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

Approche statique et dynamique pour l'évaluation de performances de codes scientifiques

Static and Dynamic Approach for Performance Evaluation of Scientific Codes

THÈSE

présentée et soutenue publiquement le 11 Juillet 2011

pour l'obtention du

Doctorat de l'université de Versailles Saint-Quentin

(spécialité informatique)

par

Souad Koliaï

Composition du jury

LRC IT@CA

Exascale computing research

Remerciements

Quel doctorant n'a pas rêvé du moment de la rédaction des remerciements? Pour ma part, ce jour là est arrivé et je tiens à remercier toutes les personnes m'ayant entourées durant ces quatre dernières années.

Je tiens, premièrement, à remercier mon directeur de thèse, William Jalby, pour m'avoir encadré durant ces quatre années. Je ne connais pas beaucoup de personnes qui ont une telle passion pour leur travail. Sa détermination à vouloir toujours aller plus loin pour comprendre, m'a permis de me rendre compte que "même quand on ne peut plus, on peut encore". Une chose que je retiendrai.

Je tiens ensuite à remercier les membres du jury sans qui ma soutenance n'aurait pu avoir lieu. Tout d'abord, merci à mes rapporteurs : François Bodin et David Levinthal, qui ont pris le soin de relire mon manuscrit et de proposer des améliorations pour mes travaux. Je remercie mes examinateurs : Julia Fedorova, qui a accepté de venir de Russie pour assister à ma soutenance ; Raymond Namyst, président de mon jury, qui s'est déplacé de Bordeaux pour ma soutenance ; Denis Barthou, qui a d'abord été mon enseignant en master et avec qui j'ai collaboré à de nombreuses reprises durant ma thèse.

Je tiens à remercier ceux qui étaient là lorsque je suis arrivée en stage. Tout d'abord Marc Pérache pour m'avoir permis de récupérer un très beau bureau et une machine. Patrick Carribault, qui a soutenu sa thèse à mon arrivée dans l'équipe et qui m'a permis de récupérer une confortable chaise. Sébastien Donadio, qui m'avait accueilli au sein de l'équipe tout d'abord en disant "haha, une fille en HPC !", puis en se rattrapant en me donnant de très bons conseils durant toutes mes années de thèse.

Et puis il y a les collègues. Tout d'abord, Stéphane Zuckerman, avec qui j'ai eu la plaisir de collaborer à de nombreuse reprises, mais qui est surtout un camarade de thèse qui-parle-aux-ordinateurs, avec qui j'ai partagé le bureau durant 3 ans. Non, sans quelques discussions animées qui se terminaient par le fait que "moi j'avais toujours raison"...

Julien Jaeger, avec qui j'étais en master, et qui a aussi été mon co-bureau durant 3 ans. Merci, pour les blagues du matin, les chansons "pas très intelligentes" qui collent à l'esprit et les parties de "tout-le-monde-veut-prendre-sa-place" !

Yuriy, le dernier locataire de mon ancien bureau, qui par sa venue a animé le bureau grâce aux discussions plus qu'animées avec Stéphane sur le communisme ! Merci à la cloche à trolls :)

Le temps est venu pour changer de bureau, de collègues de bureau et de rentrer dans l'ère de l'Exascale ! Une nouvelle aventure, dans un nouveau bureau, avec de nouvelles personnes. Parmi elles, je tiens à remercier Jean-Thomas Acquaviva, de m'avoir encadré lors de ma dernière année de thèse, loin d'être la plus simple. Je le remercie d'avoir supporté mon caractère "robotique" lors de ma rédaction de thèse. Blop ! :)

Un grand merci à Jean Christophe Beyler, que j'ai d'abord rencontré lors de mon stage aux Etats-Unis, et qui est toujours de très bon conseil. Je lui dois la relecture de mon manuscrit, la correction des nombreuses fautes d'anglais avec des commentaires écrits en franco-anglais. JC, okay, "je vire les parenthèse"! :)

Puis mes nouveaux collègues de bureau. Cédric Valensi, monsieur je désassemble, je patch et je me demande pourquoi "la vie est si injuste" :). Cédric, ton tour arrivera et tu auras le plaisir de rédiger des remerciements aussi. Emmanuel Oseret pour la pertinences de toutes ses remarques ; Eric Petit, pour le ping pong verbal et avec qui il fallait toujours être prêt à riposter.

Je tiens à saluer, Bettina Krammer, qui nous rappelait qu'il y avait un livrable à faire pour le mois dernier, ainsi que Grigori Fursin, José Noudohouhenou, Andres Charif-Rubial et tous les collègues du laboratoires Exascale.

Enfin, les crypto-kids : Aurélie Bauer, la doyenne des crypto-thésards ; Sorina et ses courbes elliptiques ; Jean-Michel à cause de qui on dit crypto-thésarDs et non crypto-thésarDEs ; Joana, que j'ai rencontré en stage et dont j'ai été le témoin de mariage il y a moins d'un mois (merci pour cette belle histoire d'amitié).

Puis viennent les amis qui m'ont soutenu durant ces quatre années de thèse et qui comprenaient mes sautes d'humeur dus à "un bug", "une deadline d'article", etc. Je les remercie de me faire toujours penser à autre chose lors des moments délicats de ma thèse. Merci à Leïla pour les weekends shopping et à Feriel pour les mails de soutien lorsque je rédigeais et qu'il faisait 25 degrés à Paris !!

Merci à ceux que j'ai rencontré aux Etats-Unis lors de mon stage. Merci d'avoir fait de mon expérience américaine, une belle expérience.

