available on Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, and Broadwell. On more recent architectures, the register is not available, it seems it is doomed to disappear.

3.3.6 Specialization

The specialization is a wide world with a plethora of possible applications. Ours only are a sub part of what is existing. It allows us to simplify the control flow (by removing branches), to set the number of iterations of a loop. It even allows to trigger another transformation (i.e. Short Vectorization). To trigger our specializations, we use the MAQAO value profiler, VProf. If we want to specialize a loop we can use the trip count data, like in the case of the loop count transformation. Or, ASSIST can also detect which variables could be interesting to specialize and ask to profile these variables with VProf. The next example presents one case of specialization of particular variables.

Example of Automatic Specialization

One of the most representative example of combining profilers and a well-known transformation is our automatic specialization. In this case we use VPROF, a MAQAO module which performs value profiling, to detect most used values of variables detected as interesting to specialize. The main goal here is to address issues of an efficient specialization while

having the best results. We implemented an automatic specialization that combines static and dynamic analysis. Our approach is composed of three steps:

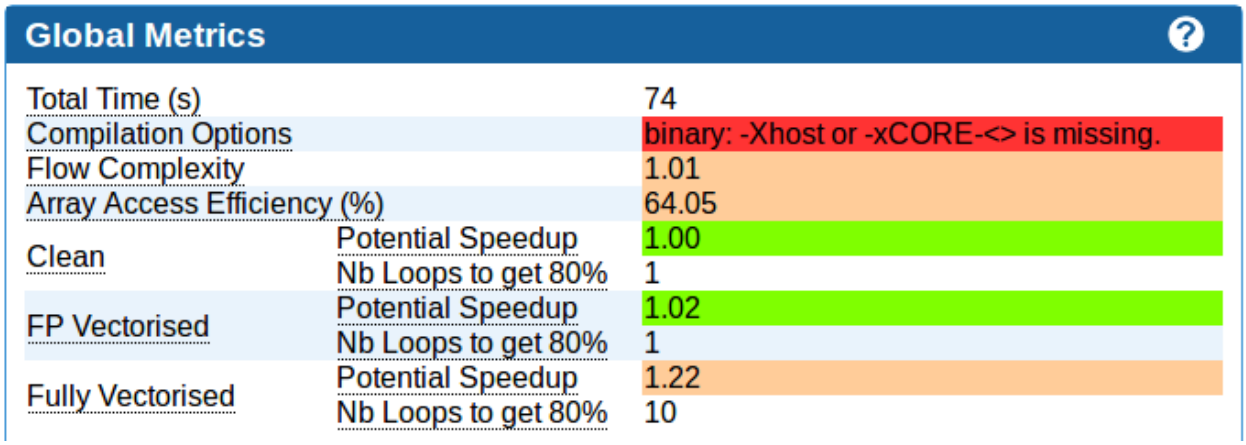
1. The static analysis: ASSIST looks for variables to specialize browsing the AST. This step presents two distinct cases:

   - either ASSIST focuses on functions (calls to the VPROF library are inserted at the beginning of functions to analyze their integer parameters),
   - or ASSIST focuses on the loop nests (trying to evaluate what variables are interesting to specialize). In this case, ASSIST starts by looking inside the innermost loop searching for variables used for the loop bounds, and these variables are added into a list. Then, backtrack search is performed in the loop nest to check if one variable of the list has not been previously computed. For an "if" statement, we also want to know if its "then" or "else" path is unused for a set of variables and values in order to remove these paths or the whole statement if it only contains an unused "then" path. To get these information, a call to the VPROF library is inserted at the beginning of each path which records the number of times a path is taken. We also add all variables contained into the "if" condition to the list and continue backwards in the loop nest. If a variable from the list is computed, then it is removed from the list, and all variables on the right hand side of the assignment are added to the list.To minimize our problem, only innermost loops are profiled. When the loop nest is browsed and the list of variables to specialize is completed, a call to VPROF library is added before the loop nest for each variable to profile. A unique label is generated and added above the loop to store information concerning which variables to specialize and their location.

   In the end of this first step, ASSIST creates two source files. The first one instrumented with calls to the VPROF library, and the second one instrumented with labels.

2. Instrumented files are compiled and the code is executed; then all data are collected in a file as a Lua table which will be provided to ASSIST in the third step.

3. Specialization is triggered on labeled files using results from the value profiling. To trigger the specialization, the distribution of the values of a variable must be inhomogeneous with a more widely used value.



Global Metrics		
Total Time (s)		74
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.01
Array Access Efficiency (%)		64.05
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.22
	Nb Loops to get 80%	10

Figure 3.2: Polaris - Metrics global metric before applied the SVT

3.3.7 Short vectorization

The short vectorization transformation allows to improve vectorization of a loop. This transformation is currently the only one which requires multiple metrics in addition of the user approval. To trigger the SVT, two metrics are requires. The first one is the loop trip count from VProf, to check that the loop only performs a few number of iterations. The second one is the vectorization ratio metric from CQA; it allows us to know if the loop has not been correctly vectorized. This transformation is very touchy and in addition of these metrics, the user have to be questioned about the legitimacy of the transformation. The next chapter presents the results obtained with the short vectorization transformation on a scientific application named POLARIS. The following example presents how we triggered the SVT as well as each step of the process that led us to these results.

Example of Short Vectorization Transformation

Figure 3.2, presents MAQAO global metrics obtained on Polaris on the original version. The flow complexity[1] is set to 1.01 and "Fully vectorize" [2] metric indicates that the application can obtain a speedup up to x1.22. These metrics also indicate that the vectorization bottleneck does not concern floating-points operations; the bottleneck must concern loads and stores. Figure 3.3 confirms this hypothesis, because as user, we know that the two hottest

[1]The Flow Complexity metric represents the average number of paths in loops for whole the application.
[2]The "Fully vectorize" metric indicates the potential speedup if the whole application was fully vectorized.

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 6916	16-18	polaris_orig_icc17:mom7.f90	pack2	6.28	3.11	0	1	1	4
Loop 6918	27-29	polaris_orig_icc17:mom7.f90	unpack2	5.17	2.56	0	1	1	4

Metric	Value
Loop id	6916
Coverage (% app. time)	6.28
Time (s)	3.11
[VPROF] Iteration Count (min)	60
[VPROF] Iteration Count (avg)	60
[VPROF] Iteration Count (max)	60

Metric	Value
Loop id	6918
Coverage (% app. time)	5.17
Time (s)	2.56
[VPROF] Iteration Count (min)	60
[VPROF] Iteration Count (avg)	60
[VPROF] Iteration Count (max)	60

Figure 3.3: Polaris - Metrics global and of the two hotspots loops before to apply the SVT.

loops, which each have 0% vectorization, are gather and scatter. Hence, the potential gain is low if floating-point operations have been fully vectorized. Moreover, the metrics of both loops, indicate that the number of iterations remains constant at sixty. The OneView report generated especially for ASSIST with the raw data is presented in Appendix A.8. These two loops gather all the conditions and we know that the SVT is legitimate on that kind of loop. We decide to apply the transformation. Due to the sixty iterations we applied the generic version with a 4 modulo.

Figure 3.4 presents the global metrics obtained after applying the SVT. Compared to figure 3.2 we have the following improvements. The application ran 3 seconds faster, the flow complexity has been improved to 1.00 and the vectorization potential speedup if the whole application had been vectorized decreased to x1.15. Moreover, figure 3.5 shows the improvement of the loops with vectorization ratio up to 100% for both and an execution time which dropped to 2.25 and 1.69 seconds against 3.11 and 2.56 seconds before transformation.

3.4 Conclusion

In conclusion, ASSIST has a set of transformations that can be triggered in different ways. First, the user can insert directives in his code source. When ASSIST analyze the code and find a specific directive it will apply the corresponding transformations without looking for if it is profitable or not. The second method is to use a transformation script. To avoid to add directives in the code, the user can provide a script with all information. Like the directives, all transformations can be trigger in this way and ASSIST will not check the potential gain of transformations. The script contains the same information as the directive

Global Metrics

Total Time (s)		71
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		62.80
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.15
	Nb Loops to get 80%	6

Figure 3.4: Polaris - Global metrics after the SVT has been applied.

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 6933	63-65	polaris_svt_icc17:mom7.f90	pack2	4.79	2.25	100	1	1	1
Loop 6951	139-141	polaris_svt_icc17:mom7.f90	unpack2	3.6	1.69	100	1	1	1

Metric	Value
Loop id	6933
Coverage (% app. time)	4.79
Time (s)	2.25
[VPROF] Iteration Count (min)	15
[VPROF] Iteration Count (avg)	15
[VPROF] Iteration Count (max)	15

Metric	Value
Loop id	6951
Coverage (% app. time)	3.6
Time (s)	1.69
[VPROF] Iteration Count (min)	15
[VPROF] Iteration Count (avg)	15
[VPROF] Iteration Count (max)	15

Figure 3.5: Polaris - Global metrics and specific metrics of the two hotspots loops after the SVT has been applied.

plus the file and lines of the statement to transform. The last way to trigger transformation is to use ASSIST as an FDO using MAQAO performance analysis tools metrics. Currently, only few transformations can be triggered using MAQAO metrics. First, the SVT which uses a combination of CQA vectorization ratio and VPROF min, max and average number of iteration of a loop. Second, the Tiling which uses the DECAN DL1 variant that estimate the gain if all data fit in L1. Third, the LCT which only uses the VPROF min, max and average number of iteration of a loop. Finally, the specialization can be also automatically trigger using a static analysis performed by ASSIST. This analyze tries to estimate which variable must be specialized. In the next chapter we will observe the impact of these transformations.

Chapter 4

Experiment

In this chapter the results of ASSIST are presented and compared with Intel compiler PGO mode denoted (IPGO) in the sequel. Intel compilers are neither open source nor free, but they provide the best performance in our tests (compared with GCC and LLVM). The second main reason behind this choice of PGO comparison lies in the lack of availability for FDO tools for regular users. As a reminder, for IPGO, the use of profiling data enables some specific optimizations but can also modify the behavior of other optimizations such as:

1. using feedback data on function entry counts. Function grouping is done to put hot/cold functions adjacent to one another;

2. value profiling of an indirect and virtual function calls. It is done to specialize indirect function call for a common target;

3. annotating the intermediate language with edge frequencies and block counts. They are then used to guide a lot of the optimization decisions made by other passes of the compiler, such as: the in-liner and partial in-liner, the basic block layout, the conversion from switch tables to "if" statements, loop transformations like unrolling, etc.

Our goal is not to "mimic" IPGO, but rather to present a complementary approach which goes beyond the observed limitations. All the measurements presented below have been gathered on an Intel(R) Skylake SP based machine (Intel Xeon Platinum 8170 CPU@2,10GHz) with Intel compiler version 17.0.4. Multiple executions (exactly 31) were performed to reach statistical stability and avoid outliers in data measurement.

In this chapter, the results of experimental transformations offered by ASSIST are presented. They are based on feed back data and user insight. This study is application centered, we have looked for an approach to get a good performance gain at minimal cost; starting from the specific needs of each application based on what MAQAO profilers return, we can trigger the right transformation that will answer these needs, thus limiting the search space

and avoiding to blindly test different useless transformations. In this chapter, a speedup is considered as a faster execution of the application.

4.1 Application Pool

Multiple fully industrial class applications were used to test our approach:

YALES2 [28] (version 0.5.0) is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. It is a finite volume code for unstructured meshes with an innovative 4th order spatial scheme for the discretization of convective and diffusive terms. It is based on the low-Mach number approximations of the Navier-Stokes equations which solves an elliptic Poisson equation at each iteration. The MPI version uses sub-domain decomposition with adjustable domain size allowing efficient cache usage. ASSIST has been tested on two of their datasets named "3D_Cylinder", a pure CFD computation, and "1D_COFFE", a combustion computation. The application is written in Fortran 2003 and approximately contains 276 000 lines of code.

AVBP [11] (version D7.0.1) is a parallel CFD code developed by CERFACS. It solves the three-dimensional compressible Navier Stokes equations on unstructured multi-element grids. It uses third space and time Taylor Galerkin numerical schemes. The code has been ported and tested in up to 200K cores with an 85% strong scaling efficiency (BG/Q) for a 200M element case (1000 elements per MPI rank). Cache coloring uses the reverse Cuthill-Mckee method. ASSIST has been tested on three representative datasets namely: SIMPLE (helicopter chamber demonstrator combustion simulation), NASA (NACA blade simulation) and TPF (large flow simulation). The application is written in Fortran 95 and approximately contains 275 000 lines of code.

ABINIT [2] (version 7.10.5) is a package allowing users to find the total energy charge density and the electronic structure of these systems made of electrons and nuclei (molecules and periodic solids) within Density Functional Theory (DFT) using pseudo-potentials (or PAW atomic data) and a planewave basis. The application approximately contains 807 000 lines of Fortran 90.

POLARIS (MD) (version 1.0.5.18) is the only code that can be used to perform microscopic simulations of high precision (especially for the treatment of interatomic interactions) for molecular systems of several millions atoms with "speed" of the order of one nanosecond a day. On this last point ("speed"), many improvements are still possible to consider "speed"

	AVBP NASA	AVBP TPF	AVBP SIMPLE	Yales2 3D Cylinder	Yales2 1D COFFEE
Number of loops	149	173	158	162	122

Table 4.1: Number of loops processed by ASSIST LCT for each application and test case.

of several nanoseconds per day. Finally, its hybrid parallelization scheme (OPENMP / MPI) makes it particularly well suited to the new generation of "many-core" systems.

Convolutional Neural Network (CNN) is a state-of-art DNN for image recognition. New CNN workloads emerged and are pushing the limits of today's hardware. One of the expensive kernels is a small convolution in such specific sizes that calculations in the frequency space are not the most efficient method when compared with direct convolutions. Training CNNs requires an enormous amount of time, making their optimization very critical. The CNN code refers to the one used in [70] and the layers used are the GoogleNet_V1. The convolution technique consists in executing all CNN layers one after the other with different sizes of filter (1x1, 3x3 and 5x5). This codelet is written in C and contains 450 lines of codes.

Mini QMCPACK is a simplified but computationally accurate implementation of the real space quantum Monte Carlo algorithms implemented in the full production QMCPACK [98] application. Mini QMCPACK and QMCPACK are available on github[1] [2]

4.2 Impact of Value Profiling

Our first FDO optimization uses loop trip counts information obtained by value profiling using MAQAO VPROF. When loops exhibit a complex control flow due to multi-versioning, the knowledge of the trip count can help the compiler simplify the decision tree.

Figure 4.1 presents the speedups obtained with LCT or IPGO and the combination of both for each application/dataset. For these applications, the combination of LCT and IPGO reaches a speed up of 14% for a sequential YALES2 run with the 3D Cylinder dataset. To

[1] Mini QMCPACK: https://github.com/QMCPACK/miniqmc
[2] QMCPACK: https://github.com/QMCPACK/qmcpack

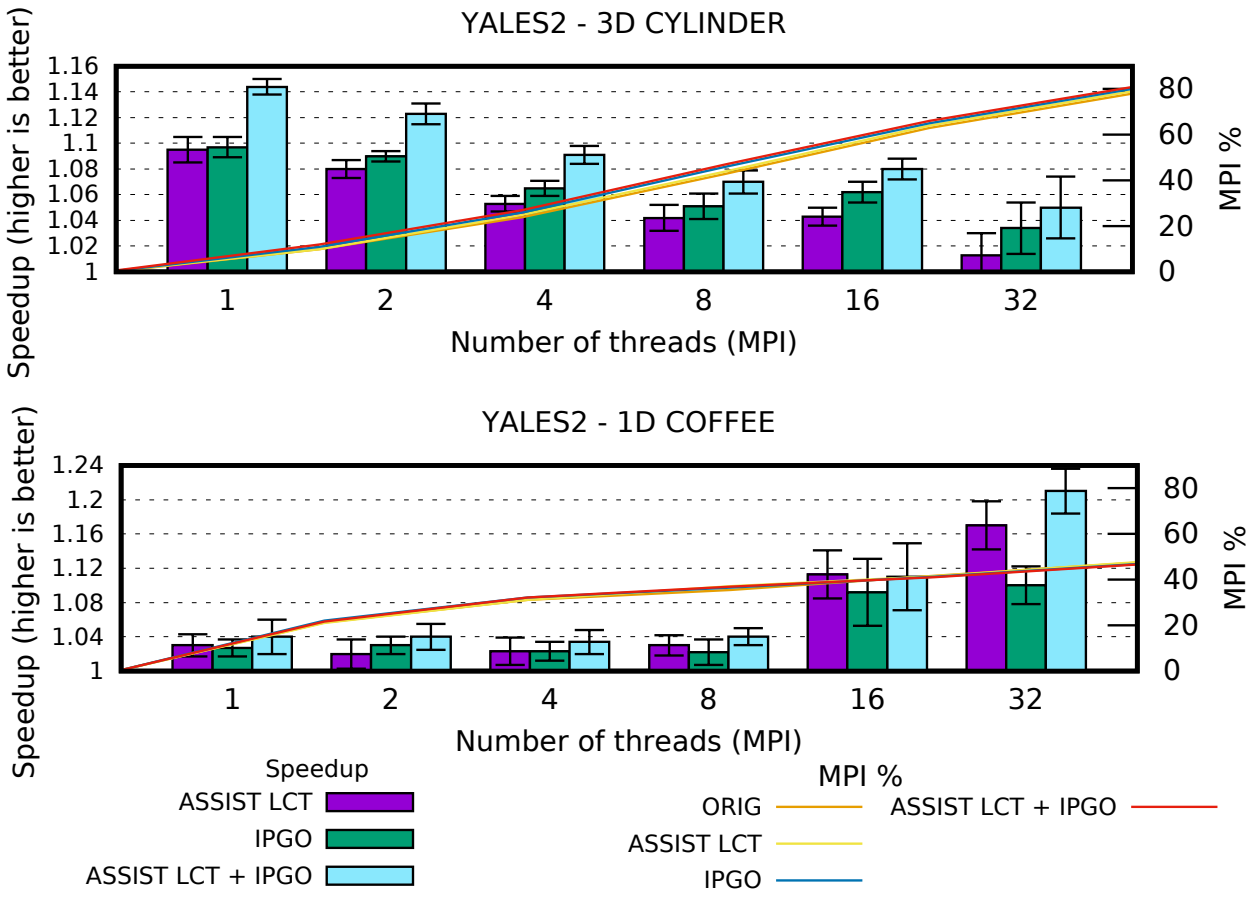


Figure 4.1: **Histograms**: impact (speedup) of ASSIST LCT, IPGO and combination of both compared with the original version for the same number of threads of two datasets Yales2 (Higher is better). **Error bars** represent original version divided by minimum speedup and original version divided maximum speedup. Plots: Percentage of execution time spent in MPI.

ensure that the optimization is still efficient in parallel, these figures present the impact of LCT, of IPGO and of these two combined compared to original versions with the same number of processes. In most cases, the speedup decreases when the number of processes increases. This is due to the communications which proportionally increase (see MPI time plots) at the same time and take most of the execution time. On the contrary, for Yales2 with 1D COFFEE dataset, we observe speedup increase with the number of processes. This is due to an Intel compiler optimization on an Intel library function that performs a copy of a string used for all communications. Moreover, the higher the number of communications, the more often this function is called. Providing the trip count of the loop containing this function to the compiler, allows it to perform advanced optimization. This explains the speedup obtained by increasing the number of processes.