Le meilleur pour la fin. Ma famille. Je tiens à remercier mes parents pour leur soutien dans mes choix, et leur encouragement durant ces quatre années de thèse. Un merci tout particulier à ma soeur Samia, d'avoir supporté une colloc' en rédaction d'article, en rédaction de thèse, une thésarde en somme. Amina, ma plus jeune soeur qui a trouvé ca drôle de soutenir un jour avant moi :) et enfin mon frère, El Hachemi qui a toujours été là pour des conseils de grand frère. Merci à vous !

À mes parents.

Résumé :

La complexité grandissante des architectures modernes, rend de plus en plus difficile la tâche des programmeurs à comprendre le comportement des programmes s'exécutant sur ces machines. De plus, les compilateurs actuels génèrent des codes difficiles à comprendre, dû à l'application d'optimisations plus agressives. Cette complexité croissante, tant au niveau des architectures qu'au niveau des compilateurs, renforce le besoin d'une analyse de performance pour aider le programmeur. Différents outils et techniques existent mais aucun outil n'est suffisant, seul, pour résoudre tous les problèmes.

Cette thèse propose deux outils, différents et complémentaires, pour l'évaluation de performances, de code binaire. Le premier outil, l'analyse statique de MAQAO, effectue une évaluation statique des performances du code, et donne une estimation pour la qualité du code, par exemple, les ratios de vectorisation. Le second outil, DECAN, est une nouvelle approche d'analyse de performances qui cible les instructions d'accès mémoire. L'objectif de DECAN est de détecter le groupe d'instructions responsable des faibles performances. Les deux outils ont été combinés pour proposer une méthodologie semi-automatique pour l'évaluation de performances.

Mots clés : analyse de performance, ré-écriture binaire, accès mémoire, optimisation de code, analyse statique, vectorisation, modèle de performance

Abstract:

Current hardware tends to increase pressure on programmers to optimize the codes. The complexity of modern architectures makes it more difficult to understand the behavior of the programs running on them. Moreover, the compilers apply aggressive optimizations which make the compiled code more difficult to understand. This increasing complexity shows that there is still a need of performance analysis to help the programmers. Different tools and techniques exist, but no single tool is a panacea; instead, different tools have different strengths.

This thesis proposes two different and complementary tools for performance analysis on binary code. The first tool, MAQAO's static analysis, performs a static evaluation of the performance of the code, and gives an estimate of the quality of the code, such as the vectorization ratios. The second tool, DECAN, is a new approach of performance analysis that targets the memory instructions to pinpoint the set of instructions responsible of the poor performance. Both tools are combined to propose a semi-automated methodology for performance evaluation.

Keywords: performance analysis, binary patching, memory access, code optimization, static analysis, vectorization, performance model

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The complexity of modern architectures makes it more and more difficult to understand the behavior of the programs running on them, and to propose a good optimization. Furthermore, the compilers applies more and more aggressive optimizations to gain performance, such as loop unrolling, inlining, loop splitting, loop fusion, and if-conversion. As a result, the compiled code is difficult to understand and even experts have difficulty to predict which optimizations will give the best performance. The best solution is a *perfect* compiler that will be able to apply the best optimization on a code to have the best performance on a target architecture. However, this *perfect* compiler does not exist, and even if it does we will always need to profile the resulting code to verify the optimization.

With a non-perfect compilers and an increasing complexity of modern processors, there is a need of some code's profilers that help the programmers to detect the problem responsible of the poor performance. With the advent of multicore, all efforts have been done to profile and optimize applications for multicore platforms: memory contention, false sharing, etc. Of course, the memory contention and accessing the memory hierarchy in general are multicore problems that should be kept in mind. However, we believe that it is only by achieving a reasonable optimization at the unicore level that a real speedup can be obtained in multicore. This is why, there is still a need for performance analysis in unicore to improve the multicore.

The code optimization is a process in which several methods of optimization are applied to improve performance code (more speedup, few use of resources). Before tackling the suitable optimization, one have to consider the code he is going to optimize. It is necessary to have an accurate idea of the target code: which data structures are used and how. This is called, code characterization. The second step in the code optimization process consists in diagnosing the problem of the program by identifying the cause of the poor performance. This step is performed with the help of performance tools, to automatize the process.

Different tools exist to analyze the performance of the programs, some of them target MPI codes, others based on statistical sampling targets both sequential and parallel codes. Since the concept of performance is large, there is no specific tool that solves all the performance problems. Any user who wants to optimize his code will use a set of tools, since different tools have different strengths.

Finally, the third and last step of the process of optimization consists in prescribing a solution to solve the problem of poor performance. This solution corresponds to a good optimization applied to achieve the performance objectives.

1.1 Contributions

In this thesis, we present two different approaches for performance analysis in unicore. The first approach is a static analysis tool based on a performance model of x86 architecture. It tackles the bottlenecks that can appear in the front-end and the back-end of the pipeline. It computes several metrics to quantify the quality of the code, such as the degree of vectorization of the code, the number of cycles spent in the execution ports, and an estimation of performance in the different levels of cache and RAM.

The second contribution of the thesis is a new technique for performance analysis called DECAN, for decremental analysis. It performs on SSE memory access instructions on the binary code. It patches the binary code by transforming the memory access instructions with `nop`s to evaluate the impact on performance of these accesses. This technique allows to deal with load and store instructions and to have a more precise information about which instruction is responsible of the poor performance.

Combining these two tools, we proposed a semi-automated methodology for a better evaluation process. This methodology combines the use of MAQAO's static, analysis, DECAN, and significant hardware performance counters. We applied this methodology on two real-life HPC applications from RECOM Services and Dassault-Aviation. We achieved a speedup of 1.4 on the `RBgauss` routine of the RECOM application, and a speedup of 2.5 on the `EUFLUXm` routine of the Dassault application.