After applying ASSIST LCT, we used our verification system based on CQA to statically verify that the compiler did not generate a worse performing code. The verification system is not fully implemented, so we decided to only apply it on hot loops and confirm that the transformation does not downgrade performances. The strength of this transformation comes from the number of loops processed by ASSIST; as shown on figure 4.2, the first twenty loops provide more than fifty percent of the total speedup gain but 130 loops are necessary to reach a maximum speedup for Yales2. For AVBP, it only requires 15 loops to reach half of the total speedup and 90 loops for the maximum. We can observe some performance degradation on a few loops but in general these degradations are limited to 0.01 second and can be due to the approximation. Number of loops processed for each test case and application is defined in table 4.1.

This study shows that providing the compiler with a loop trip count feedback (minimum, average and maximum values) results in significant performance gains. When compared with IPGO, performance gains are lower but it should be kept in mind that IPGO and LCT ASSIST are using different optimizations. The most important point is that both can be combined and that their combination leads to higher gains.

4.3 Impact of Specialization

While optimizing applications, we notice that we often resort to function or loop specialization before applying other transformations. The following examples show how specialization alone, or coupled with other transformations, can provide significant performance gain.

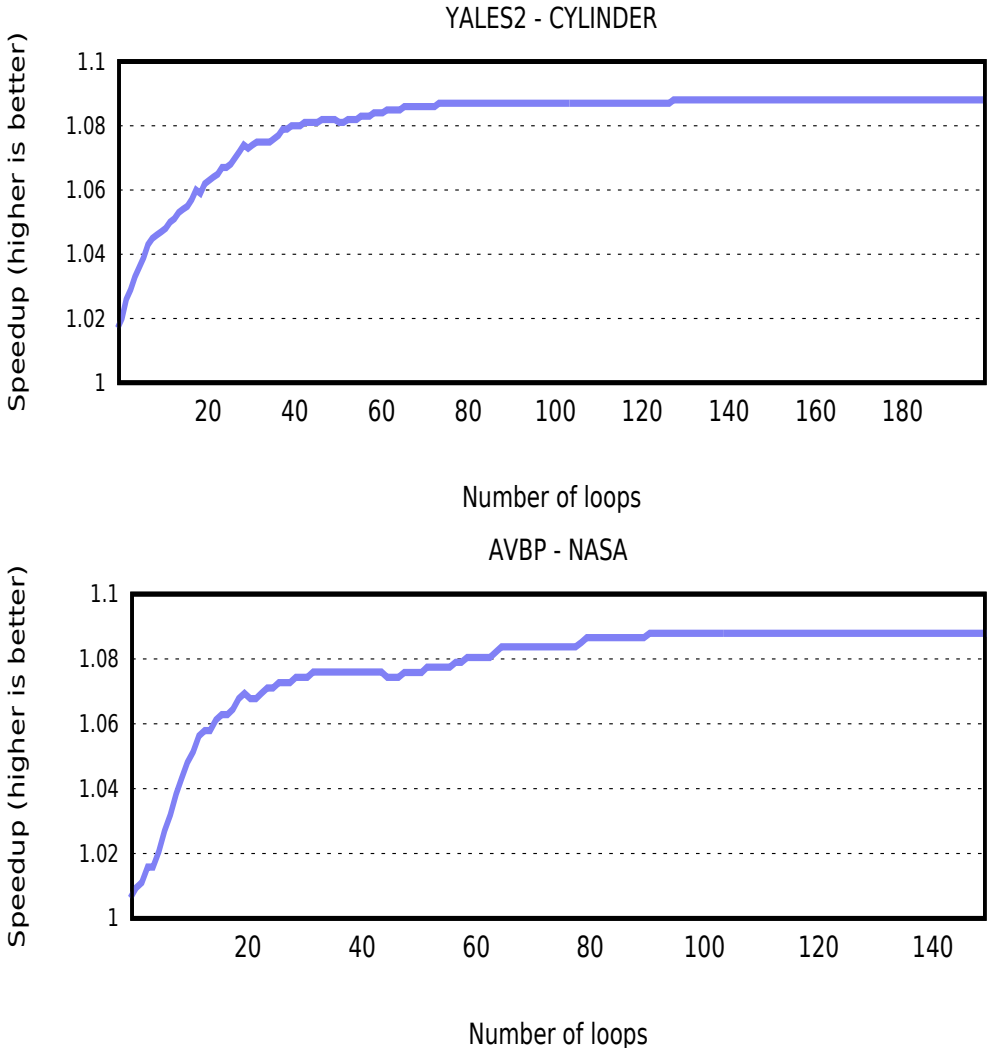


Figure 4.2: Cumulated speedup versus number of loops processed by ASSIST, sorted by their coverage, on Yales2 using the 3D CYLINDER test case and AVBP using the NASA test case.

4.3.1 Specialization Only

In this example, our target loop nest is composed of seven nested loops and ASSIST is used in two steps: first, as an automatic tool, using the automatic specialization to detect variables that can be automatically specialized. In this case, ASSIST finds that by specializing variables for certain values, it is possible to set bounds of the two innermost loops within the nest. It is also possible to remove the "if" statements that are in these two loops; then, as users, we know that two variables - which are computed inside the loop nest - only have three possible values for most layers. These are calculated within the loop nest which prevents the previous automatic specialization. After both specializations the loop nest has increased from 30 lines to 922 lines to handle all cases of specializations, this transformation can hardly be done manually without making mistakes.

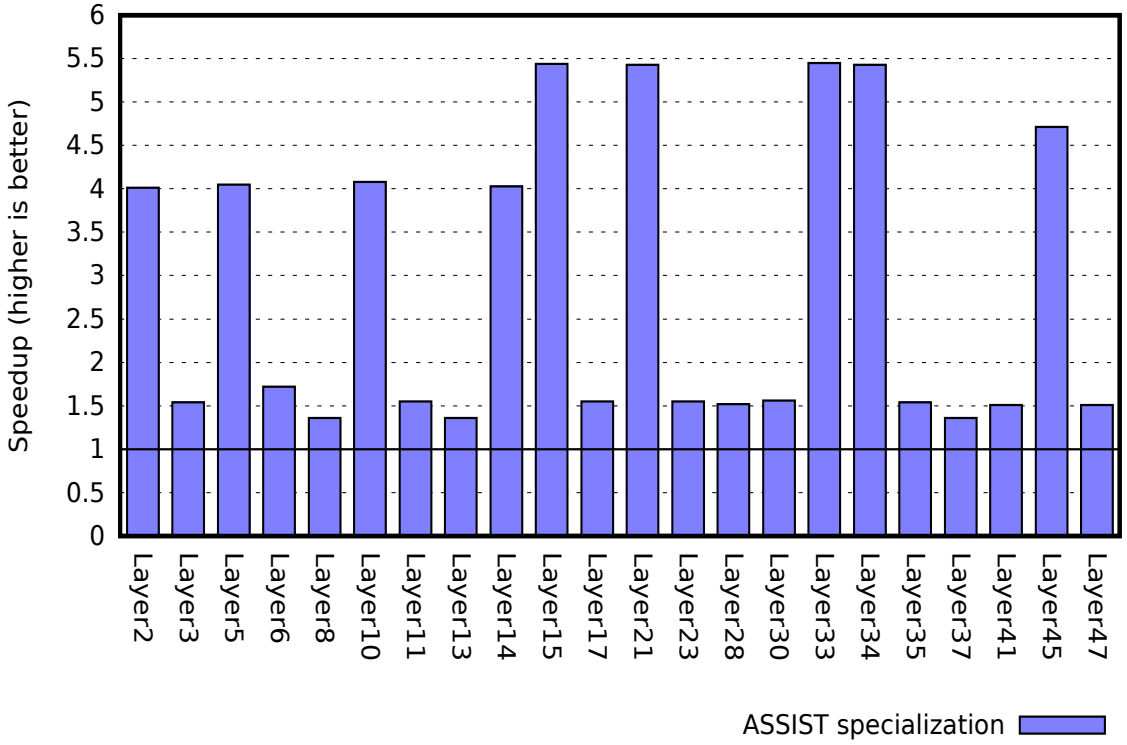


Figure 4.3: Convolution Neural Network - Speedup of GoogleNet_V1 layers after specialization, compared to the original version.

Figure 4.3 presents the speedups after specializations compared with the original version. Specializations offer a gain between 1.4x and 5.4x on all tested layers by creating multiple less complex versions of the loop nest that the compiler can more easily optimize. The layers used in this case are those with a (1x1) and (3x3) filters. IPGO does not appear on this figure because it does not gain any performance.

4.3.2 Combined With SVT

With AVBP

In this example, MAQAO indicates that, in the ten most time-consuming functions, there are loops presenting a poor vectorization efficiency and a low trip count for the three datasets: NASA, TPF and SIMPLE. We use ASSIST to couple both specializations and SVT on these functions. We first apply loop and function specializations separately, then we apply short vectorization on the most efficient version. Figure 4.4 only presents results on the dataset SIMPLE because compared to speedup obtained with IPGO and ASSIST LCT, it is the most relevant.

Figure 4.5 compares the speedup ratios of each version (LCT, IPGO, LCT + IPGO and SVT). For the TPF dataset, SVT allows to gain as much as the combination of LCT and IPGO. But for the NASA dataset, the best results of LCT+IPGO only allows to reach half of the speedup obtained with SVT for one MPI thread. It is more blatant with the SIMPLE dataset, the speedup of LCT and IPGO does not reach more than 2% individually and 4% when combined, contrary to ASSIST SVT which reaches a 12% speedup for the SIMPLE dataset. When the compiler fails to vectorize a loop properly, SVT is very effective given that it explicitly exposes a simpler loop structure with no peel or tail loops to the compiler.

There are two mains reasons why the compiler does not vectorize: first, the dependence analysis reveals dependencies preventing vectorization, and second, the cost model used by the compiler produces estimates indicating that a vectorization is not beneficial. For other cases, the compiler performed an outer vectorization on loops with a small number of iterations, CQA detects a bad "vectorization efficiency" on these loops. CQA offers multiple vectorization metrics such as vectorization-ratio or a vector-efficiency ratio on loads, stores, etc. allowing to assess the performance level obtained. In our case, we use these metrics to provide ASSIST with quality estimates of the vectorization carried out by the compiler. We can then decide whether or not to perform a good vectorization in order to finally trigger the transformation. The short vectorization transformations force the compiler to vectorize small loops with a small number of iterations; the compiler also fully unrolls these loops. After

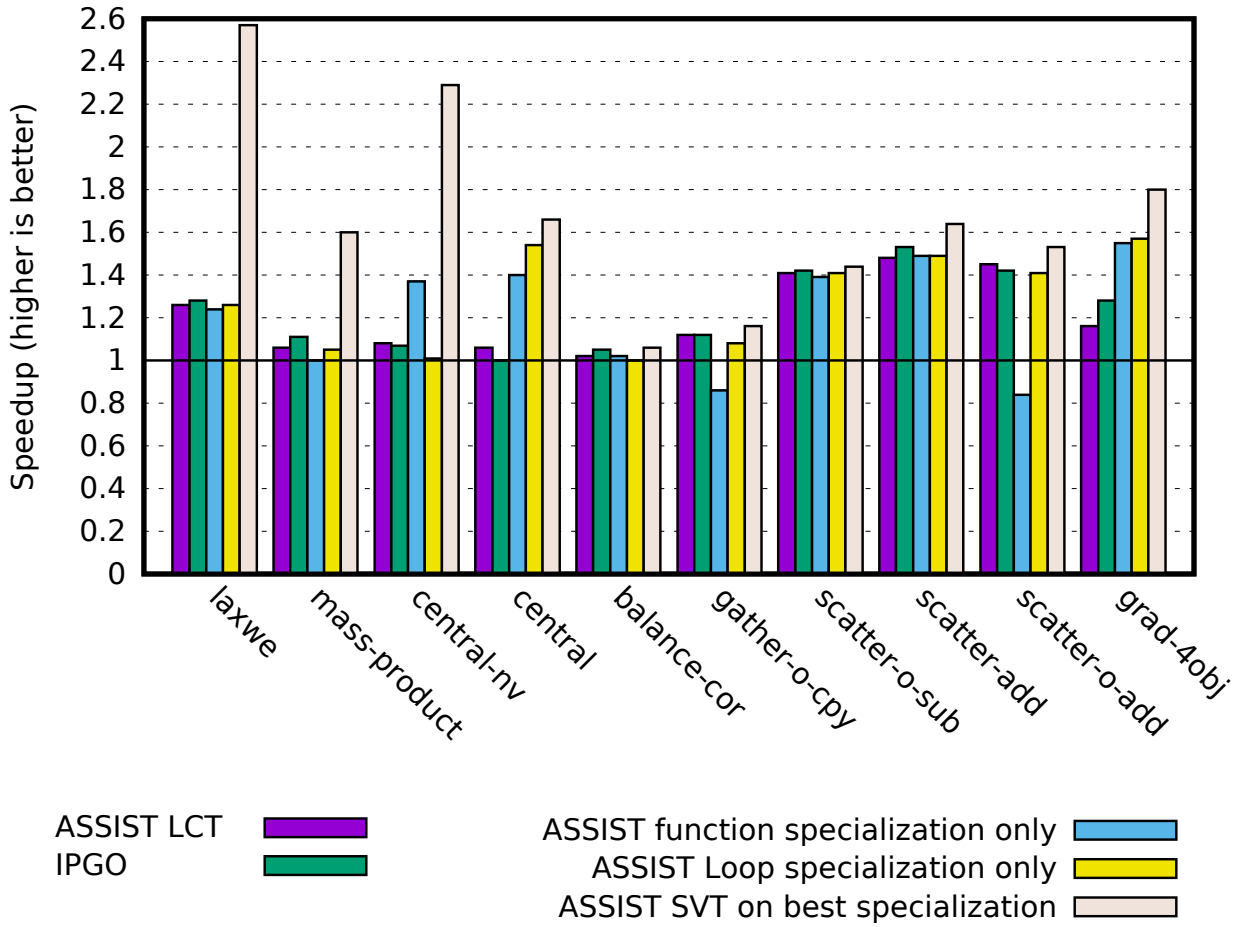


Figure 4.4: Speedups by function before and after applying transformations with ASSIST (SVT, function/loop specialization, LCT) and IGO compared with the original version (higher is better) on AVBP using the SIMPLE test case (sequential version).

loop id	# ite. min	# ite. max	# ite. avg	Potential speedup	Coverage
loop 2690	4	4	4	6.40	0.61
loop 2587	5	5	5	2.00	0.97
loop 2308	3	3	3	8.00	0.33
loop 2551	5	5	5	8.00	0.37
loop 2723	5	5	5	2.00	0.35
loop 16182	4	4	4	4.00	11.25
loop 13641	4	4	4	4.00	1.52
loop 13752	4	4	4	2.67	3.46
loop 13692	4	4	4	4.00	2.98
loop 13902	3	3	3	6.67	2.51

Table 4.2: CQA & VPROF metrics of loops of the hotspot functions of AVBP, with the SIMPLE dataset, before applying the SVT.