1.2 Organization

This dissertation is organized as follows:

- Chapter 2 presents an overview of the different fields addressed by our tools. It makes an overview of the evolution of the microarchitecture, the memory hierarchy and presents the existing performance analysis tools.
- Chapter 3 presents MAQAO's static analysis, the first contribution in this thesis.
- Chapter 4 describes the second contribution, DECAN, the decremental performance analysis tool.
- Chapter 5 proposes a semi-automated methodology of performance evaluation, which combines our two tools.
- Finally, a conclusion and perspectives are presented.

Chapter 2

Computer Evolution and Performance Analysis

2.1 Context

This thesis presents two techniques/tools to evaluate the performance of programs. The first technique is a static analysis tool, Maqao's static analysis which is a fast technique to have first information about the quality of the code. It is based on a x86 performance model, so it helps to understand better, the behavior of the code on this microarchitecture. The second tool is Decan, a new approach to profile and understand memory operations. It targets load and store operations to detect the impact on performance of these memory accesses.

The process of optimization is not a *one step process*. Applying an optimization to a program requires diagnosing performance problems. To define a good optimization process for a software running on a target architecture, one should have a good understanding of: the behavior of the software, the microarchitecture the program is running on, and the interactions between the two. With the two tools proposed in this thesis, we target two important parts of the modern architectures: the processor (ie. microarchitecture) and the memory (since the gap between processor speeds and memory accesses is growing every year).

Microarchitectures are becoming more and more complex. Some mechanisms in modern microprocessors are not easy to understand for a programmer, so he can tune his program to best utilize scarce resources (e.g. bandwidth, functional units, cache). For example, for *vectorization* mechanism, it is not clear when the compiler converts scalar programs into vector programs. Depending on the programming language used, it helps the compiler to know when to use vector instructions. The *vectorization* mechanism consists in using SIMD instructions such as MMX/SSE/3DNow! instructions for x86 architectures (Intel [58, 59, 61, 60], and AMD [20, 17, 18, 22, 21, 19]), and Altivec [57, 56] for Power Architecture.

The complexity of the microarchitectures makes the process of bottleneck detection a tedious problem. They tend to place increasing pressure on programmers and tools to optimize scientific codes. Numerous tools and techniques exist, but no single tool is a panacea; instead different tools have different strengths. Most of these tool perform dynamic profiling on routine level with significant overhead to have accurate details about the profiled region.

Before presenting the tools, we propose in this chapter to describe the different fields they address. We presents an overview of the evolution of the complexity of the microarchitecture (field addressed in Maqao static analysis), the memory hierarchy and and how memory latencies influence performance (since memory accesses are addresses in Decan), and an overview of the existing performance analysis tool with a comparison with our tools.

2.2 Memory Wall

At the beginning of the computer manufacturing, the different components such as CPU, memory, network interfaces, were developed together and were well balanced in their performance. Nowadays, this balance is no longer possible. The CPU microarchitecture was developed and a lot of efforts were done to optimize this component and to improve its performance. However the memory did not follow this progress, for cost reasons, and an important gap [32] appeared between the CPU and the memory buses performance as shown in Figure 2.1. Indeed, the microprocessors are now generally clocked at 3GHz while the memory buses are clocked at 1,33GHz.

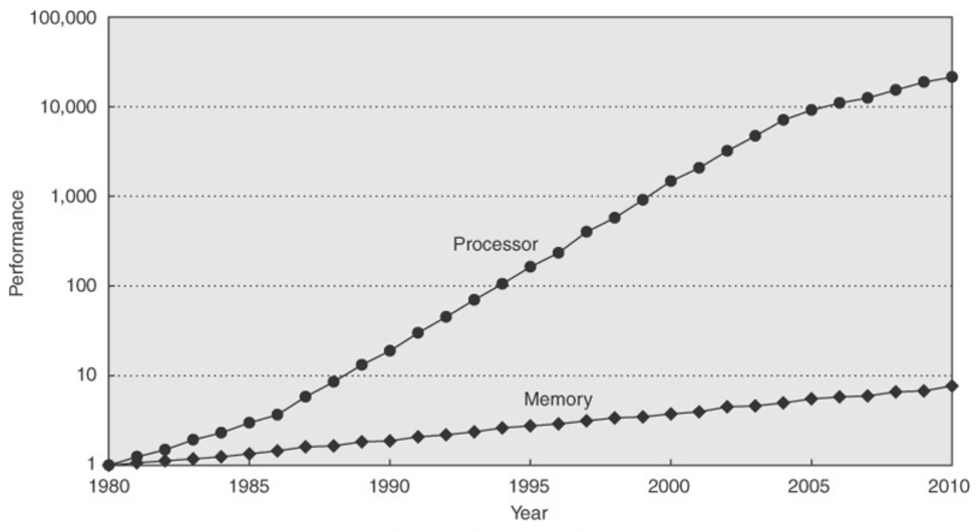


Figure 2.1: Processor-memory performance gap: starting in the 1980 performance, the microprocessor and memory performance over the years.

To alleviate this, memory caches are added. Different approaches are proposed to fully exploit the caches [25]. In the rest of the section, we describe the memory caches, and cache prefetching.