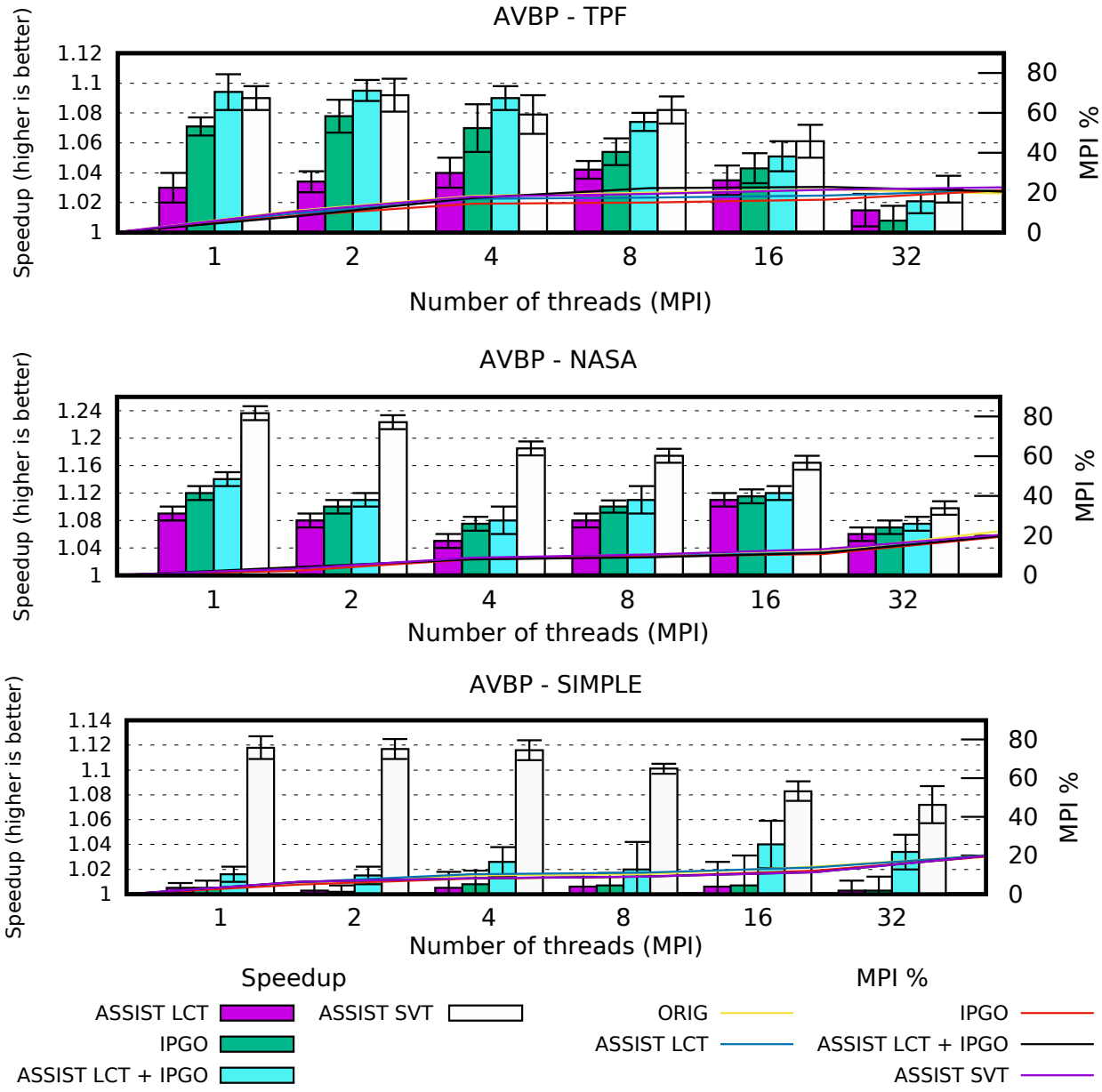


Figure 4.5: **Histograms**: Speedups of ASSIST SVT (i.e. short vectorization+function/loop specialization), ASSIST LCT, IPGO and ASSIST LCT+IPGO compared with the original version for the same number of threads (Higher is better) on AVBP using NASA, TPF and SIMPLE test cases. **Error bars** represent original version divided by minimum speedup and original version divided maximum speedup on AVBP. **Plots**: Percentage of execution time spent in MPI.

transformations, we use our verification system with CQA to validate the transformations. Indeed, before transformations, CQA only detects 33% of vectorization and after, CQA reports the loop as fully vectorized.

To apply SVT, loop bounds have to be known. To set these bounds, we specialize functions on one side, and loops on the other; we then apply the SVT on the best specialization for each function. Figure 4.4 presents the speedups obtained at each step to show their individual impact, we add ASSIST LCT and IPGO for comparison. We observe that SVT can raise up to 2.6x while the loop and function specialization only achieves, at best, a speedup of 1.5x. Performing only loop or function specialization may be counterproductive in some cases because of the induced complexity of the control flow, if no further induced optimizations are possible. Table 4.2 presents metrics from CQA and VPROF of loops before applying the SVT; these metrics did motivate our choice to use SVT. For all of these loops, the number of iterations is smaller than five and with a good potential speedup if fully vectorized.

To understand why the function specialization degrades performances of the function "gather_o_cpy", we analyze and compare the next three versions: the original version and the versions of both specializations. First, the original loop nest is presented on figure 4.6 and the results on the figure 4.7. They show the original loop nest and the execution information collected by MAQAO on that loop nest. As we can see, the compiler has created several versions of this loop nest. Each loop of the figure 4.7 represent a different version of the same loop nest (which start at line "219" to "223"). The sum of the execution time of all of these versions is 7.1 seconds out of the 36.48 seconds of the execution of the function. For example, the loop 13955 corresponds to a version where the loop nest has been unrolled 72 times with the loop 13944 as peel/tail.

Now, we compare both specialization versions. On one side, figure 4.8 shows the results for the function specialization version. All versions of the function are detailed with their coverage, execution time by function and associated loop nests. On the other side, figure 4.9 shows the results for the loop specialization version. All loops versions are in the same function. To easily compare them, we numbered each loop nest. Each number correspond to the same specialized version. There was no number in figure 4.7 because all loops correspond to the same source loop.

We can observe one main reason for this performance degradation. The compiler differently managed the two versions. For the function specialization version, most of loops exist in multiple versions while on the loop specialization version, loops exist in only one version. This is the main reason why function specialization is slower. The difference between the two versions without loops is close of five seconds. By duplicating the loop, the compiler can see that is multiple versions of the same loop; while when the function is duplicated, the

```
218   ...
219   DO n=1, nel
220     DO no = 1, nvert
221       !DIR$ SIMD
222       DO k= -nproduct+1, 0
223         zobj(no * nproduct + k , n) = z(ielob(no,n) * nproduct + k)
224       END DO
225     END DO
226   END DO
227   ...
```

Figure 4.6: The loop nest of the function "gather_o_cpy".

Name	Coverage (%)	Time (s)
▼ gather_o_cpy	13.67	21.18
▼ Loop 16183 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0.27	0.42
○ Loop 16182 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	3.83	5.93
▼ Loop 16181 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0.23	0.36
○ Loop 16180 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	2.32	3.59
▼ Loop 16178 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0.09	0.14
○ Loop 16177 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0.58	0.9
▼ Loop 16194 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0.08	0.12
○ Loop 16193 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	2.07	3.21
▼ Loop 16192 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0.07	0.11
○ Loop 16191 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	1.82	2.82
▼ Loop 16214 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16213 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16216 - gather_o_cpy.f90:200-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16215 - gather_o_cpy.f90:200-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16201 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16202 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16218 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16217 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16179 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16212 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16211 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16196 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16188 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16187 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16176 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16206 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16207 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16208 - gather_o_cpy.f90:200-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16209 - gather_o_cpy.f90:200-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16220 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16219 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16186 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
▼ Loop 16185 - gather_o_cpy.f90:197-201 - AVBP_V7.0.1_orig	0	0
○ Loop 16184 - gather_o_cpy.f90:198-201 - AVBP_V7.0.1_orig	0	0

Figure 4.7: Original version: Execution time details for the function "gather_o_cpy" and all the variants of its loop.

	Name	Coverage (%)	Time (s)
	▼ gather_o_cpy_flat_2d_assist_nverte4_nproductmod42	7.55	12.21
1	▼ Loop 9633 - gather_o_cpy.f90:314-318 - AVBP_V7.0.1_speF	0.36	0.58
1	▼ Loop 9632 - gather_o_cpy.f90:315-318 - AVBP_V7.0.1_speF	2	3.23
1	○ Loop 9635 - gather_o_cpy.f90:317-318 - AVBP_V7.0.1_speF	3.79	6.13
1	○ Loop 9634 - gather_o_cpy.f90:317-318 - AVBP_V7.0.1_speF	1.37	2.21
	▼ gather_o_cpy_flat_2d_assist_nverte4_nproductmod41	3.92	6.34
2	▼ Loop 9637 - gather_o_cpy.f90:288-292 - AVBP_V7.0.1_speF	0.27	0.44
2	▼ Loop 9636 - gather_o_cpy.f90:289-292 - AVBP_V7.0.1_speF	1.51	2.44
2	○ Loop 9639 - gather_o_cpy.f90:291-292 - AVBP_V7.0.1_speF	1.7	2.75
2	○ Loop 9638 - gather_o_cpy.f90:291-292 - AVBP_V7.0.1_speF	0.43	0.7
	▼ gather_o_cpy	2.88	4.66
4	○ Loop 9567 - gather_o_cpy.f90:155-158 - AVBP_V7.0.1	2.17	3.51
5	▼ Loop 9569 - gather_o_cpy.f90:339-343 - AVBP_V7.0.1_speF	0.04	0.06
5	○ Loop 9568 - gather_o_cpy.f90:340-343 - AVBP_V7.0.1_speF	0.58	0.94
	▼ gather_o_cpy_flat_2d_assist_nverte4	2.14	3.46
3	▼ Loop 9645 - gather_o_cpy.f90:235-239 - AVBP_V7.0.1_speF	0.07	0.11
3	▼ Loop 9644 - gather_o_cpy.f90:236-239 - AVBP_V7.0.1_speF	0.39	0.63
3	○ Loop 9647 - gather_o_cpy.f90:238-239 - AVBP_V7.0.1_speF	1.28	2.07
3	○ Loop 9646 - gather_o_cpy.f90:238-239 - AVBP_V7.0.1_speF	0.4	0.65

Figure 4.8: Function Specialization version: Execution time details for the function "gather_o_cpy" and its loops.

compiler can miss the fact that is the same loop in the specialized version.

Such cases can be detected by subtracting the time of the targeted loop to the time of the function containing this loop. It will help to decide which one of both loop and function specializations to perform and thus avoiding such performance slowdowns. However, subtract the execution time of the loop to the one of the function only work if we want to only specialize this loop and the specialization has no impact on the rest of the function.

With Polaris

Polaris has the same problem as AVBP with two most time-consuming scatter/gather loops with poor vectorization efficiencies. Their number of iterations is higher than usual (around 60 for both) so we use the ASSIST generic SVT with a modulo of four. Table 4.3 presents the results of these two loops when using the dataset "test_1.0.5.18". These two

	Name	Coverage (%)	Time (s)
	▼ gather_o_cpy	13.91	21.31
4	○ Loop 9465 - gather_o_cpy.f90:165-168 - AVBP_V7.0.1_speL	2.21	3.39
1	▼ Loop 9473 - gather_o_cpy.f90:223-227 - AVBP_V7.0.1_speL	0.29	0.44
	○ Loop 9472 - gather_o_cpy.f90:224-227 - AVBP_V7.0.1_speL	3.87	5.93
2	▼ Loop 9471 - gather_o_cpy.f90:214-218 - AVBP_V7.0.1_speL	0.21	0.32
	○ Loop 9470 - gather_o_cpy.f90:215-218 - AVBP_V7.0.1_speL	2.55	3.91
1	▼ Loop 9486 - gather_o_cpy.f90:223-227 - AVBP_V7.0.1_speL	0.09	0.14
	○ Loop 9485 - gather_o_cpy.f90:224-227 - AVBP_V7.0.1_speL	2.05	3.14
3	▼ Loop 9484 - gather_o_cpy.f90:241-245 - AVBP_V7.0.1_speL	0.08	0.12
	○ Loop 9483 - gather_o_cpy.f90:242-245 - AVBP_V7.0.1_speL	1.78	2.73
5	▼ Loop 9468 - gather_o_cpy.f90:232-236 - AVBP_V7.0.1_speL	0.07	0.11
	○ Loop 9467 - gather_o_cpy.f90:233-236 - AVBP_V7.0.1_speL	0.59	0.9

Figure 4.9: Loop Specialization version: Execution time details for the function "gather_o_cpy" and its loops.

loops represent 10% of the coverage of the whole application. ASSIST has been used on only these two loops for two reasons. First, 70% of the application time is passed in a function which computes each point of the matrix without using a loop and where we can not perform any transformation. Second, other loops are array line fortran representations; compilers refuse any loop-directive above this kind of loop, so we cannot apply the LCT on Polaris.

SVT has not been applied on Yales2 because CQA indicated that vectorization would lead to the use of scatter/gather instructions which are costly and make vectorization not beneficial. The level of indirection is high, due to irregular geometric access and the main

	Execution Time Before Transformation (sec)	Execution Time After Transformation (sec)	Speedup (higher is better)	Coverage
Polaris	73.32	70.26	1.04	
loop 6909	4.27	3.14	1.36	5.72%
loop 6911	3.64	2.36	1.54	4.98%

Table 4.3: Execution time and speedups of ASSIST SVT (i.e. generic short vectorization) compared with the original version on Polaris using the "test_1.0.5.18" test case.

bottleneck is the address computation.

4.3.3 Combined With Tiling

In this example, ASSIST is used as a semi-automatic tool and is fully driven by the user. At first, a full profiling of the code is performed followed by a value profiling on one of the main hotspots of the application. Three input parameters were found to be of importance.

First, the function can be called with two different types of input data, either real-valued data or complex-valued data. A given test case will almost exclusively use one or the other. As those data are expressed as an array with one or two elements in a part of the code, specialization of this value simplifies address computations and vector accesses by making the stride a compile-time constant rather than a dynamic value.

Second, multiple variants of the algorithm are implemented in the function. Which exact variant is used, depends on two integer parameters. Again, a given test case is usually heavily biased towards a small subset of possible cases. The specialization of one case allows to remove multiple conditionals. For a given case, different branches appears in the loop nests. This removal of conditionals exposes the true dynamic chaining of the loop nest directly to the compiler with no intervening control-flow break.

Once specialized with ASSIST, the function becomes much simpler to study. A study using MAQAO DECAN previously indicated that data access was very costly and that tiling would be very beneficial. More precisely, a large array is updated in its entirety inside a loop; a bad pattern for cache usage. Loop tiling makes it possible to update the array by block, and to only scan and update the array once. While this work would not be particularly difficult to do by hand, more than two dozen variants of the loop nest with similar properties appear in the original function. As the transformed loop adds an extra loop to the nest, this complicates indexes and requires a remainder loop. It is much easier and much more reliable to automate the transformation process.

Part (a) of figure 4.10 shows the directives on an extract of the function. Three specialized variants are produced for the common use cases in our reference test Ti-256, by the first three lines of the figure. The critical loop nest is subsequently tiled, but only in the specialized version, by the directive immediately above the loop nest. Part (b) shows extracts from the output of ASSIST. The specialized variants are called whenever the parameters are appropriate.

Every condition previously dynamically encountered is now collapsed into that one test. The original function figure 4.10 also shows the new loop nest with the loop tiling transformation applied. Only height elements (a friendly value for a vectorizer) are computed in the

```
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt=3,cplex=2)
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt<3,cplex=2)
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt>3,cplex=2)
subroutine opernlb_ylm(choice,cplex,paw_opt,...)
 ...
 if(choice==1) then
  !DIR$ MAQAO IF_SPE_choicee1_TILE_INNER=8
  do ilmn=1, nlmn
   do k=1, npw
    z(k)=z(k)+ffnl(K,1,ilmn)*cplx(gxf(1,ilmn) &
         ,gxf(2,ilmn),kind=dp)
   end do
  end do
 end if
 ...
end subroutine
```

(a) Before specialization + tiling

```
SUBROUTINE opernlb_ylm(...)
 IF ((choice.EQ.1).AND.(paw_opt.EQ.3)AND(cplex.EQ.2)) then
  CALL opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
  RETURN
 END IF
 IF ((choice.EQ.1).AND.(paw_opt.LT.3)AND(cplex.EQ.2)) then
  CALL opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
  RETURN
 END IF
 IF ((choice.EQ.1).AND.(paw_opt.GT.3)AND(cplex.EQ.2)) then
  CALL opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
  RETURN
 END IF
 ...
END SUBROUTINE
...
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
 ...
 lt_bound_npw = (npw / 8) * 8
 DO lv_var_k = 1, lt_bound_npw, 8
   DO ilmn = 1, ilmn
     DO k = lt_var_k, lt_var_k + (8-1)
      z(k)=z(k)+ffnl(K,1,ilmn)*cplx(gxf(1,ilmn) &
          ,gxf(2,ilmn),kind=dp)
     END DO
   END DO
 END DO
 ...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opti3_cplexe2(...)
 ...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opts3_cplexe2(...)
 ...
END SUBROUTINE
```

(b) After specialization + tiling

Figure 4.10: ABINIT - Example of function specialization coupled with loop tiling, performed with ASSIST, for the use case Ti-256. Boxes highlight the tiling transformation of the innermost loop.

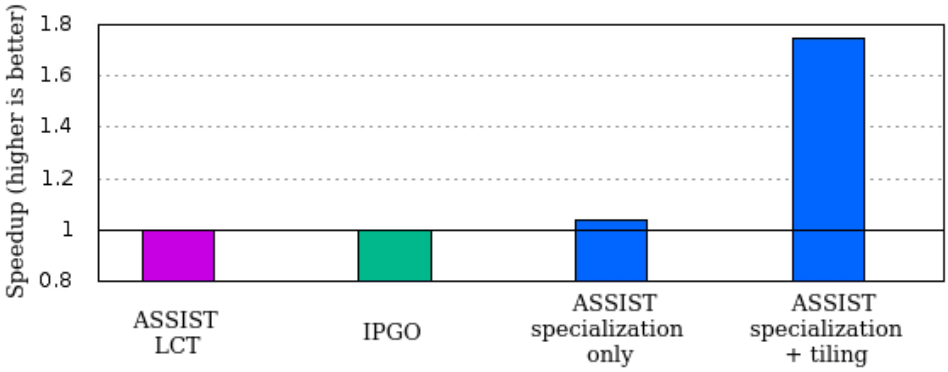


Figure 4.11: ABINIT - Ti-256 - Speedups of IPGO, ASSIST LCT, specialized with ASSIST, specialized and tiled with ASSIST compared to the original version

innermost loop versus the entire array previously. An outer loop has been added which scans the entire array by block of size height. In practice, the innermost loop is removed by the compiler which fully unrolls and vectorizes it.

Speedup results are shown in figure 4.11. We added IPGO to show the potential of our approach. Specialization offers a small gain but the dominant issue is still the time spent in the critical loop nest. Adding tiling offers a large gain of almost 1.8x in total by significantly reducing the memory bandwidth of the critical loop nest. Despite the complexity of the original function, the same transformations could be easily applied to other cases using ASSIST.

4.4 Impact of Prefetchers

Prefetchers setting has an important impact on performance. The default setting of all prefecthers on is not necessarily the best one. In this section, we analyze the impact of different prefetcher configurations on three applications with three different behaviors.

4.4.1 With Mini QMCPAK

The first application used is Mini QMCPACK. Specific runs have been made to test various prefetcher configurations, presented on table 4.4. All prefetchers ON are encoded as 0 and all

Activated prefetchers	Config 0	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6	Config 7
L2 hardware prefetcher	0	1	0	1	0	1	0	1
L2 adjacent cache line prefetcher	0	0	1	1	0	0	1	1
DCU prefetcher	0	0	0	0	1	1	1	1
DCU IP prefetcher	0	0	0	0	0	0	0	0
Activated prefetchers	Config 8	Config 9	Config a	Config b	Config c	Config d	Config e	Config f
L2 hardware prefetcher	0	1	0	1	0	1	0	1
L2 adjacent cache line prefetcher	0	0	1	1	0	0	1	1
DCU prefetcher	0	0	0	0	1	1	1	1
DCU IP prefetcher	1	1	1	1	1	1	1	1

Table 4.4: The different prefetcher configurations, according to Intel: https://software.intel.com.
0=prefetcher ON, 1=prefetcher OFF.

prefetchers OFF are encoded as f. As reminder of section 2.3.8, there are 4 prefetchers. Each of them can be turned ON or OFF, making a total of 16 different prefetcher configurations available. For each prefetcher configuration a full run including profiling at the function and loop level, was performed. We compared the impact on performance at three levels: full application, function and loops.

With this experimentation we want to analyze the impact of prefetchers on the Mini QMCPACK application using different datasets. All executions have been done on an Haswell-E. The application takes as input : "-n <I> -g <X Y Z>", where "<I> represents the number of iterations and "<X Y Z>" the size of the problem. The tests cases used for our experiments are : "-n 320 -g "2 1 1" ", "-n 160 -g "2 2 2" ", "-n 40 -g "2 2 2" ", "-n 20 -g "4 2 2" ". Figure 4.12 represents the speedup by function for all configurations for the eleven hottest functions of the application. Figure 4.13 represents the speedup by loop for all configurations for the eleven hottest functions of the application. During the different runs, we noticed instabilities (between 2 to 10%) for all configurations, but it was interesting to see that trends were preserved.

Figures 4.12 and 4.13 show is that each time the "L2 hardware prefetcher" is disabled, performances are degraded. Another observation is that other prefetchers have less impact on the application. Even if at loop or function level, modifying prefetcher configuration allows to obtain good speedups; the configuration with all prefetchers enabled stays the one that

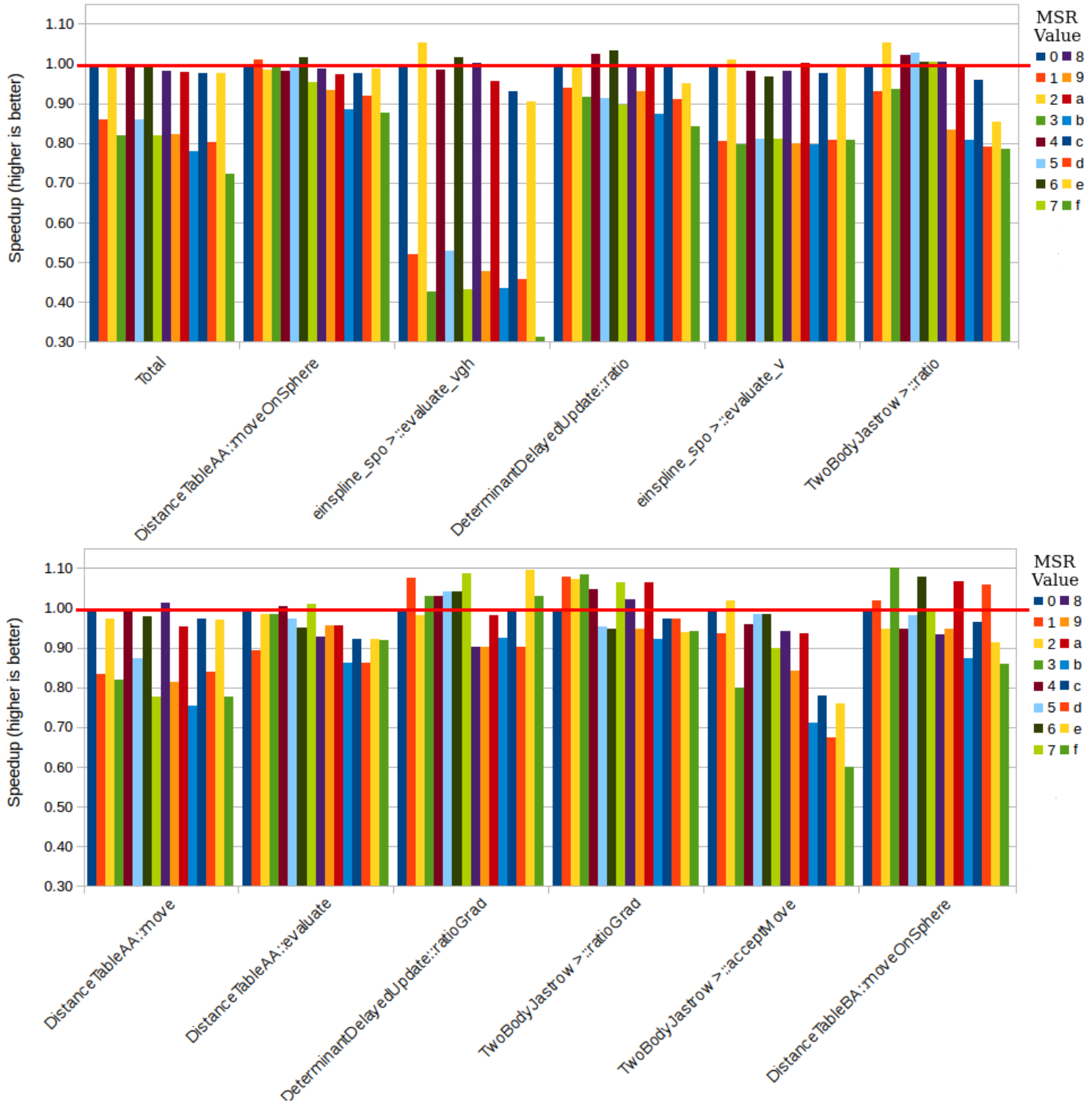


Figure 4.12: Mini QMCPACK - <-n 20 -g "4 2 2"> - Speedup by function for all configurations. All speedups are compared to the configuration 0 (all prefetchers ON). The graph is divided into two parts.

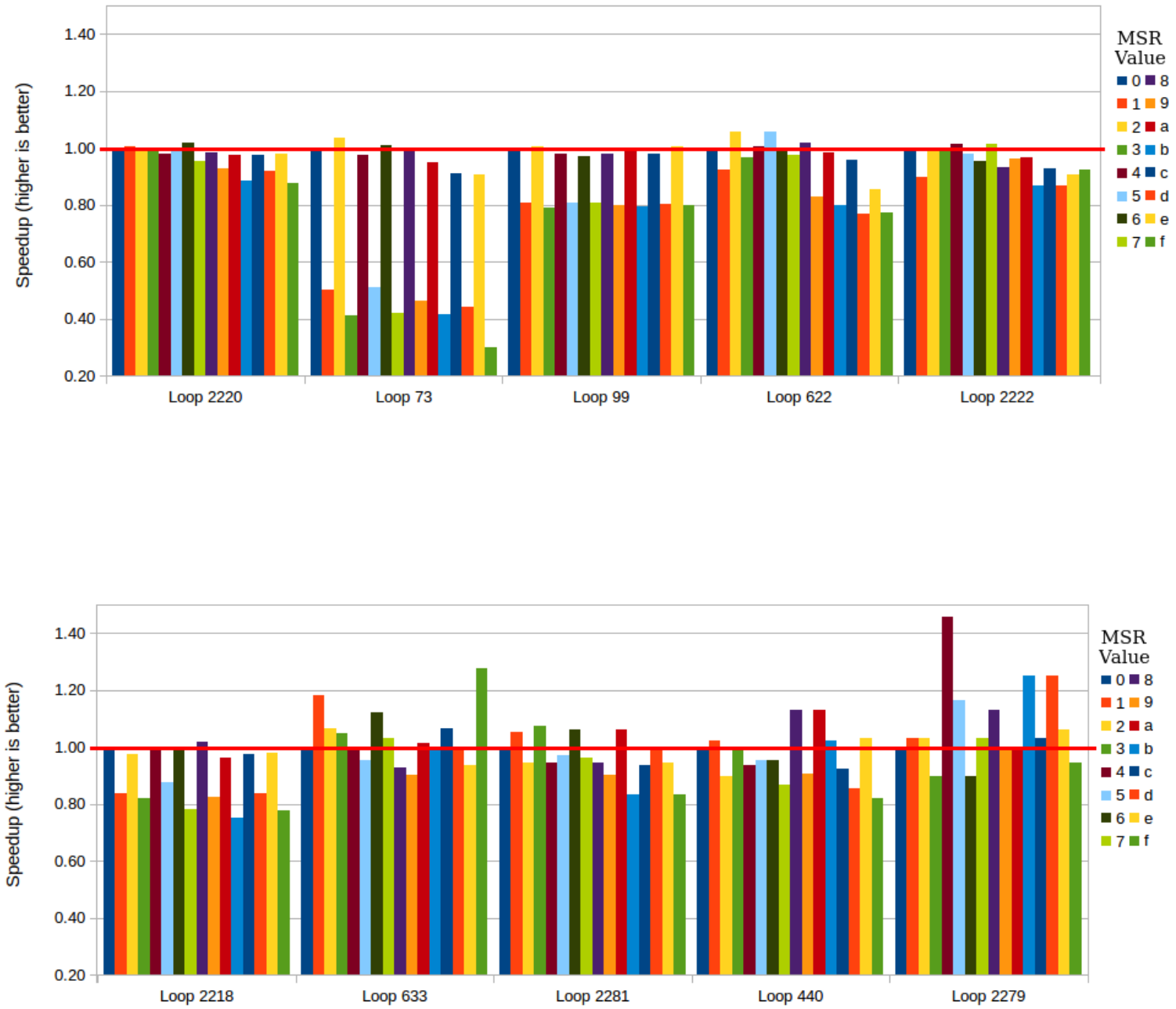


Figure 4.13: Mini QMCPACK - <-n 20 -g "4 2 2"> - Speedup by loop for all configurations. All speedups are compared to the configuration 0 (all prefetchers ON). The graph is divided into two parts.

allow to obtain best performances on the whole application. However, if our aim would not be reducing execution time but reducing energy consumption, the configuration "e" would be the best choice because we disabled three of the four prefetchers while reaching 98% of the best combination configuration. Since we do not have metrics on energy consumption, we will not dwell on that part in this chapter.

Our first tests of using ASSIST to locally modifying prefetchers configuration by adding calls at function level according to the best configuration were not conclusive. By calling the code in Appendix B.1, which only writes the value corresponding to the desired configuration in a specific register for each CPU, the operating system added guards that add a slowdown to the execution. This slowdown can multiply the execution time from three to ten times. The same effect has been observed even on small codes and benchmarks such as Numerical Recipes (NR) and no alternatives were found. A last remark is that trends are similar at function and loop levels. This can be explained by the fact that most of the time, functions are composed of one loop or loop nest which represents the most part of the execution of the function. Subsequently, we only present results obtained at function level.

4.4.2 With AVBP

The second application used is AVBP. Figure 4.14 presents the speedup of each version of the MSR compared to all prefetchers enabled of hottest functions (functions with at least 1% coverage). As we can see, most of the time, having all prefetchers enabled is more often efficient at application level, except for the configuration 4 and 6 where we have a little speedup at application level. With some combinations, enabling only a few prefetchers allows to obtain closed performances gain but each times that "L2 hardware prefetcher" is disabled performances dropped significantly. At function level we observe that we could obtain good results by modifying prefetchers configuration at function level. They are several function where different configuration allow to obtain better speedup that the configuration 0. In some cases, speedups up to x1.20.

4.4.3 With Yales2

The last application used is Yales2. Previous results on AVBP and Mini QMCPACK have showed that when executed in sequential, the best prefetchers combination for best performance is often when all prefetchers are enabled. We also saw that by disabling some of prefetchers could save energy while still being competitive in term of performance.

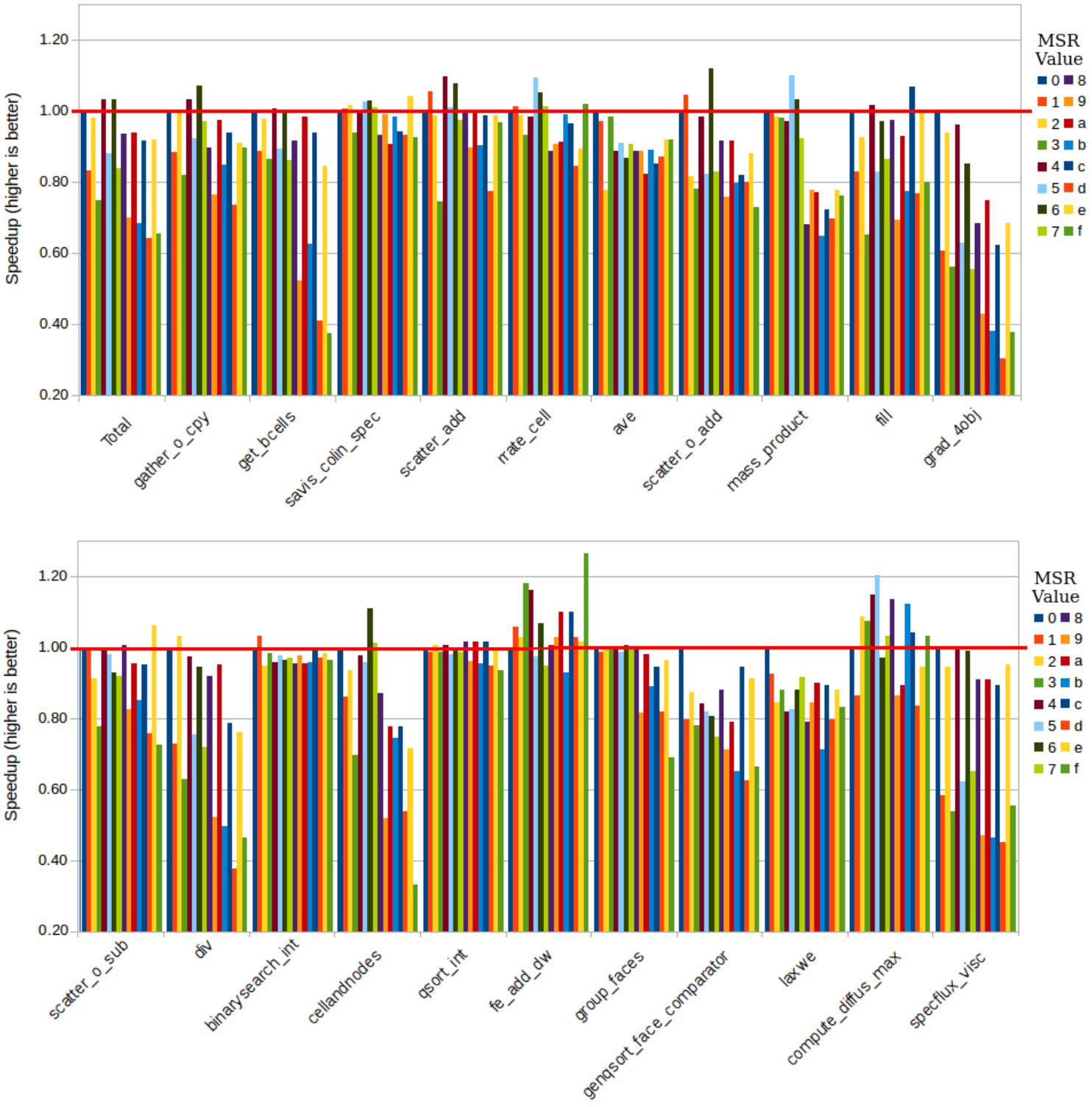


Figure 4.14: AVBP - SIMPLE: Speedup by function for each prefetcher configuration. All speedups are compared to the configuration 0 (all prefetchers ON). The graph is divided into two parts.

Figure 4.15 presents the speedup of each version of the MSR compared to all prefetchers enable of hottest functions (functions with at least 1% coverage). As for others applications, at application level, having all enable prefetchers is the most efficient combination and each time the "L2 hardware prefetcher" is disabled performances drops. However, with some combinations, enabling only few prefetchers allows to obtain closed performances gain and even better at function level, where we obtain speedups up to x1.40. But, even at function level, each times that "L2 hardware prefetcher" is disabled performances drop significantly.

We now analyze the impact of the different prefetchers setting on a parallel application, Yales2. Results for Mini QMCPACK in parallel were not presented because they are less interesting. Figure 4.16 presents speedups of the Yales2 dataset "3D Cylinder", with different prefetcher settings, for one, two and four MPI processes. All settings are compared to the configuration 0. As we can see on figure 4.16, all prefetchers on is the best configuration when the application is executed in sequential. However, the more the number of processes increases, the more the trend changes. With four processes the trend is inverted and the performance is improved. Parallelism adds a new perspective about the prefetcher settings. For applications that have to be executed with multiple processes, it is interesting to profile which configuration can bring performance. Nevertheless, all settings have to be profiled because it is impossible to predict performances according to one or another configuration.

4.5 Impact of Intrinsic Prefetcher Function

Prefetchers can also be controlled more precisely by adding an intrinsic prefetcher function call[3] in the source code. This function is used to indicate to fetch a line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint. The hints indicates if its is a temporal data, a temporal data with respect to second level cache misses, or a non-temporal data.