2.2.1 Caches: implementation and behavior

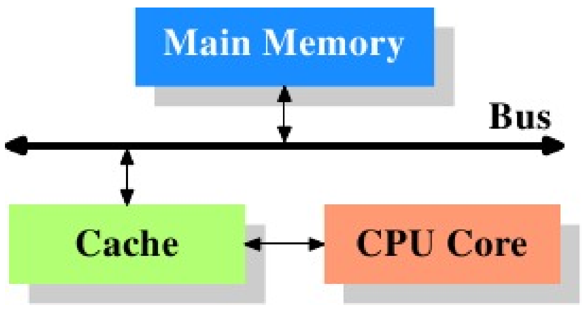


Figure 2.2: Minimum cache configuration [42].

Caches play the role of buffers between the processor and the memory (ie. RAM). The concept of caches consists in having a copy of RAM data in the caches, so the data can be reused without accessing the RAM once again. This reuse consists in code or data, spatial and temporal locality. This means that in a small period, the code or the data has a great chance to be reused. For example, in a loop, the same code is executed over and over (*spatial locality*). Or, the same data has a great chance to be reused in a short period (*temporal locality*).

The size of the cache is always smaller than RAM. If the working set of a program is small enough to fit in the cache, so the program's execution is efficient. However, in current programs that run multiple processes, each process has its own working set. In that case, the cache does not store the working set of all the processes. To deal with the limited cache size, some strategies are needed to know what should be cached.

Since not all the working set is needed by a program at exactly the same time, one solution, is to replace the data in the cache. This can be done, when the program is executing so there will be no delay for the replacement, and the cache appear bigger than it actually is. This replacement is a kind of prefetching, a strategy that will be described further.

Figure 2.2, shows a cache configuration like it was in the early cache system. The cache is connected directly to the CPU core. The cache communicates with memory through a bus called `FSB` for front-side bus. Soon after the introduction of the caches, microarchitecture became more complex. The gap of speed between the cache and the memory increased and new levels of caches are added. These added caches are bigger in size and slower than the first level of cache. Figure 2.3 is a schematic of what the current system look like.

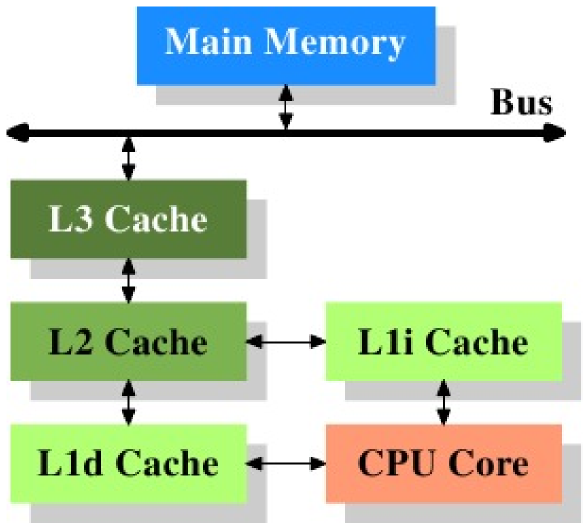


Figure 2.3: 3-level caches configuration [42]. The L1i is the instruction cache while the L1d is the data cache.

To give an idea of the cost of an access to the cache, we take as an example the Intel Core 2 and the Intel Nehalem processors. In the Intel Core 2 the cost of a `SIMD` load to the first level of cache is 1 clock cycle while it costs up to 1.22 clock cycle for a `SIMD` load in the second level of cache. These times do not correspond to cache latency. They correspond to the troughput, which means in that case, the number of `SIMD` loads that can be executed per cycle (1 per cycle in Core 2, L1 cache level). On the Nehalem, the throughput is the same for the 1st (1 clock cycle) and the 3rd level (1.22 clock cycle) of caches. The 1st level of cache in both Core 2 and Nehalem has the same size.

Different techniques are used to make the data stay in the cache as long as possible. The most famous technique is *the data blocking* or *tiling*. This technique is applied in many cases [83] for linear algebra kernels. Some attempts have been made to automatically tile loop nests [27, 88]. Most often, they target regular loops using the polyhedral model [46, 47].

In multicore processors, all the cores have a copy of almost all the hardware resources. In each core, the threads share the resources of the processor. Figure 2.4, shows what a multicore processor looks like. There are two cores, containing two threads each. The threads share the same first level cache, while each core has its own first level cache. The cores share the high level caches and the processors (core + high level caches), if there is more than one, do not share any caches.

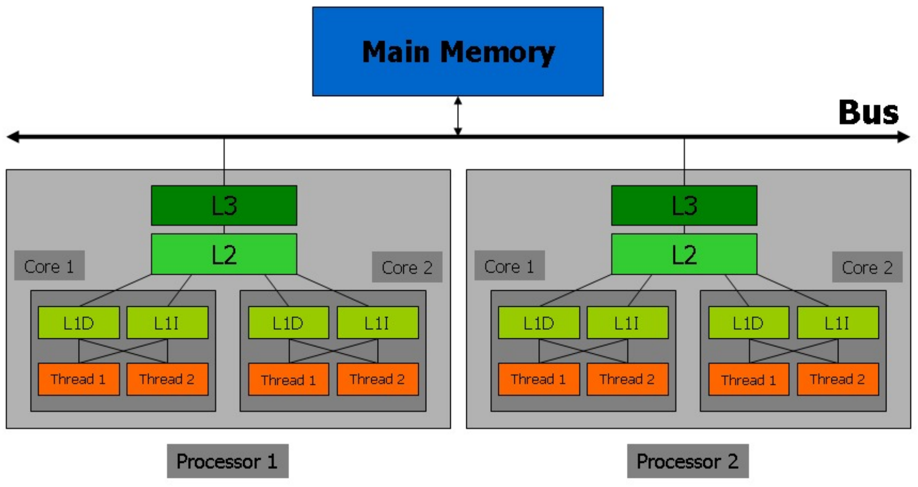


Figure 2.4: Multicore configuration [42]. There are two processors with two cores each sharing the main memory. Each core has 2 threads that are sharing the first level cache (instruction and data). Each core has its own first level cache, and the cores of the processor share the high level caches. The processors do not share any cache.