4.5.1 With Numerical Recipes

We first try this optimization on the well-known benchmarks Numerical Recipes (NR) [96]. We worked on three benchmarks, "s319", "s1244" and "matadd". When the intrinsic function was inserted in the code source, the compiler modified its optimization choices and disabled all optimization performed before that the call was added. Vectorization and unroll were disable. This is one of the main issue when optimizations are done at source level, the

[3]Intrinsic function : https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_prefetch

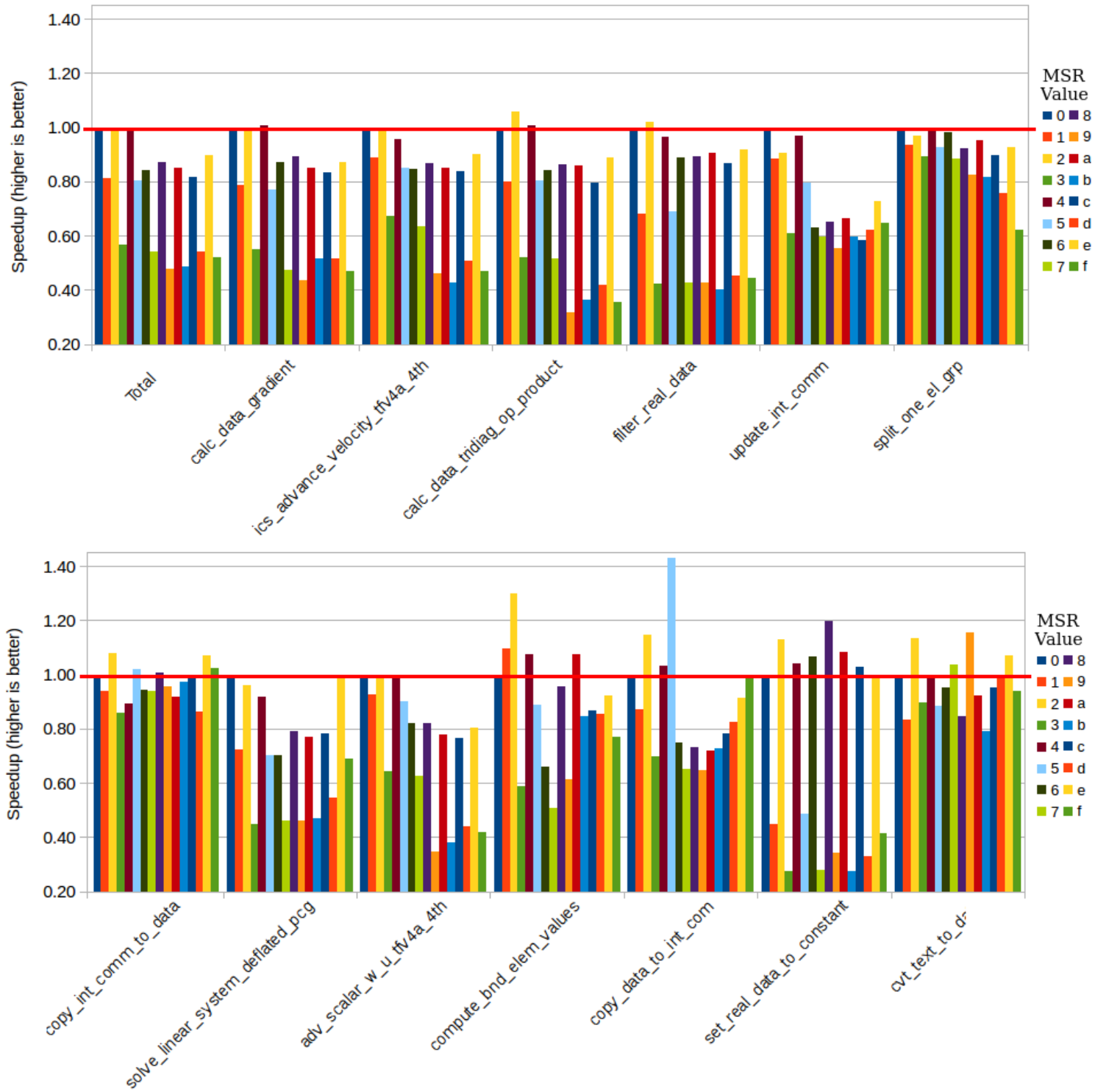


Figure 4.15: Yales2- 3D_Cylinder: Speedup by function for each prefetcher combination. All speedups are compared to the configuration 0 (all prefetchers ON). The graph is divided into two parts.

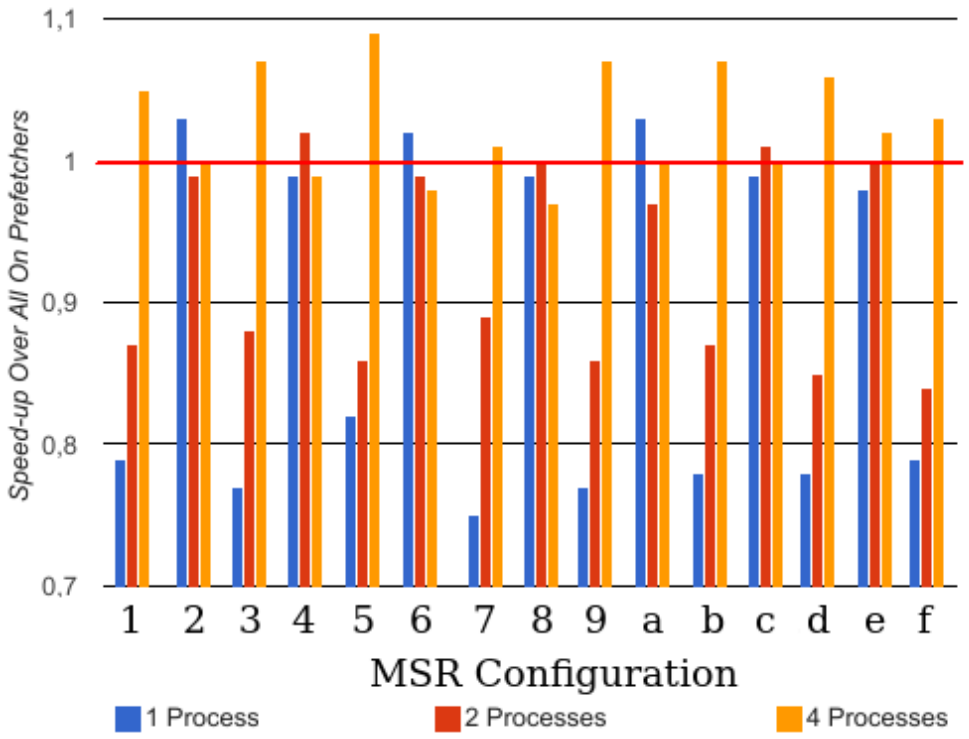


Figure 4.16: Yales2 - 3D_Cylinder: Speedup by prefetcher combination compared to all prefetchers enabled for one, two and four processes.

compiler can modify its optimization choices and it harms to our predictions. We forced the vectorization using the directive "simd" and unrolled the loop at source level but the compiler rewound the loop and still not vectorized. To verify if our optimization worth it anyway, we apply the optimization at binary level to keep previous compiler optimization in addition of the prefetch. The prefetch function has been only added for stored data. Table 4.5 shows that by prefetching store data with different distances allows an important improvement. From 12% speedup for s1244 to 45% for s319.

4.5.2 With QMCPACK

Same observations have been done with QMCPACK, the compiler disables all previous optimizations when intrinsic prefetcher function is added at source level. As for NR, the function has been added at binary level. Table 4.6 shows results on different loops of the application. We can see that the behavior may differ from a loop to another one. Fetching a data allows to obtain performance gains (loop 19042) as it can degrade (loop 19064). Sometimes it can

NR	ORIG	prefetch	prefetch 64	prefetch 128	prefetch 256	prefetch 512	prefetch 1024
s319	172936	112116	112248	112268	112172	117300	111832
s1244	131296	103236	102684	105352	104128	106724	103644
matadd	341436	278708	278388	281464	278040	274956	274228

Table 4.5: NR - Number of cycle for the target loop. Prefetch 64, 128, 256, 512 and 1024 indicates the distance of the data to prefetch.

have both depending of the distance prefetched (loop 30871).

Loop id	ORIG	prefetch	prefetch 64	prefetch 128	prefetch 256	prefetch 512	prefetch 1024
30954	5212	4892	5032	5012	5008	5040	5024
30944	1840	1220	1220	1220	1224	1220	1220
30871	490	500	456	484	500	492	488
19064	5672	5756	6188	5896	6816	6032	6348
19042	2726	1640	1520	1592	1648	1672	1628

Table 4.6: QMCPACK - Number of cycles for the target loop. Prefetch 64, 128, 256, 512 and 1024 indicates the distance of the data to prefetch.

For this transformation, the compiler remains our main limitation by refusing to apply optimizations as soon as the loop contains a function call, even if it is an intrinsic function. The second limitation is how to detect when to trigger this transformation. Prefetching data improves performance when the cache line is not in L1 and have to be loaded for not waiting when we need it. DECAN can inform us for these cases with the "REF" variant. This variant adds a load before stores data and compares the number of cycles for both versions. A last issue is for the distance to choose. Currently, we do not have metric to decide which distance choose, we have to test them all to know which one is the most efficient.

4.6 Impact of other common transformations

4.6.1 With QMCPACK

QMCPACK is a good example of when compiler failed to optimize and what is remained to do in term of transformations with ASSIST. With this example, we observe that the compiler tends to refuse to vectorize some loops. It prefers optimizes according to a pattern that we

	Orig	FU	DIV	SIGNBIT	SIMD
Total	143.18	138.74	127.22	129.8	127.76
ParticleBConds3DSoa.h	10.52	7.5	7.7	8.22	7.94
bspline_create.c	51.24	50.86	40.92	40.96	40.1
BsplineFunctor.h	1.4	1.32	1.46	2.5	1.4

Table 4.7: Time in second of multiple versions of QMCPACK. Files have been used as identifier because they contain multiple loops that have been optimized at the same time. Orig: Original version; FU: Full unroll version; DIV: FU + Divison replaced by multiplications; SIGNBIT: DIV = use signbit function to replace an if statement; SIMD: SIGNBIT + use of the directive "simd" above signbit loop.

must recognize without using vector instructions. Due to the lack of C++ management of Rose which considers template statements as string nodes we could not apply our optimizations with ASSIST. However, we tried to resolve manually the issues which where pointed to by MAQAO. At first, we worked on two categories: 1) All of the loops with bad vectorization ratio. 2) The loops with high flow complexity. Table 4.7 presents results obtained with different optimizations. All optimizations have been applied iteratively. The first loop we work on is an interesting case because it was a loop where inside there was an innermost loop with 7 iterations. This innermost loop was flagged with a directive to perform full unroll. In fact the compiler generated a very complex code with a lot of branching (more 128 paths, making static analysis difficult). We simply hand unrolled the loop which gave an overall decent performance gain around 2.5% execution of the whole application. This optimized version is called "FU". The second loop is in "bspline_create.c". It has three divisions using the same denominator. These divisions have been replaced by, one division of the inverse and three multiplications of the results of the division. This version is called "FU_DIV". For the "FU_DIV_SIGNBIT" verson, we use the "signbit" function to replace an if statement as follows. The "signbit"[4] function determines if the given floating point number in argument is negative and returns 1 if it is else 0.

[4]std::signbit : https://en.cppreference.com/w/cpp/numeric/math/signbit

```
#pragma vector always
for ( int jat = 0; jat < iLimit; jat++ ) {
  real_type r = distArray[jat];
  if ( r < cutoff_radius )
     distArrayCompressed[iCount++] = distArray[jat];
}

```

(a) Before "signbit" transformation

```
 #pragma simd
 for (int jat = 0; jat < iLimit; jat++)
 {
   real_type r = distArray[jat];
   int signbit = std::signbit(r - cutoff_radius);
   distArrayCompressed[iCount] = r * signbit;
   iCount+=signbit;
 }

```

(b) After "signbit" transformation

By applying this transformation, there are most instructions executed at each iteration but they can be vectorized. Therefore, the more the number of iterations, the more the code will be efficient. However, even if the code can be vectorized with this new implementation; the Intel compiler only vectorize the loop at 20%, even with the "#pragma vector always". We had to add the "#pragma simd" directive instead to be more aggressive and force him to vectorize this loop. The loop where the "signbit" transformation was applied has not enough iterations in this dataset to be efficient, but when vectorized we do not degraded performances. This last transformation is very complex to automate it, but it is interesting to present this solution to improve vectorization when compiler cannot perform it.

4.7 Conclusion

In this chapter, we have shown the efficiency of our approach: how and when already well-known transformations allow to gain on real-world HPC applications by using either static and dynamic feedback data, or user's guidance. No new techniques are developed, but new combination of transformations have shown worthwhile and it remains non-exploited promising analyzes and profiles. Moreover, our approach allows to remain portable across compilers and architectures. This was done using a combination of different performance analysis tools. A static analysis with CQA is used to give a first view of the quality of the code in term of vectorization. Then, a dynamic analysis is done with a profiling of: the

hotspots with Lprof, the number of iterations of loops with Vprof and the root cause of certain bottlenecks, such as data location, with DECAN. Better execution times are achieved on real-life HPC applications, with a speedup of up to x2.6 at loop level and x1.24 at application level. This study shows that the compiler, even with its PGO mode, cannot produce an optimal code for all cases. We also show that profiling more information is useful to detect precisely the bottleneck and thus apply the adequate transformation. However, our methodology has limitations which are detailed in the next chapter.

Chapter 5

Issues & Limitations

Our approach involved several elements during the process and we saw in previous chapters in what way each element is an asset to obtain performance gain. However, despite their strength, each element has flaws and can be a possible source of error. In this chapter, we list and discuss the different issues and limitations that may occur at each step of the process. Each limitation is divided into two types, on the one side, the weak limitations and on the other side, the strong limitations. Strong limitations represent all issues or limitations where presenting no ways to overcome the limits. Weak limitations are the others.

Analysis step:

This is the first step of the process. During this step, profilers and analyzers are executed to examine the program in all its aspects. All the MAQAO modules use debug information added by the compiler to locate what they analyze. Debug information contain, among other points, the start/end lines of each statement and the file and function where they are located. However, even when we explicitly ask to the compiler to add them with the "-g" option (for most of compiler), these are not necessarily very precise. For example, source lines depend on several parameters: for a loop, the indicated start line may be either the line where a variable used in the loop header was first seen, or most frequently, any of the one to three lines around. This also applies to the end line. This limitation is a weak one, because even if we cannot do anything to improve what the compiler returns, in general, we manage to find most of the elements within the few lines around.

Another weak limitation concerns the number of analysis performed; we only execute what is useful to trigger our transformations, but executing more profilers would allow to trigger more transformations, even if the analysis phase could become longer. The main issue is to find which transformation to associate to which profiler or metric. Even if the analysis phase becomes longer, the user can choose not to execute this or that profiler. Some users are ready to wait longer if it can add an additional speedup. This limitation is mainly due to what we can obtain from our analysis.

Transformation step:

The first limitation during this step is the Rose compiler. Even if it is the best framework to perform source-to-source transformations, as discussed in section 2.1.2, whether at input or at output, the Rose Compiler has a lot of lacks and limitations. Some of them are listed in the manual [71] or in the tutorial [110], but most have never been mentioned in the literature. Due to its lacks, Rose represents our main limitation in terms of the number of applications on which we can work and on the quality of produced outputs. As explained in section 2.1.2, Rose uses two frontends; the first one is the EDG frontend, a black box in which it is impossible to change anything. Moreover, we would have to pay a license to get a better version. The second frontend is the Open Fortran Parser, this one is free and can be modified. We have already proceeded to some changes. It is both a weak and strong limitation. Even if it remains a lot of work to correctly manage all input languages, Fortran is the only one for which we can handle its features and the missing elements. We cannot do anything to improve the C/C++ frontend. Moreover, some C++ statements and keywords handled by Rose are not fully managed. For example, the template statements such as functions, class, etc. are just nodes only containing a string of the whole statement. Each sub-element of these statements are not existing in the AST. Recently, researchers of the Lawrence Livermore National Laboratory (LLNL) started to work on the Rose git to improve how Fortran is handled. However, it only concerns the frontend and the inputs management and it seems unfinished yet. The rewriting system also needs to be improved. Keywords are missing, some lists are written in a bad order, comments and directives are not set at the right place, sometimes after the statement, instead of before. All this can change the whole program behavior if we do not pay attention and let it be compiled, etc.

Performing source code transformation has advantages, but forces us to be compiler dependent because it requires to re-compile source files after transformations. We are never sure of what the compiler will do with our transformations, it can decide not to follow what we have in mind and may decide to perform any other optimization instead. For example, we cannot be sure that after specialization, the compiler will optimize as we planned. Figure 4.4 in section 4.3 perfectly illustrates this example. In that case, the function specialization was less efficient than the loop specialization version. However, we would have thought that the compiler would optimize both in the same way. Operating at source level forces us to orientate the compiler as best as we can, we make our transformation as explicit as possible in the hope that the compiler will not try to modify it by performing an unexpected optimization afterwards. Most of the time, compilers act as we would expect and everything goes well, so it is a weak limitation.

Another limitation linked to the compiler dependency concerns transformations using directives (i.e. Loop Count Transformation). This kind of transformations adds a directive

above a statement to guide the compiler behavior. However, these transformations are compiler specific; all compilers define their own directives and for non standard directives, each compiler defines or names them differently. LCT is the perfect example, because the Intel compiler is the only one to provide this directive, other compilers will just ignore the directive. This time, it is a strong limitation because, when we use this kind of transformation the user is forced to use a specific compiler.

It can be hard to predict performance for some transformations. For example, to have the best results by modifying prefetcher behavior, each combination of prefetchers must be tested, that means executing sixteen times the whole program. This is a weak limitation because we just have to execute the program sixteen times to know which version to use at which moment of the program. It could be improved by testing the sixteen possibilities during the first iterations of the loops and keeping the best one at the seventeenth iteration. It requires a lot of specializations and we do not know yet if the performances would be enough to cover the loss in branches. When the prefetcher intrinsic call is added in the code, we saw that the compiler modify its own behavior and do not apply previous optimization such as the loop unroll. Due to his unpredictable behavioral change, this transformation must be applied at the binary level. This is a strong limitation because we cannot perform it at source level.

About the specialization, we have a weak limitation; we currently apply the loop specialization above the loop, but when that loop is in a loop nest, it sometimes could be better to apply the specialization at the top. This option is not as easy as it looks because in some loop nests, the bound of the innermost loop can change according to previous statements in the nest. Moreover, it is not always more efficient to apply the specialization upper in the nest.

When analyzers and profilers detect at least two possible transformations, we currently do not know which choice to make and how to advise the user without just giving him all the collected information with the raw data from profilers and analyzers.

The situation becomes worse if we detect a case where our analyzer knows what to do but it misses the right transformation. Currently, a whole API has been developed with already implemented transformations but some common and complex transformations are still missing. For example, one complex missing transformation is the array-of-structure to structure-of-array and vice versa. CQA can propose this transformation as a hint to users but we can not do anything right now with ASSIST. This is a strong limitation due to the complex missing transformations.

On the contrary, we have several transformations without any metrics. Currently, the unroll, full unroll, loop interchange, etc, are not associated to any metric to trigger them. It is a weak limitation, it only requires to add the good metric with the right profiler/analyzer.

All our transformations are based on analysis of the program executing a unique dataset. We cannot be sure that our transformations remain efficient for other datasets; it is a weak limitation because, in a first time, we could execute the different datasets and couple all information gathered to only perform the ones which have a positive impact for all the datasets. A beginning of work has already been done on this subject but not tested enough to present any results. We add the possibility to perform the LCT based on information collected from different datasets. The Intel compiler authorizes to insert the same directive multiple times above a same statement. The only advice is not to insert the same directive twice, so duplicates are removed to avoid this case. We have noticed that if we add two loop count directives above a loop, one with a small number of iterations and the other with a high number of iterations, the compiler does not take care of any of the two directives and we fail to guide the compiler. If a directive is already inserted, we can check if the new one is very different, to adapt or replace the previous one. Because, if the loop of the other dataset has a small coverage compared to the one we currently handle, it could be better to take care of the most important. It is a weak limitation because most of the work has already been done and it only requires times to finish and implement it correctly for the different cases as well as test it on real-world applications. This feature could be adapted to all transformations once we will have succeeded to gather and couple all information concerning all the metrics from multiple reports.

A last limitation about transformations is that we cannot be sure that it will be efficient before we execute it. We propose transformations, but we do not know their legality. As said previously, compilers can perform optimizations that were not expected and even if the compiler does what we want, it is possible that the transformation degrades performances due to unexpected side effect or just to a misprediction of the outcome. It is a strong limitation because we cannot predict and control the whole process.

We chose a semi-automatic model because we estimated that the fully automatic one was not a realistic model for real-world applications. However, it still has a default; on one hand, we provide a framework that performs automatic transformation to avoid the users to make errors when performing the changes manually; and on the other hand, we get the user involved in the process. Letting the user interfere in the process can be a good idea. He knows what happens in his code and if a transformation is legitimate or not. If not, he will want to try all transformations, especially the vectorization, but this last one can be a double-edged sword. When it is well placed, it can significantly increase performance gains, but if misused, performance will be greatly degraded. Letting the user choose his transformations can be tricky. In the end, it is a weak limitation because it is up to the user to be parsimonious in his choices of transformation.

Verification step:

Our last step of the process is the assessing transformation verification. It remains a prototype and remains the most unstable part of ASSIST. Limitations have already been listed in section 2.4, we just make a quick reminder. The main limitation of this verification system is to compare loops before and after transformations. Some transformations can make a loop disappear, the full unroll transformation for example, or can duplicate the loop, the SVT or specialization. It becomes then hard to completely compare different elements.

Our second limitation is that we stop at this point. The user have to be questioned whether he considers the transformed version as better than the previous one. We decided to involve the user at each step of the process. That makes our method stronger and safer in performance gains. It also slows down the process.

5.1 Conclusion

In conclusion, our approach allows to obtain good performance results, but involved several elements during the process. From the accuracy of the analysis to the verification of the impact of our transformations; each step has limitations of which we are dependent and which can be a source of error. Most of them have a weak impact and can be improved, but some are strongest such as the lacks of the Rose compiler frontend on which we are based.

Chapter 6

Conclusion

In this last chapter, we conclude this dissertation by first, listing the contributions of this thesis and then, discussing the different research perspectives and future works.

6.1 Contributions

This thesis presents an approach allowing to apply code transformations at source level being guided by both static and dynamic profiler analysis tools. This approach has been implemented through the ASSIST framework. ASSIST is a new open source semi-automatic source-to-source manipulation with code transformations based on static and dynamic feedback and on users knowledge. It aims at providing assistance with respect to productivity and performance efficiency.

- Our first contribution is a novel study of how and when, well known transformations allow to gain performance on real-world HPC applications. We have shown that by using well-known and standard transformations coupled with tools metrics and user's knowledge, it is possible to obtain good performance gain. Our novel FDO source-to-source approach, includes transformations from loop unrolling to loop/function specialization and short vectorization transformation. It is a technique that efficiently helps the compiler to fully exploit vectorization on loops, especially those with a low trip count.

- It is a novel semi-automatic and user controllable method with a system open to user advices. At different steps of the process, ASSIST can let the user apply his expertise to guide transformations by choosing between different optimizations according to profilers results; if the user finds that all proposed optimizations will not be efficient, he also may refuse to perform them. This system allows all kind of users to use ASSIST. From the non expert developer who can let ASSIST be guided by profiler and analyzer metrics, to the expert who can add his knowledge to guide ASSIST.

- This FDO tool can use both dynamic and static analysis information to guide code optimization while other existing tools only use static or dynamic feedback and never let users intervene in the optimization process.
- It offers a verification system using a static analyzer to check that proposed the transformations do not have a negative impact on performances by comparing two executions of the program, before and after transformations.
- Our last contribution is a more flexible alternative to the compilers PGO / FDO modes. Existing compilers PGO modes search through a limited space and only perform dynamic analysis. Our method explores a larger search space in term of performance analysis and can be more expensive due to the large amount of gathered performance metrics but it is more efficient. Last but not least we show that our approach can be combined to PGO.

6.2 Perspectives

In this section we discuss the possible perspectives and future works.

One of our first limitations is from Rose. Even if it is the best framework to perform source-to-source transformations, there is a lack at the frontend level that needs to be improved to be able to manage more codes and in particular C++ codes.

A first extension will be to increase the accuracy of analyzers by sharing source code information to other MAQAO modules. We previously saw that compiler debug information can be more or less accurate and depends of a lot of parameters. To be able to return source information to other MAQAO modules could help them improving comprehension of the code, to better understand what really happens and what the compiler has done and thus to return a better analysis. We first think of CQA which can return more accurate hints on how to overcome bottlenecks.

Several transformations can be implemented as extensions. Some of them have already been a base for future features. For example, the transformation of prefetcher behavior modification is already implemented but not sufficiently tested, and especially on real-world applications. We show that modifying prefetchers behavior can have an impact at fine grain (function level) depending on what is done at this level. It requires more research about how to detect a case which needs specific prefetcher behavior and how to chose the right prefetcher behavior without testing them all. It would take a long time to categorize functions and loops to know when and which prefetcher to apply. Ideas expressed in [104] and [93] could be starting points for that work but it should be applied at least at function level. As we saw in

section 4.4, choosing the right combination of prefetchers allows to keep performances closed to the best configuration but it could also reduce energy consumption.

Another already implemented transformation, which can be used as a base is the specialization transformation. This transformation has many possibilities of evolution. Currently, it only concerns integers, but it could be interesting to extend it to floats, while paying attention to float precision when making comparisons. In scientific computation, a number of constants are floats with only a few digits. Specialization could allow to help the compiler on these parts of the code. Moreover, our dead code elimination could be improved by removing more parts of the code. For example, when we specialize, we only remove the code in the specialized function or loop and do not change the original statements. However, we could look at these paths that are ignored in the original version, because they can only be taken in the specialized version. Thus we could simplify the control flow in both versions. Another possibility is, when a function is specialized, to replace all the calls to the original function by the specialized one when we are in the right case. Concerning the control flow, it could be interesting to have a dynamic profiler to detect which paths are always taken in the code, especially on a complex control flow that compilers have difficulties to manage. Thereby, we could create specialized versions to simplify each of them and help compilers do a better job. Another use case of specialization is to improve energy consumption. We could orientate this transformation with the aim of reducing the energy consumption. A first part of this idea has been developed in [26]. A last possible idea for an extension of specialization, is to perform pattern matching. In scientific codes we often find same pattern, especially with simulation codes for scientific purpose. Knowing that, we have shown that our transformations work on that kind of code, we could study what transformation always gains performance on what kind of pattern and why in some cases it never succeeds. Then we could apply pattern matching to trigger transformation without profiler and analyzer.

We saw in the previous chapter 5 that some transformations are hard to implement like the array-of-structures to structure-of-array and vice versa, or more simply by reversing two or more array dimensions. Depending of programming language, array dimensions are read differently. For example, in C, a two dimensions array (i.e. array[i][j]) is read line by column; in Fortran it is the opposite. By reversing array dimensions, it could help the compiler to better vectorize these multiple dimensions arrays.

Another transformation of code structure could be for Fortran code. In Fortran it is possible to work an array, or a sub part of an array with the following writing: $a(:) = b(:) + 1$. This means that each element of b is affected to a $+1$. The compiler does not consider that as a loop but transforms it into a loop at binary level. This false loop prevents to apply any transformation. Even the compiler forbids directives above this kind of statement because it is not considered as loop.

Memory alignment is also an issue which can be improved [49]. Several researches and

works have been done on how to detect and improve memory alignment [61, 87]. Data alignment can make a big difference in performance. It allows the processor to efficiently fetch data from the memory. To improve data alignment, we could work on data structure and padding. Usually compilers align data structures allowing to read an object using 4 bytes. Therefore, memory addresses have to be divisible by 4. Moreover, in most of HPC languages, the address of a structure is the same as the address of its first member. By miss-positioning declarations in a structure, we can easily waste a lot of space [1]. In a same way, pad arrays allows to align data on 4-bytes and thus allow to use large and efficient chunks for the entire array. The "memcpy" implementations in gcc is a good example. During the copy of a structure of 0x43 bytes, we may find an implementation that copies one byte leaving 0x42 bytes. Then it copies 0x40 bytes using large efficient chunks. Finally, the last 0x2-bytes are handled as two individual bytes or as a 16 bit transfer. "Alignment and target come into play if source and destination addresses are on the same alignment say 0x1003 and 0x2003, then we could do the one byte, then 0x40 in big chunks then 0x2, but if one is 0x1002 and the other 0x1003, then it gets really slow." [2]

All transformations available in ASSIST are only sequential transformations. We can more orientate our transformation through the parallelization by inserting OpenMP or OpenCl directives or by transforming C/C++ codes into Cuda codes like CAPS with HMPP [31] for example. Another way to orient our transformation through parallelism is to transform loops using the polyhedral system to be more efficient once executed in parallel. Numerous works have already been done on the subject but polyhedral and automatic parallelization remains a hard problematic that is still being studied.

Before implementing the previous complex transformations, it could be useful to have currently missing common transformations such as: 1) Loop splitting, which splits a loop into multiple parts; the main problem with this transformation is that it is not always trivial to know where to split a loop. 2) Padding, with arrays and then on loop working on that array; it will allow the compiler to improve vectorization by using aligned vectors for example. 3) Loop fusion, which gathers two loops or more; the main difficulty is to find at least two loops with same bounds which can be gathered. The second difficulty is that this case can not be detected with the currently used analyzer. The analysis has to be done at source level. These are some examples but a few more exist in the literature which have already been discussed.

Currently, most transformations have been thought to fix issues from a unique dataset. However, it could be interesting to gather information from several datasets and compare all information to apply a transformation that could be a little bit less efficient on one

[1] http://www.catb.org/esr/structure-packing/#_structure_alignment_and_padding

[2] https://softwareengineering.stackexchange.com/questions/328775/how-important-is-memory-alignment-does-it-still-matter

dataset but more efficient on more use cases. This could be more interesting than another transformation that could be very efficient on one use case but degrading performance for others. A beginning of these works have already been done for LCT. ASSIST can use several Oneview files and will apply the LCT without duplicating on all loops. The compiler allows multiple directives above a loop but it is only advisable not to put twice the same. The main problem is that the LCT is only efficient if directive bounds are not scattered, or if they are two not too different directives. For example, it will not help the compiler to indicate that the loop only performs three iterations for a dataset but thousands for another one. It will not optimize as it should be and can even be counter productive.

All profiler metrics are associated with one transformation. VPROF trip count is associated with LCT, DECAN DL1 to tiling, etc. However, we could go one step further by increasing the number of elements and by coupling them to increase the accuracy of analysis and propose a more adapted transformation and not wait for an iterative step to perform multiple transformations. For example, we could couple VPROF trip count and CQA vectorization efficiency to perform the short vectorization transformation in one step.

In addition to couple information, we should start by associating more metrics to triggered transformations. MAQAO has multiple non used modules and metrics. Increasing the number of analysis which can trigger transformation allowing to go further with finer grain analysis to more precisely guide transformation and thus be sure that what is produced is more efficient. For example, there is a tool in MAQAO that allows to simulate the processor and the memory of different architectures. We could use it to know if unrolling or tiling a loop could be efficient. Another example should be to use UFS, a module comparable to CQA but more accurate, to guide our transformation and help to check before and after that we do not degrade performances. Obviously all these profilers and analyzers have a cost but the user remains the only decision maker to trigger what kind of analysis he wants.

A last possible extension should be to offer to the users an iterative compilation where ASSIST would use the transformation verification system developed in section 2.4 to know if the next step is correct or if it requires to change transformations to be applied. The user must keep a way to intervene during the process but he will not have to recompile new source files. Also if ASSIST detects that transformations degrade performances, it will test the other transformations before returning the version to the user. Most parts of the work have been done with the verification system but it stops after one iteration and returns the report to the user without any analysis by ASSIST. Currently, the user has to cancel a branch of transformations detected as non performant and test other transformations.

Appendix A

Appendix: ASSIST

A.1 ASSIST Help

As seen in the previous section our tool can get multiple options. In this section, we will describe all available options in ASSIST, they can also be found in the help section of the module.

-src=<path/to/file>[,<file2>,...]	The input source file(s) to transform; if they are multiple files, they must be separated by a coma.
-oneview=<path/to/oneview/file>	A Lua file generated by ONEVIEW for ASSIST.
-I=<path/to/includes>[,<other/path>,...]	Add all directories to be included, multiple directories must be separated by a coma, no space. This option is recursive and include all sub-directories.
-E=<path/to/exclude/files>[,<other/path>,...]	Write all files to be excluded want to analyze and transform; separate them by a coma, no space.
-generate-descriptor	Generate a transformation script template.
-config=<path/to/file>	To use the transformation script.
-replace=[<new name>]	The new output name and location. If not filled, the file will be moved at the current location

	with the same name as the input file.
-generateAllRMOD=<path/to/dir>	Extract all modules of Fortran files in a directory and sub directories and create rmod files which represent the header of each module (if exists) under the name: <module_name>.rmod
-removeAllRMOD=<path/to/dir>	Remove all rmod files in a directory and sub directories.
-f2008	Comment all Fortran2008 features framed between #ifdef F2008 ... #endif macro in the code. Used to handle fortran codes where f08 featutres have not to be modified.
-handle_macro	For a better management of C/C++ macro. Rose can forget to restore all macro. This option add a pass to check after transformations that all macro are well restored.
-option=<options>	Available options are:
"apply-directives"	Search all MAQAO directives in files and apply corresponding transformation.
"vprofcalltrans"	Add a call to the vprof lib at the beginning of each function on all integer parameter. Use for auto-specialization.
"generatePDF"	Generate a PDF file with all AST nodes with the ROSE representation.
"generateDOT"	Generate a .dot file which can be transformed into a graph with Graviz.
Metrics Comparator	
–create-auto-config	Create a configuration file template named auto-config.lua, used for the semi-auto feature.
–semi-auto=<path/to/configfile> [<options>]	Execute the semi-automatic feature using a specific configuration file. Additional options can be added :
–max-loop=<number>	The nth first hottest loops to compare
–output=<path/to/file>	Comparative table is written in a file instead of printed on t

	standard output
–xp-dir=<path/to/xpdir>	Move or rename the experiment directory. By default the name is : "maqao_s2s_ov_results_<root_dir_name>"
–let-tags	Do not remove labels in source code after the analyze.
–options="apply-directives"	Apply only directives in source code instead of use MAQAO metrics.
–clean-auto=<path/to/dir_Vx>	Clean all created intermediate sources directories.
–clean-after[=<N>]	Clean all versions of source code above N. By default, only the last version is cleaned.
–ov-compare –ovdir1=<ov1> –ovdir2=<ov2> [options]	Compare two OneView directories.
	Available options are :
–max-loop=<number>	Set the maximum of loop compared.
–loop-time-min=<number>	Set the minimum coverage of loop to compare in percent. By default, set at 1.5%.
–output=<path_to_file>	The comparative table is written in a file instead of the standard output.

A.2 ASSIST Comparator Configuration file

The configuration file is used to indicate what to automatically modify and where are information needed. To create a template the following command can be used : $maqao s2s –create-auto-config. A configuration template is then create named "auto-config.lua" by default. Each field has a description inside the configuration file.

- Makefile : The path to the Makefile
- Makefile_options : If the Makefile requires specific options.
- src : path to source files root.
- loop_time_min : Set the minimum coverage of loop to compare in percent. By default, set at 1.5%.
- Bin_oneview_path : If the user want to use a different maqao binary.
- report : Indicate the level of report that ONEVIEW have to use. By default it have three level named : "one", "two" or "three", but personalized report can be added. If the field is not filled, the report "two" is taken.

- Oneview_config_file : The path to ONEVIEW configuration file containing experiment parameters. The OneView file can be generated using the following command : "$maqao oneview –create-config".

A.3 Metrics Used for the Comparator

the ONEVIEW global metrics :

- the total time of the program execution in seconds;
- the flow complexity with the average number of paths in the loops;
- the array access efficiency;
- the speedup "if clean", if all instructions performing addresses computations and scalar integer computations have been deleted;
- the number of loops to optimize to get 80% of the speedup;
- the speedup "if floating points are vectorized", an optimistic speedup if all floating point instructions are vectorized;
- the number of loops needed to get 80% if floating points are vectorized;
- the speedup "if fully vectorized", an optimistic speedup if all instructions are vectorized;
- the number of loops needed to get 80% if fully vectorized;
- the speedup "if data are in the L1 Cache", an optimistic speedup if all instructions fits in the L1 cache;
- the number of loops needed to get 80% if data are in the L1 Cache;
- compilation options used.

CQA compared metrics are :

- Bottlenecks : Table of detected bottlenecks.
- Unroll confidence level : Confidence level about unroll information.

- Cycles L1 "if clean" : Loop throughput (number of cycles per iteration) if all data are in L1 and scalar integer instructions removed.
- Cycles L1 "if fully vectorized" : Loop throughput (number of cycles per iteration) if all data are in L1 and fully vectorized.
- "Vector-efficiency ratio all" : Vector efficiency ratio (average proportion of used vector length) of instructions processing FP or integer elements.
- "Vectorization ratio all" : Vectorization ratio (proportion of vectorizable instructions that was vectorized) of instructions processing FP or integer elements.
- "FP op per cycle L1" : FLOPs per cycle if all data are in L1.
- "Speedup if clean" : speedup of instructions if clean (Cycles L1 / Cycles L1 if clean).
- "Speedup if fully vectorized" : speedup of instructions if fully vectorized (Cycles L1 / Cycles L1 if fully vectorized).

A.4 Installation Requirements

Before to use ASSIST, some requirements are needed. The list below shows the required packages and libraries.