2.2.1.1 Miss factors

A cache miss is a failed attempt to read or write data from/to the cache. This means that the data is not fetched in the cache and should be brought from main memory: this results in an access with a high latency. We list three kinds of misses:

- Instruction cache read miss: it is a miss which is due to the absence of the instruction in the instruction cache. This miss results in the most delay. The instruction needs to be fetched from the main memory.
- Data cache read miss: it is a miss that causes less delay because instructions that are independent from the read miss can be executed while the data is brought from main memory.
- Data cache write miss: in this case the write is queued and the process can

continue until the queue is full. This miss is the one that causes the least delay.

To decrease the cache miss rate, many analyzes have been done to find the good combination between, cache size, associativity, block size, and so on. The cache misses are separated into the three known Cs misses: compulsory, capacity, and conflict misses.

- Compulsory miss: they are misses caused by the first reference to data.
- Capacity miss: they are misses due to the finite size of the cache.
- Conflict miss: they are misses that can be avoided. They are due to associativity and replacement policy.

Figure 2.5, shows the cache miss rate versus the cache size for different cache implementations. The compulsory misses give the insight that there is no need to increase the size of the cache beyond 1MB to increase the performance of the machine on SPECINTCPU2000. The fully associative cache miss rate is almost representative of the capacity miss rate, the difference is in the replacement policy which is a *perfect replacement policy* for the capacity miss rate and *LRU* for the simulates fully associative cache. We see that the capacity miss rate falls between 32K and 64K, this indicates that the benchmark has a working set of roughly 64K. Finally, the difference between the direct-mapped and fully associative caches corresponds to the conflict miss rate. This shows that the secondary misses benefit from the high associativity.

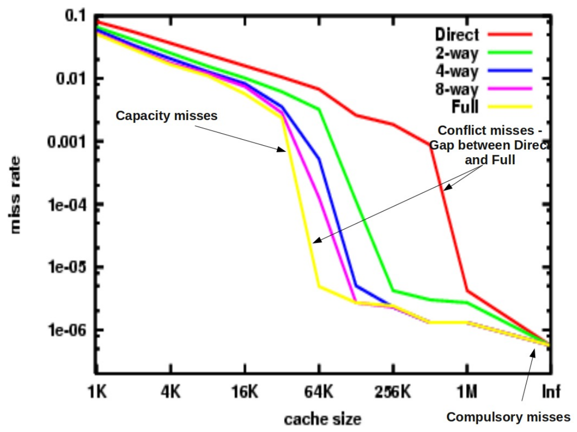


Figure 2.5: Cache miss rate versus cache size in the SPECINTCPU2000 [31, 105].

2.2.1.2 Implementation

The main problem for cache implementors is to find a cache line to store each cell of main memory. This is not realistic, as we know the working set of actual programs is larger than the cache size. Moreover, the size of the main memory is up to 1000x larger than the cache size, for example 4MB for a cache against 4GB for the main memory.

Associativity Different implementations to fully exploit the cache have been proposed. Before detailing these implementation, we give a brief description of the structure of a cache and main memory entries.

Figure 2.6 shows the structure of a cache line entry. The *data block* contains the data fetched from the main memory to the cache, and the *valid bit* is a bit that is set when the data in the cache entry is valid. The *tag* contains the value associated to each cache line (to distinguish all the memory cells that are stored in the same cache set).

TAG
DATA BLOCK
VALID BIT

Figure 2.6: The cache line entry structure with the tag, the data block and the valid bit.

Some cache implementations cache store any memory location in any cache line. This is called a *fully associative cache.* The tag of each cache line is compared to the tag of the memory address requested. If the tags match, then the memory location is stored. This method of "caching" is very efficient for small caches where the number of entries is not high. Indeed, in current L2 caches of 4MB, where the size of the cache line is 64B, there will be 65,536 entries for the cache. Thus the tag of the requested memory address must be compared to 65,536 entries in few cycles. However, the *fully associative* caches are mainly implemented for very small caches such as the *Translation Lookaside Buffer (TLB)*, where there a few dozen entries.

For bigger size caches, we must restrict the search. The most restrictive way is to dedicate to each memory location one cache line. This method is called *direct mapping.* It is easy to implement. To find a slot in the cache, a very simple hash function is used. The drawback of this method, is the risk to always choose the same slot, so the same cache line is often evicted while other cache lines are never used. The *direct mapping* can be compared to a parking lot problem, with many students having permits for the same spot when the other spots have very few permits. In this case the *direct mapping* works poorly.

A trade off between the *fully associative* and the *direct mapping* can be found. It is called a *set associative* cache. It combines the advantages of the two previous implementations. The tag and data storage are divided into sets. Each tag/data storage is selected by the address of a cache line. In each set, a *direct mapping* is applied, but instead of having one cache line for each set, there are several cache lines values for the same set. When a request for a memory location is done, then the tag of all the sets is compared to the requested memory address, which is close to the behavior of a *fully associative* cache.

Actually, most of the caches used are *set associative.* In the Sandy Bridge, unlike in the Nehalem, the L1 instruction cache is 8-way associative (4-way in the Nehalem). Figure 2.7 shows the effects of the associativity and the cache size on L2 cache misses.

Figure 2.8, illustrates the 3 implementations of cache described previously.

Multi-processors support In the previous section, we presented cache implementations in the case of one processor. When it comes to multi-processor architectures a problem in the caches that are not shared appears.

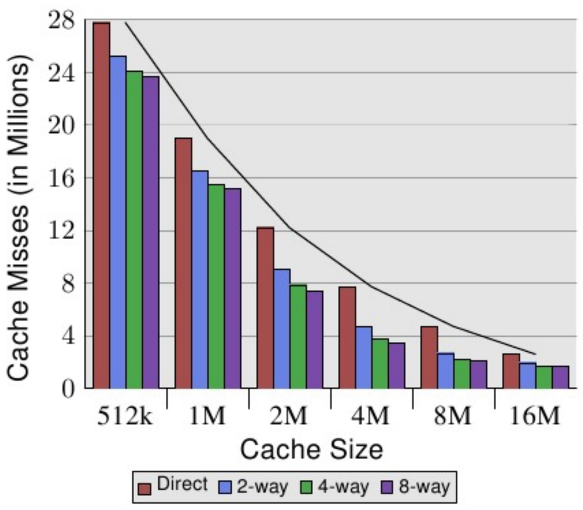


Figure 2.7: The impact of the cache size and the associativity on L2 cache misses, on gcc benchmark (cache line = 32 bytes) [42].

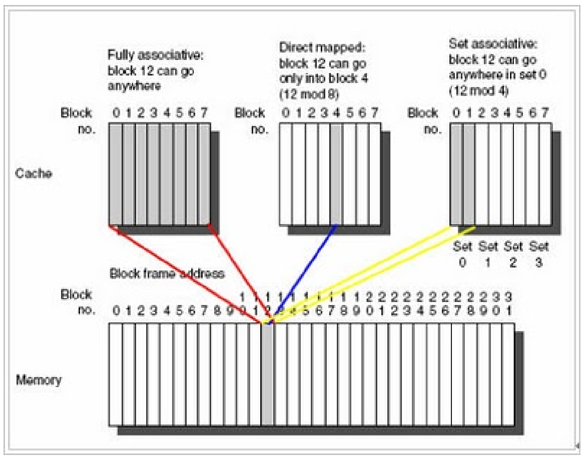

Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
Cache
Set 0
Set 1
Set 2
Set 3
Block frame address
Memory

Figure 2.8: Fully associative, direct mapping and set associative cache implementations.

It is costly to provide a direct access to the cache to an extern processor. The question is when to transfer a needed cache line and how? If the cache line has been modified by a processor, how will the other processor that needs the cache line know that the data has changed? This problem is known as *cache coherency* and the *MESI protocol* has been developed to palliate the problem.

The *MESI* protocol shown in Figure 2.9 is a *cache coherency* protocol based on 4 states:

1. Modified: the cache line is only present in the current cache and is modified by the local processor.

2. Exclusive: the cache line is only present in the current cache and not modified (but known to be modified).

3. Shared: the cache line is not only present in the current cache. It may be present in another cache's processor and it not modified.

4. Invalid: the cache line is invalid.

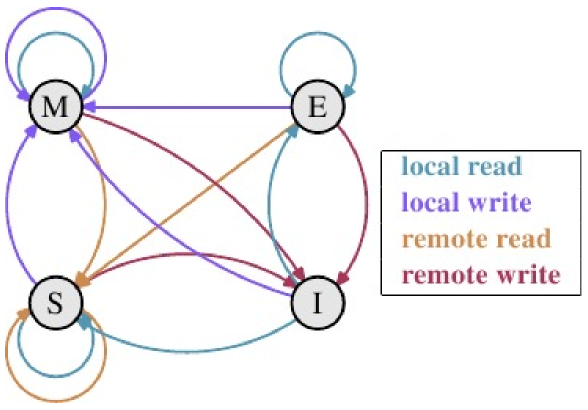


Figure 2.9: MESI protocol transitions [42].

At the beginning, all cache lines are empty and their state is Invalid. When data is requested by a processor for writing then the cache changes to Modified. If data is requested for reading then the cache changes to Shared or Exclusive depending if the cache line is present in another cache's processor or not.

If the cache line is Modified, and a reading or writing is requested from the local processor, then the state does not change. If a remote reading is requested from another processor, then the local processor send the content of the cache line to the applicant and change the state to Shared. If a remote writing is requested then the local processor sends the content of the cache line to the applicant and the state is changed to Invalid. When the state changes to Shared the content of cache line is also written in the memory. When a remote write is requested, then the write operation should be announced to the other processors by an *RFO (Request For Ownership)*. This operation invalidate all the processors' cache lines that holds the written memory line.

If the cache line is Shared, and a local request for reading occurs then nothing happens. If a local request for write occurs then the state is changed to Modified and all other copies in other caches are set to Invalid and a *RFO* is announced. When a remote reading is requested, nothing happens. If a remote writing is requested then the state is changed to Invalid and a *RFO* is announced to invalidate all other copies.

When the cache line is Exclusive, the behavior is almost the same then is a Shared case, with one difference: when a remote write is requested there is no need to announce a *RFO*, since the local cache is the only cache that stores the target data.

What we can conclude from the *MESI protocol* is that the write announcement with the *RFO* messages is costly because is requires to pass through the bus. In case of multicore, the *RFO* may occur if, for example, a thread migrates from a processor to another (all cache line should be moved from a processor to another), or the same cache line is needed by more than one processor.