- Openjdk-6-jdk: version 6 or higher (not tested with older versions). Then you have to add the path to the libjvm.so in your LD_LIBRARY_PATH.
- ROSE libraries, available at git.maqao.org:S2S/LIBS.git
- Boost Libraries, available at git.maqao.org:S2S/LIBS.git

ASSIST can be downloaded from the git, as well as the required libraries and a pre-compiled binary :

- binary (containing MAQAO, ASSIST and all libraries) : `git clone git@git.maqao.org:S2S/RELEASE.git`.
- Sources : `git clone git@git.maqao.org:S2S/S2S.git`.
- Libraries required : `git clone git@git.maqao.org:S2S/LIBS.git`

A.5 How to Use ASSIST

This section presents how to use ASSIST, by giving examples and a description of different existing commands; all the commands can be found ins appendix A.1.

A.5.1 With an Annotated Source File

One common way to use ASSIST is to provide a (or multiple) source file(s) annotated by the user with specific directives to guide ASSIST in its transformations. In this case the user assumes transformations that will be performed (if possible). The following command will look for directives in the source file(s) and apply corresponding transformations.

```
maqao s2s -option="apply-directives" -src=<path/to/file>[,<path/to/file2>,...]
```

In this case, the user wants to apply all transformations according to custom directives contained in a source file. If the file to analyze requires files to be included, the option "-I" takes a list of paths separated by a coma as followed. This option is recursive and also includes all sub-folders.

```
maqao s2s -option="apply-directives" -src=<path to file> -I=<path to includes>
```

A.5.2 With Profilers Results

If a user does not know what to do to optimize his code he can use metrics and analysis provided by MAQAO performance evaluation tools. These tools include VPROF, a value profiler, CQA, a code quality analyzer, DECAN, a binary modificator and comparator. These tools are managed by ONEVIEW, another MAQAO module which can execute MAQAO modules automatically and can generate a file with all metrics specifically arranged for ASSIST. If in the list of files one cannot be parsed, it can be excluded by using the "-E" option and not block the process.

```
maqao s2s -oneview=<assist.lua> -E=<filetoexclude>
```

A.5.3 Transformation Script

The transformation script is a Lua file with all information about loops and function that have to be modified. It allows not to add directives in the source code but still guides to

apply transformations. A template of this file can be generated as follows:

```
maqao s2s -generate-descriptor
```

To fill this file, you should look first the documentation about it. After completing all fields, you can run the following command to apply transformations:

```
maqao s2s -config=<path to the descriptor file>
```

The transformation Script will be developed in a following section.

A.6 Transformation Script

ASSIST can take as input a file which contains all information to transform a specific file; the next example is a generated template and it is composed as follow:

```
File="<path/to/the/src/file>"
Arch= {
  All = {
    Loops = {
      {line = <line>, transformation = {"<trans>"}},
      {line = <line>, transformation = {"<othertrans>"}},
      {label = "<label>", transformation = {"<trans>"}}
    }
    Functions = {
      {line = <line>, transformation={"<trans1>","<trans2>",...}},
      ...
    }
  }
  x86 = {
    ...
  }
}
```

The script starts by the field File which is the path to the file or at least the path from where you execute ASSIST to the file. The field "Arch" contains all architecture you want to handle. The user can name all "arch" field with the name he wants, he just has to call with the option -arch=<archName> to apply the transformation contained in the field of his defined "arch". By default, transformation contained in "All" will be apply.

In all architectures we can find two fields: “Loops” and “Functions”, which respectively represent all loops and functions in the file. They are defined by their first line or a label above the statement, and transformations to apply on it. The label must be defined as a directive (!DIR$ or #pragma according to the language used) and looks like: !DIR$ MAQAO <label>; <label> must exactly correspond to the label in the configuration file, no sub-string or regular expression are allowed. If they are multiple transformations to apply, they must be separated by a coma. Transformations will be applied in the order, from last to first. <trans> represents the transformation to apply, available values are the same as for directives.

A.7 ASSIST API

ASSIST has been developed as a wrapper of Rose with a complete API to manipulate objects like loops and functions. This API is flexible; it contains all transformations already implemented and allows to add new ones to existing structures, while leaving the possibility of using the Rose API.

A.8 Example of OneView Report Generated for ASSIST

Figure A.1 presents an example of file generated by Oneview for ASSIST. It contains raw data from the others MAQAO modules about the execution of the program. This example comes from POLARIS.

OneView is global module of MAQAO. It is used to execute and gather data and metrics from all MAQAO modules previously presented. A report can be generated for ASSIST to give all information gathered by different modules executed for a specific binary. Figure A.1 presents an example of this report. This report contains two main tables; the first one: "oneview_global_metrics" contains information on the whole application with a global view: the execution time of the dataset, options used to compile the program and other metrics from CQA. The second table: "oneview_report" contains, for each entry, information and metrics by loop. On one side, locations information, with the source file and first/last lines

of the loop; and on the other side, metrics from performance analysis tools, with the minimum, maximum and average number of iterations, the vectorization ratio, and minimum, maximum and average ratio of the original version of the loop compared to its equivalence if data fits in L1 cache.

By using these metrics for each loop we can determine which transformation is most likely to bring performance gain and if a doubt persists or if several choices seem to us legitimate, we ask the user if our choice seems correct to him or if he wants to make another transformation or even none.

A.9 Caveats & Limitations

Even if we improved some of ROSE limitations, Rose still being a research project and ASSIST being a prototype, there is still room for improvements.

A.9.1 Preprocessor

First, we will talk about preprocessors directives. We must differentiate two cases: the first one is for preprocessor statement in Fortran; they will not be taken in charge. The Fortran frontend in Rose is a not well-tested frontend and it is not intended to taken into account type C directives. In this case, the file has to be preprocessed before transformation. The second case is for C/C++; Rose was designed as a compiler so it needs to know if the body of a "#ifdef" statement will be used (or not). If we analyze the body of the "#ifdef" statement we will not analyze the "#else" part (and vice versa). We worked to restore all the elements of each part of the directive in the output file, but the user will be able to modify only one part at a time.

A.9.2 Languages

The Rose frontend is the OpenFortranParser, a Java parser for Fortran. We modified many things in Rose to better handle Fortran statements, but we did not add the management of all keywords and new statements available in Fortran2008 due to the amount of work that would be involved. The file to transform has to be included between Fortran77 and Fortran2003. We have the same kind of problem in C++, Rose handle C++ up to C++03.

```
oneview_global_metrics = {
   total_time_s = 74,
   compilation_options = "binary: -Xhost or -xCORE-<> is missing.",
   flow_complexity = 1.01,
   array_efficieny = 64.05,
   speedup_if_clean = 1.00,
   nb_loop_80_if_clean = 1,
   speedup_if_fp_vect = 1.02,
   nb_loop_80_if_fp_vect = 1,
   speedup_if_fully_vect = 1.22,
   nb_loop_80_if_fully_vect = 10,
}

oneview_report = {
   {
      loop_id = 6916,
      lineStart = 16,
      lineStop = 18,
      file = "mom7.f90",
      ite_min = 60, ite_max = 60, ite_avg = 60,
      vecRatio = 12.5,
      dl1_ratio_min = 1.2, dl1_ratio_max = 1.8, dl1_ratio_avg = 1.6,
   },
   {
      loop_id = 6147,
      lineStart = 867,
      lineStop = 871,
      file = "fmm-dipole.f90",
      ite_min = 15, ite_max = 15, ite_avg = 15,
      vecRatio = 50,
      dl1_ratio_min = 0.4, dl1_ratio_max = 2.2, dl1_ratio_avg = 0.8,
   },
   . . .
}
```

Figure A.1: Example from POLARIS of Oneview internal report for ASSIST, with on one side global metrics and on the other, the "oneview_report" with all metrics by loops

Appendix B

Appendix: Codes

B.1 Prefetcher

The following code is used to modify the Model Specific Register (MSR) behavior. By calling the function "prefetch" with an hexadecimal number between 0 and f the function modify the corresponding register to the right value. It requires to be in sudo mode to modify the register.

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <getopt.h>
#include <inttypes.h>
#include <sys/types.h>
#include <dirent.h>
#include <ctype.h>

/* filter out ".", "..", "microcode" in /dev/cpu */
int dir_filter(const struct dirent *dirp) {
  if (isdigit(dirp->d_name[0])) {
    return 1;
  } else {
    return 0;
  }
}
```

```
void wrmsr_on_cpu (uint32_t reg, int cpu, int valcnt, char *regvals[]) {
  uint64_t data;
  int fd;
  char msr_file_name[64];

  sprintf(msr_file_name, "/dev/cpu/%d/msr", cpu);
  fd = open(msr_file_name, O_WRONLY);
  if (fd < 0) {
    if (errno == ENXIO) {
      fprintf(stderr, "wrmsr: No CPU %d\n", cpu);
      exit(2);
    } else if (errno == EIO) {
      fprintf(stderr, "wrmsr: CPU %d doesn't support MSRs\n", cpu);
      exit(3);
    } else {
      printf("error wrmsr: open");
      perror("wrmsr: open");
      exit(127);
    }
  }
  while (valcnt--) {
    data = strtoull(*regvals++, NULL, 0);
    if (pwrite(fd, &data, sizeof data, reg) != sizeof data) {
      if (errno == EIO) {
        fprintf(stderr,
                "wrmsr: CPU %d cannot set MSR "
                "0x%08"PRIx32" to 0x%016"PRIx64"\n",
                cpu, reg, data);
        exit(4);
      } else {
        perror("wrmsr: pwrite");
        exit(127);
      }
    }
  }
  close(fd);
  return;
}
```

```
void wrmsr_on_all_cpus(uint32_t reg, int valcnt, char *regvals[]) {
  struct dirent **namelist;
  int dir_entries;

  dir_entries = scandir("/dev/cpu", &namelist, dir_filter, 0);
  while (dir_entries--) {
    wrmsr_on_cpu(reg, atoi(namelist[dir_entries]->d_name),
                 valcnt, regvals);
    free(namelist[dir_entries]);
  }
  free(namelist);
}

void prefetch_(char *s) {
  wrmsr_on_all_cpus(420, 1, &s);
}
```

B.2 Intel Optimization Directives/Pragmas

The Intel compiler provides a wide range of directives and pragmas , especially to give hints to the compiler and drive its optimization choices. Table B.1 presents some of these available directives and pragmas with the Intel Compiler. Currently, only the "loop count" is handled by ASSIST but we could add others if we find how to trigger them.

Directive \ Pragma	Definition
ASSUME	Provides heuristic information to the compiler optimizer.
BLOCK_LOOP & NOBLOCK_LOOP	Enables or disables loop blocking for the immediately following nested DO loops.
FMA & NOFMA	Enables (or disables) the compiler to allow generation of fused multiply-add (FMA) instructions, also known as floating-point contractions.
INLINE, FORCEINLINE, NOINLINE	Tells the compiler to perform the specified inlining on routines within statements or DO loops.
IVDEP	Assists the compiler's dependence analysis of iterative DO loops.
LOOP COUNT	Specifies the typical trip count for a DO loop; this assists the optimizer.
NOFUSION	Prevents a loop from fusing with adjacent loops.
OPTIMIZE & NOOPTIMIZE	Enables or disables optimizations for the program unit.
PREFETCH & NOPREFETCH	Enables or disables hint to the compiler to prefetch data from memory.
SIMD	Requires and controls SIMD vectorization of loops.
UNROLL & NOUNROLL	Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop.
UNROLL_AND_JAM & NOUNROLL_AND_JAM	Enables or disables loop unrolling and jamming.
VECTOR & NOVECTOR	Overrides default heuristics for vectorization of DO loops.

Table B.1: Non-exhaustive list of optimization directives and pragmas available with the Intel Compiler. Sources : https://software.intel.com/en-us/node/524560#EE255A8D-F0AC-4022-A6C0-DA92E6BFC8CE, https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-compiler-directives

Appendix C

Appendix: Additional results

C.1 Prefetchers

The section 3.2 in Chapter 3 presents how transformations are triggered. This section presents experimental results obtained on Yales2 with the test case 3D_Cylinder. The figure C.2 is an array of speedups of the thirteen hottest functions for each prefetcher behavior compared to the version where all prefetchers are disabled. Figure 4.15 presents these data. Green boxes represent positives speedups, greys represent equivalent speedup and red ones represent the negatives ones.

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
► calc_data_gradient	3D_cylinder_orig_decan_v2_icc17	19.54	19.34	1	0.00
► ics_advance_velocity_tfv4a_4th	3D_cylinder_orig_decan_v2_icc17	10.37	10.26	1	0.00
► calc_data_tridiag_op_product	3D_cylinder_orig_decan_v2_icc17	8.85	8.76	1	0.00
► filter_real_data	3D_cylinder_orig_decan_v2_icc17	8.47	8.38	1	0.00
► update_int_comm	3D_cylinder_orig_decan_v2_icc17	8.28	8.2	1	0.00
► split_one_el_grp	3D_cylinder_orig_decan_v2_icc17	5.13	5.08	1	0.00
► adv_scalar_w_u_tfv4a_4th	3D_cylinder_orig_decan_v2_icc17	4.57	4.52	1	0.00
► solve_linear_system_deflated_pcg	3D_cylinder_orig_decan_v2_icc17	4.51	4.46	1	0.00
► copy_int_comm_to_data	3D_cylinder_orig_decan_v2_icc17	4.1	4.06	1	0.00
► copy_data_to_int_comm	3D_cylinder_orig_decan_v2_icc17	2.32	2.3	1	0.00
► cvt_text_to_data	3D_cylinder_orig_decan_v2_icc17	1.6	1.58	1	0.00

Figure C.1: Oneview view of functions managed.

Functions \ MSR	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Exec time	1	0.81	0.99	0.57	0.98	0.80	0.84	0.54	0.87	0.48	0.85	0.49	0.81	0.54	0.89	0.52
calc_data_gradient	1	0.79	0.99	0.55	1.01	0.77	0.87	0.47	0.89	0.44	0.85	0.51	0.83	0.52	0.87	0.47
ics_advance_velocity_tfv4a_4th	1	0.89	0.99	0.67	0.96	0.85	0.85	0.64	0.87	0.46	0.85	0.43	0.84	0.51	0.90	0.47
calc_data_tridiag_op_product	1	0.80	1.06	0.52	1.01	0.80	0.84	0.52	0.86	0.32	0.86	0.36	0.80	0.42	0.89	0.35
filter_real_data	1	0.68	1.02	0.42	0.96	0.69	0.89	0.43	0.89	0.43	0.90	0.40	0.87	0.45	0.92	0.44
update_int_comm	1	0.89	0.90	0.61	0.97	0.80	0.63	0.60	0.65	0.56	0.66	0.59	0.58	0.62	0.73	0.64
split_one_el_grp	1	0.94	0.97	0.89	1.00	0.92	0.98	0.88	0.92	0.82	0.95	0.81	0.90	0.75	0.92	0.62
copy_int_comm_to_data	1	0.94	1.08	0.86	0.89	1.02	0.94	0.94	1.00	0.95	0.92	0.97	1.00	0.86	1.07	1.02
solve_linear_system_deflated_pcg	1	0.72	0.96	0.45	0.92	0.70	0.70	0.46	0.79	0.46	0.77	0.47	0.78	0.55	1.00	0.69
adv_scalar_w_u_tfv4a_4th	1	0.93	1.00	0.64	1.00	0.90	0.82	0.63	0.82	0.34	0.78	0.38	0.77	0.44	0.80	0.42
compute_bnd_elem_values	1	1.09	1.30	0.59	1.07	0.89	0.66	0.50	0.95	0.61	1.07	0.85	0.87	0.85	0.92	0.77
copy_data_to_int_com	1	0.87	1.14	0.70	1.03	1.43	0.75	0.65	0.73	0.65	0.72	0.73	0.78	0.82	0.91	0.99
set_real_data_to_constant	1	0.45	1.13	0.27	1.04	0.48	1.07	0.28	1.20	0.34	1.08	0.28	1.03	0.33	1.00	0.42
cvt_text_to_data	1	0.83	1.13	0.90	0.98	0.88	0.95	1.03	0.85	1.15	0.92	0.79	0.95	1.00	1.07	0.94

Figure C.2: Speedup by function for each prefetcher behavior.

Bibliography

[1] J. K. Hollingsworth A. Tiwari. "End-to-end Auto-tuning with Active Harmony". In: Performance Tuning of Scientific Applications. Chapman and Hall/CRC Computational Science Series, 2010.

[2] ABINIT. https://www.abinit.org/.

[3] L. Adhianto et al. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs". In: vol. 22. 6, pp. 685–701. doi: 10.1002/cpe.1553. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1553. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1553.

[4] Advisor. https://software.intel.com/en-us/intel-advisor-xe.

[5] Kim et al. "Multi-level tiling M for the price of one". In: ACM/IEEE conference on Supercomputing. ACM, 2007, p. 51.

[6] Mehdi AMINI. "Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators". In: PhD Thesis. Paris, France, 2012.

[7] AnandTech. https://www.anandtech.com/.

[8] L. Andersien. "Program Analysis and Specialization for the C Programming Language". In: Ph.D thesis. University of Copenhagen: DIKU, 1994.

[9] APS. https://www.alcf.anl.gov/files/velesko_vtune_may.pdf.

[10] ATLAS. http://math-atlas.sourceforge.net/.

[11] AVBP. http://www.cerfacs.fr/avbp7x/.

[12] D. Barthou et al. "Performance Tuning of x86 OpenMP Codes with MAQAO". In: Parallel Tools Workshop. Desden, Germany, 2009, pp. 95–113.

[13] Cedric Bastoul. "Code generation in the polyhedral model is easier than you think". In: International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, 2004, pp. 7–16.

[14] Z. Bendifallah et al. “PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis”. In: Tools for High Performance Computing 2013: Proceedings of the 7th International Workshop on Parallel Tools for High Performance Computing. Springer International Publishing, 2014, pp. 107–127.

[15] Zakaria Bendifallah. “Generalization of the decremental performance analysis to differential analysis”. Theses. Université de Versailles-Saint Quentin en Yvelines, Sept. 2015. url: https://tel.archives-ouvertes.fr/tel-01293039.

[16] A. D. Biagios. [llvm-dev] [RFC] llvm-mca: a static performance analysis tool. url: {https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html}.

[17] J. Bilmes et al. “Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology”. In: ACM International Conference on Supercomputing. ACm, 2014, pp. 253–260.

[18] U. Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: ACM SIGPLAN Notices. ACM, 2008, pp. 101–113.

[19] J. Brant and D. Roberts. “The SmaCC Transformation Engine”. In: OOPSLA ’09. Orlando, Florda, USA, 2009.

[20] Martin Burtscher et al. “PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications”. In: Nov. 2010, pp. 1–11. doi: 10.1109/SC.2010.41.

[21] callgrind. http://kcachegrind.github.io/html/Home.html.

[22] A. S. Charif-Rubial et al. “MIL: A language to build program analysis tools through static binary instrumentation”. In: 20th Annual International Conference on High Performance Computing. 2013, pp. 206–215.

[23] A.S. Charif-Rubial et al. “CQA: A code quality analyzer tool at binary level”. In: HiPC. IEEE Computer Society, 2014, pp. 1–10.

[24] C. Chen, J. Chame, and M. Hall. “CHiLL: A Framework for Composing High-Level Loop Transformations”. In: 2008.

[25] D. Chen, X. David Li, and T. Moseley. “AutoFDO: Automatic Feedback-directed Optimization for Warehouse-scale Applications”. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. CGO ’16. New York, NY, USA: ACM, 2016. isbn: 978-1-4503-3778-6.

[26] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. “Automatic Source Code Specialization for Energy Reduction”. In: Proceedings of the 2001 International Symposium on Low Power Electronics and Design. ISLPED ’01. Huntington Beach, California, USA: ACM, 2001, pp. 80–83. isbn: 1-58113-371-5.

[27] James R. Cordy. “Source Transformation, Analysis and Generation in TXL”. In: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. PEPM ’06. New York, NY, USA: ACM, 2006, pp. 1–11.

[28] Coria. http://www.coria-cfd.fr/index.php/YALES2.

[29] cube. http://www.scalasca.org/software/cube-3.x/.

[30] Dave and al. “Cetus: A source-to-source compiler infrastructure for Multicores”. In: Computer. 2009, pp. 36–42.

[31] R. Dolbeau, S. Bihan, and F. Bodin. “HMPP: A hybrid multi-core parallel programming environment”. In: Workshop on general purpose processing on graphics processing units (GPGPU 2007). Vol. 28. 2007.

[32] S. Donadio and al. “A Language for the Compact Representation of Multiple Program Versions”. In: International Workshop on Languages and Compilers for Parallel Computing. springer, 2005, 136–151.

[33] J. J. Dongarra et al. “a set of level 3 basic linear algebra subprograms”. In: ACM Transactions on Mathematical Software. ACM, 1990, pp. 1–17.

[34] Agner Fog. https://www.agner.org/.

[35] Agner Fog. https://www.agner.org/optimize/microarchitecture.pdf.

[36] M. Frigo and S.G. Johnson. “FFTW: An adaptive software architecture for the FFT”. In: Processings of the ICASSP Conference. Nowhere: void, 1998, p. 1381.

[37] J. Labarta G. Ozen E. Ayduade. “MACC: Mercurium ACCeletator Model”. In: International Workshop on OpenMP. Universitat Politecnica de Catalunya, Barcelona, Spain: springer, 2014.

[38] gcov. https://en.wikipedia.org/wiki/Gcov.

[39] Markus Geimer et al. “The SCALASCA Performance Toolset Architecture”. In: International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece. 2008, pp. 51–65.

[40] S. Girbal and al. “Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies”. In: International Journal of Parallel Programming. International Journal of Parallel Programming, 2006, pp. 261–317.

[41] GNU. http://www.gnu.org.

[42] X. Gonze and al. “ABINIT: First-principles approach to material and nanosystem properties”. In: Computer Physics Communications. Elsevier, 2009, pp. 2582–2615.

[43] Google. https://github.com/google/autofdo.

[44] M. W. Hall and al. “Maximizing Multiprocessor Performance with the SUIF Compiler”. In: IEEE Computer, 1996.

[45] Julian Hammer et al. “Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels”. In: vol. abs/1702.04653. 2017. arXiv: 1702.04653. url: http://arxiv.org/abs/1702.04653.

[46] A. Hartono, B. Norris, and P. Sadayappan. “Annotation-based empirical performance tuning using Orio”. In: 2009 IEEE International Symposium on Parallel Distributed Processing. 2009, pp. 1–11.

[47] S. Henry, H. Bollore, and E. Oseret. “Towards the Generalization of Value Profiling for High-Performance Application Optimization”. In: http://www.hsyl20.fr/home/files/papers/shenry_2015_vprof.pdf.

[48] IACA. https://software.intel.com/en-us/inproceedingss/intel-architecture-code-analyzer.

[49] Jonathan Rentzsch (IBM). https://www.ibm.com/developerworks/library/pa-dalign/.

[50] Irigoin and al. “Interprocedural Analyses for Programming Environments”. In: Workshop on Environments and Tools For Parallel Scientific Computing. Saint-Hilaire du Touvier, France, 1992.

[51] O. Rüthing J. Knoop and B. Steffen. “Partial dead code elimination”. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. New York, NY, USA: ACM Press., 1994, 147–158.

[52] W. Jalby et al. “The Long and Winding Road Toward Efficient High-Performance Computing”. In: vol. 106. 11. 2018, pp. 1985–2003. doi: 10.1109/JPROC.2018.2851190.

[53] B. Norris K. Meng. “Mira: A Framework for Static Performance Analysis”. In: Cluster Computing (CLUSTER). Honolulu, HI, USA: IEEE, 20017, pp. 103–113. isbn: 978-1-5386-2326-8.

[54] P. Klint, T. van der Storm, and J. Vinju. “RASCAL A Domain Specific Language for Source Code Analysis ad Manipulation”. In: IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, 2009, pp. 168–177.

[55] Andreas et al. Knüpfer. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir”. In: Tools for High Performance Computing 2011. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[56] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: vol. abs/1801.01203. 2018. arXiv: 1801.01203. url: http://arxiv.org/abs/1801.01203.

[57] S. Koliaï et al. “Quantifying Performance Bottleneck Cost Through Differential Analysis”. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. ICS ’13. Eugene, Oregon, USA: ACM, 2013, pp. 263–272. isbn: 978-1-4503-2130-3.

[58] Souad Koliaï. “Static and dynamic approach for performance evaluation of scientific codes”. Theses. Université de Versailles-Saint Quentin en Yvelines, 2011.

[59] O. Krzikalla and al. “Scout: A Source-to-Source Transformator for SIMD-Optimizations”. In: Euro-Par. Springer, 2012, pp. 137–145.

[60] L.. et al. “Automatic configuration of GCC using irace”. In: Artificial Evolution. 2017, pp. 202–216.

[61] S. Larsen, E. Witchel, and S. Amarasinghe. “Techniques for Increasing and Detecting Memory Alignment”. In: 2001.

[62] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. “When Prefetching Works, When It Doesn’t, and Why”. In: vol. 9. Mar. 2012, pp. 1–29. doi: 10.1145/2133382.2133384.

[63] Jean Baptiste Lereste and Andres S. Charif-Rubial. https://www.maqao.org/release/MAQAO.Tutorial.LProf.v1.pdf.

[64] S. Liao et al. “Machine learning-based prefetch optimization for data center applications”. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. 2009, pp. 1–10.

[65] Moritz Lipp et al. “Meltdown”. In: vol. abs/1801.01207. 2018. arXiv: 1801.01207. url: http://arxiv.org/abs/1801.01207.

[66] G. Llort et al. “On the usefulness of object tracking techniques in performance analysis”. In: SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2013, pp. 1–11. doi: 10.1145/2503210.2503267.

[67] LLVM. https://llvm.org/.

[68] P Lu et al. “PIT: A Framework for Effectively Composing High-Level Loop Transformations”. In: Computing and Informatics. Open Journal Systems, 2012, pp. 943–963.

[69] R. vermaas M. Bravenboer K. T. Kalleberg and E. Visser. “Stratego/XT 0.17. A language and toolset for program transformation”. In: Science of Computer Programming. Elsevier, 2008.

[70] A. Mandal and al. “Using Dynamic Compilation to Achieve Ninja Performance for CNN Training on Many-Core Processors”. In: Europar. IEEE, 2018.

[71] Rose user manual. http://www.rosecompiler.org/ROSE_UserManual/ROSE-0.9.7.161-UserManual.pdf.

[72] MSR-Tools. https://01.org/msr-tools.

[73] Harm et al. Munk. "ACOTES Project: Advanced Compiler Technologies for Embedded Streaming". In: vol. 39. 3. June 2011, pp. 397–450.

[74] R. Muth, S. Watterson, and S. Debray. "Code Specialization based on Value Profiles". In: International Static Analysis Symposium. Springer, 2000, pp. 340–359.

[75] G. C. Necula et al. "CIL: Intermediate language and tools for analysis and transformation of C programs". In: International Conference on Compiler Construction. University of California, Berkeley, USA: Springer, 2002, pp. 213–228.

[76] Diego Novillo. "SamplePGO: The Power of Profile Guided Optimizations Without the Usability Burden". In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC. LLVM-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 22–28. isbn: 978-1-4799-7023-0.

[77] OpenACC. https://www.openacc.org/.

[78] OpenC++. http://opencxx.sourceforge.net/.

[79] OpenMP. https://www.openmp.org/.

[80] GCC Instrumentation Options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#Instrumentation-Options.

[81] GCC Optimizations Options. https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html#Optimize-Options.

[82] PGO Overview. https://software.intel.com/en-us/node/522721.

[83] J.N. Amaral P. Berube. "Aestimo: a feedback-directed optimization evaluation tool". In: Austin, TX, USA, USA: IEEE, 2006. isbn: 1-4244-0186-0.

[84] Marek Palkowski and Wlodzimierz Bielecki. "TRACO Parallelizing Compiler". In: Soft Computing in Computer and Information Science. Ed. by Antoni Wiliński, Imed El Fray, and Jerzy Pejaś. Springer International Publishing, 2015, pp. 409–421.

[85] Vincent Palomares. "Combining static and dynamic approaches to model loop performance in HPC". In: 2015.

[86] Maksim Panchenko et al. "BOLT: A Practical Binary Optimizer for Data Centers and Beyond". In: vol. abs/1807.06735. 2018. arXiv: 1807.06735. url: http://arxiv.org/abs/1807.06735.

[87] P. Ranjan Panda et al. “A data alignment technique for improving cache performance”. In: Proceedings International Conference on Computer Design VLSI in Computers and Processors. 1997, pp. 587–592.

[88] Paraformance. http://paraformance.weebly.com/.

[89] PerfExpert. https://www.tacc.utexas.edu/research-development/tacc-projects/perfexpert.

[90] D. Plotnikov et al. “An Automatic tool for tuning compiler optimizations”. In: Ninth International Conference on Computer Science and Information Technologies Revised Selected Papers. 2013, pp. 1–7.

[91] Dmitry Plotnikov et al. “Automatic Tuning of Compiler Optimizations and Analysis of their Impact”. In: vol. 18. 2013 International Conference on Computational Science. 2013, pp. 1312 –1321.

[92] Mihail Popov et al. “Piecewise Holistic Autotuning of Parallel Programs with CERE”. In: vol. 29. 15. Wiley, 2017, e4190. url: https://hal-uvsq.archives-ouvertes.fr/hal-01542912.

[93] Allan Porterfield. “Software Methods for Improvement of Cache Performance on Supercomputer Applications”. PhD thesis. 1989.

[94] L.-N. Pouchet et al. “Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time”. In: ACM SIGPLAN Notices. ACM, 2008, pp. 90–100.

[95] LLVM PGO presentation. https://llvm.org/devmtg/2013-11/slides/Carruth-PGO.pdf.

[96] William H. Press et al. Numerical Recipes 3rd Edition: The Art of Scientific Computing. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. isbn: 0521880688, 9780521880688.

[97] M. Puschel and al. “SPIRAL: Code generation for DSP transforms”. In: Proceedings of the IEEE. IEEE, 2005, pp. 216–231.

[98] QMCPack. https://www.qmcpack.org/.

[99] Quinlan and al. “ROSE: Compiler Support for Object-Oriented Framework”. In: Parallel Processing Letters. Lawrence Livermore National Laboratory, Livermore, CA, USA: World Scientific, 2000, pp. 215–226.

[100] A. Petitet R. Clint Whaley and J. J. Dongarra. “Automated empirical optimizations of software and the ATLAS project”. In: Parallel Computing. Elsevier Science, 2000, pp. 3–35.

[101] Gang Ren et al. “Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers”. In: 2010, pp. 65–79. url: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68.

[102] T. Schonfeld and M. Rudgyard. “Steady and Unsteady Flow SimulationsUsing the Hybrid Flow Solver AVBP”. In: AIAA Journal. AIAA ARC, 1999, pp. 1378–1385.

[103] Sameer S. Shende and Allen D. Malony. “The Tau Parallel Performance System”. In: vol. 20. 2. Thousand Oaks, CA, USA: Sage Publications, Inc., 2006, pp. 287–311. doi: 10.1177/1094342006064482. url: http://dx.doi.org/10.1177/1094342006064482.

[104] S. Srinath et al. “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”. In: IEEE 13th International Symposium on High Performance Computer Architecture. 2007, pp. 63–74. doi: 10.1109/HPCA.2007.346185.

[105] Reiji Suda, Hiroyuki Takizawa, and Shoichi Hirasawa. “Xevtgen: Fortran code transformer generator for high performance scientific codes”. In: International Journal of Networking and Computing. 2016, pp. 263–289.

[106] W.J. Tan and al. “A Code Generation Framework for Targeting Optimized Library Calls for Multiple Platforms”. In: IEEE Transactions on parallel and distributed systems. University of Singapore, China, 2014, vol 26, No 7.

[107] Thiago S. F. X. Teixeira et al. “Locus: A System and a Language for Program Optimization”. In: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 217–228.

[108] A. Tiwari and al. “A Scalable Auto-tuning Framework for Compiler Optimization”. In: Parallel and Distributed Processing. IEEE, 2009, pp. 1–12.

[109] MAQAO toolsuite. http://www.maqao.org.

[110] Rose tutorial. http://rosecompiler.org/ROSE_Tutorial/ROSE-Tutorial.pdf.

[111] Cédric Valensi. “A generic approach to the definition of low-level components for multi-architecture binary analysis”. Thesis. Université de Versailles-Saint Quentin en Yvelines, 2014.

[112] vampir. https://vampir.eu/.

[113] S. Verdoolaege and al. “Polyhedral Parallel Code Generation for CUDA”. In: ACM Trans. Architec. Code Optim. ACM, 2013.

[114] Chris Lattner et Vikram Adve. “DMS/spl reg: program transformations for practical scalable software evolution”. In: Software Engineering, ICSE 2004. Proceedings. 26th International Conference on. IEEE, 2004, pp. 625–634.

[115] Chris Lattner et Vikram Adve. “LLVM A compilation framework for lifelong program Analysis and Transformation”. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE, 2004.

[116] VTune. https://software.intel.com/en-us/inproceedingss/intel-compiler-new-feature-hardware-based-pgo.

[117] R. Vuduc, J. W Demmel, and K. A Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: Journal of Physics: Conference Series. IOP Publishing, 2005, p. 521.

[118] Warp3d. http://www.warp3d.net/.

[119] APS website. https://software.intel.com/sites/products/snapshots/application-snapshot/.

[120] Chengyong Wu et al. “An Overview of the Open Research Compiler”. In: Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers. Ed. by Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 17–31.

[121] Wm. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: vol. 23. 1. ACM, 1995, pp. 20–24. doi: 10.1145/216585.216588. url: http://doi.acm.org/10.1145/216585.216588.

[122] X. Xiao et al. “An Approach to Customization of Compiler Directives for Application-Specific Code Transformations”. In: 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs. 2014, pp. 99–106.

[123] Qing Yi. “POET: A Scripting Language For Applying Parameterized Source-to-source Program Transformations”. In: Software Practice And Experience. University of Texas at San Antonio, USA: John Wiley and Sons., 2012, pp. 675–706.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Optimisation de code basée sur des transformations source-à-source guidées par des métriques issues de profilages

Mots clés: Optimisation de code, transformations de code, source-à-source, autotuning, pgo, analyse de code

Résumé: Les processeurs modernes traitent les problèmes de performances en s'appuyant fortement sur une taille de vecteur croissantes et des hiérarchies de mémoire avancées pour offrir de bonnes performances. L'optimisation de code est donc devenue une tâche difficile. Les développeurs font généralement confiance aux compilateurs pour résoudre automatiquement ces problèmes de performances. Cependant, les compilateurs utilisent des modèles de performances statiques et des heuristiques qui les obligent à rester prudents. D'un autre coté, on a des outils d'analyse de performance qui sont très efficaces pour détecter des problèmes spécifiques, mais ils ne renvoient que des observations sur la qualité et l'exécution du code. Le but est de développer d'un outil permettant de réaliser des transformations de code source basées sur des métriques d'outils d'analyse de performances. Cet outil sera intégré à la suite d'outils MAQAO. Nous présentons des transformations source-à-source automatique, guidées par les métriques provenant des différents outils de MAQAO et en restant ouvert aux conseils de l'utilisateur. Cet outil peut aussi servir à simplifier le développement, en permettant d'effectuer des transformations simples, mais chronophage et sources d'erreurs (e.g. spécialisation de boucle/fonction).

Title: Code optimization based on source-to-source transformations using profile guided metrics

Keywords: Code optimization, source-to-source, code transformations, autotuning, pgo, code analyze

Abstract: Modern high performance processor architectures tackle performance issues by heavily relying on increased vector lengths and advanced memory hierarchies to deliver high performance. Manual optimization is became a difficult task. Developers usually trust compilers to automatically address these performance issues, but they deploy static performance models and heuristics that force them to remain conservative. On the other hand, performance analysis tools are pretty good at detecting specific performance issues, but only return observations on the quality and on the execution of the code. Our goal is to develop a framework allowing to perform of source code transformations based on performance analysis tools metrics. This framework will be integrated into the MAQAO tool suite. We present an FDO tool with a set of source-to-source transformations guided by metrics coming from the various MAQAO tools and open to user advices. This framework can also be used to simplify the development by automatically performing some simple, but time-consuming and error-prone transformations (e.g. loop/function specialization).

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France