2.2.1.3 conclusion

In the current modern processors, most of them have a multi-level cache, from the first to the third level of cache. Most of them target the problem of the cost of cache/RAM and the execution time. In current research, interest has been focused on energy efficiency, fault tolerance and other purposes. Wang *et al.* in [102] present a new cache architecture that can simultaneously have low miss rates, short access times, and is power efficient. They used the CACTI cache simulator [96] to validate their design.

2.2.2 Cache prefetching

Prefetching is used to hide the latency due to memory accesses. Prefetching can be triggered by special hardware events and is called *Hardware Prefetching* or *Software Prefetching* if it is requested explicitly by the programmer.

2.2.2.1 Hardware prefetching

The trigger to the CPU that enables the hardware prefetcher is in general a sequence of cache misses in a certain pattern. In old implementations of the hardware prefetcher, only cache miss pattern were recognized and handled. In the modern processor, not only cache misses trigger a hardware prefetch but also stride patterns that are then handled appropriately.

The hardware prefetcher is not triggered for every cache misses. This can be very bad for performance because of the limited bandwidth. This is why the hardware prefetch is triggered only if at least two sequences of cache misses are needed. A too aggressive prefetching policy can lead to a waste of memory bandwidth, cache thrashing, and so on.

Modern processors assign a stream to each cache miss. When a determined number of streams is obtained then the hardware prefetcher starts.

2.2.2.2 Software prefetching

The advantage of hardware prefetching is that the program does not need to be modified. However, the drawback of the hardware prefetch is that the access patterns must be relatively simple to be recognized. In the case of complex patterns, one solution is to use the software prefetcher.

One method for software prefetching is inserting pragmas/special instructions into the target code. In x86 and x86-64 microarchitectures, Intel's convention for compiler intrinsics to insert these special instructions can be used. For example:

```
#include <xmmintrin.h>
enum _mm_hint{
_MM_HINT_T0 = 3,
_MM_HINT_T1 = 2,
_MM_HINT_T2 = 1,
_MM_HINT_NTA = 0
};

void _mm_prefetch(void *p,enum _mm_hint h);
```

The different hints request that the data be fetched to all cache levels or only to the L2 cache, for example. The details of the hints can be found in the Intel 64 Instruction Manual [60].

The most practical software prefetching techniques proposed are techniques used inside the compiler [92, 79], so the user-programmer has no need to modify his program.

2.2.3 Conclusion

In this section we presented part of the hardware that is generally considered as performance bottleneck: the memory. We described the implementations of the caches and the behavior of the cache in the case of misses. With multicore processors the problem is that, when memory is shared, only one processor can access this memory, making the other processors stall.

To palliate this problem, the `Non-Uniform Memory Access` is designed, providing a separate memory zone for each processor. The problem with `NUMA` is to keep the coherency of the data across the shared memory.

The problem of memory is tackled by several research fields. Because of the existing gap between the memory and the processor performance, the memory is often considered as a bottleneck. In this thesis, the memory criterion is taken into account in the decremental analysis tool Decan, presented in Chapter 4.

2.3 Microarchitecture Complexity

In this section, the purpose is not to make a historical of the microprocessors' architecture. This work has already been done in depth by Hennesy and Patterson in [54] or Baer [23]. The purpose is to describe a subset of the hardware that contributes in increasing the performance of a program if its usage is maximized.

The details of the microarchitecture described in this section have been implemented in Maqao for the static analysis. Indeed, Maqao's static analysis, which is one of the contributions of this thesis in Chapter 3, implements a detailed performance model for the Core 2 architecture. This detailed performance model leads to accurate performance data.

2.3.1 Instruction pipeline

2.3.1.1 Description

The instruction pipeline is a powerful technique to improve the processor performance. The scalar microprocessors are considered as the simplest form as they perform one operation at a time. Because the amount of computation is very high, there was a need to speed up the execution of a program. The instruction pipeline is considered as a solution to overcome this problem. The technique consists in splitting the process of an instruction into small steps, thus some steps (of different instructions) can be processed at the same time. Figure 2.10 shows the instruction pipeline stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EXE)
4. Memory Access (MEM)
5. Register Write-Back (WB)

When an instruction is processed, it crosses the different stages of the pipeline. When an instruction `I1` is in the decoder stage, an instruction `I2` can be fetched.

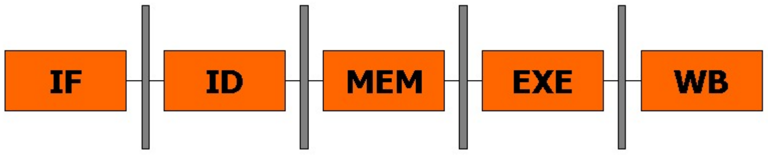


Figure 2.10: A view of a five-stages pipeline.

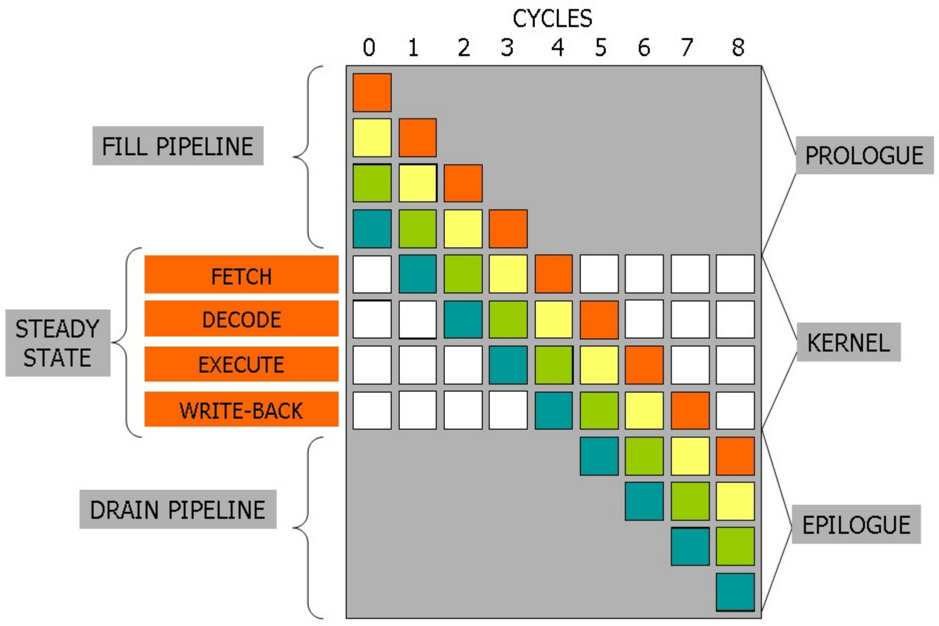


Figure 2.11: Instruction pipelining in a 4-stages pipeline. When full, there is a throughput of one instruction per cycle.

After decoding, `I1` is executed and `I2` is decoded, then a third instruction `I3` can be fetched at the same time, etc. When each stage of the pipeline is processing an instruction, the throughput is one instruction processed per cycle. Figure 2.11 illustrates the instruction pipelining.

However, having instructions dependencies prevents from the maximal throughput of one instruction per cycle. Indeed, if the instruction `I2` needs the result of the instruction `I1`, then `I2` should wait until the result of `I1` is written back. This pending is called *a pipeline stall*.

In reality, the pipeline is more complex than that. For example, more stages can be added so the functional units (integer and floating-point) can be pipelined. Also, depending if the program is memory- or compute-bound more stalls can occur. The rest of the section explains the different stages added in the modern pipelines, and how these new stages may cause more stalls.

In the next section, we present the evolution of the pipeline on different architectures from Intel and AMD and describe the difference in complexity from an architecture to another on each stage of the pipeline.

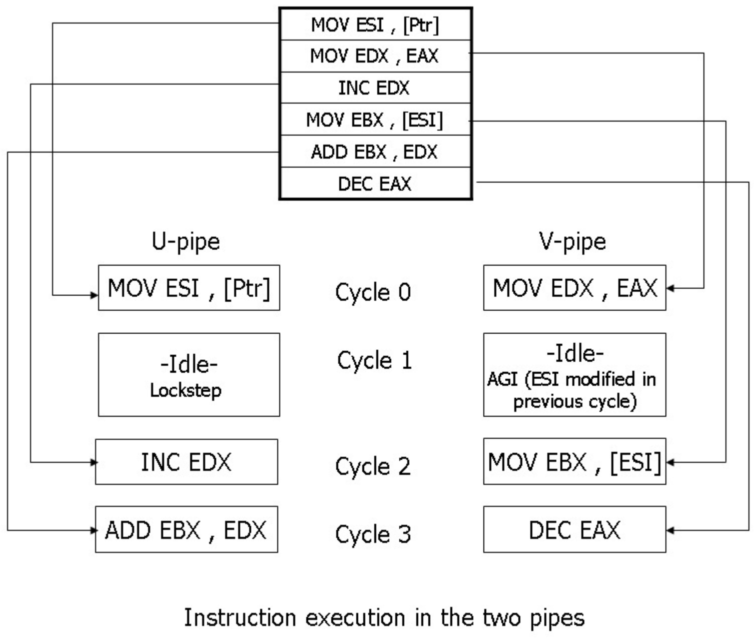


Figure 2.12: Instruction execution in the two pipes in a Pentium 1. An AGI occurred in the V-pipe because `ESI` register is modified in an instruction and used to address memory 3 instructions later. The U-pipe is idle because of the lockstep execution of the Pentium 1.

2.3.1.2 Instructions pairing

The Pentium 1 processor has two execution units: the U and the V pipes. The U-pipe executes any instruction while the V-pipe executes only simple instructions [13]. When two instructions execute in parallel in the two pipes, the two instructions are considered to have been paired. This execution in two pipes can almost divide the execution time by 2. If the instructions are well ordered in the program to be paired correctly, this can have a large benefit on the performance. By a good reordering of instructions, we mean to create pairs of instructions with one simple instruction that can be executed in the V-pipe and the other one in the U-pipe. This mechanism of instruction pairing is applied for the integer instructions.

It takes one cycle for the Pentium 1 to calculate an address. If any part of a register during a cycle is modified, this register can not be used to address memory during that cycle or the next cycle. This is called *AGI: Address Generation Interlock*. Instructions execute in *lockstep* on the Pentium 1, so if one pipe stalls for a cycle, making its instruction take one cycle longer, that extends by one cycle the time until the other pipe can begin its next instruction, as well. Figure 2.12 shows an example of an AGI/Lockstep stall.

This mechanism of pairing exists in both Pentium 1 and Pentium MMX where the instructions are executed in the order they appear in the program.

2.3.1.3 Instruction fetch

The instruction fetching is the step of the pipeline where the address of the next instruction to execute is retrieved from the `PC (Program Counter)`. In the Pentium

II, III, and Pentium Pro architectures, the instructions are fetched in an aligned 16-bytes block from the cache code into a double buffer that can hold two 16-bytes blocks. The double buffer allows the decoder to decode instructions that cross the 16-bytes boundary. The code is passed from the double buffer to the decoder in fetched-blocks of 16 bytes that start at an instruction boundary. If the instruction length is not available in time, then the fetched-block starts at a 16-bytes boundary. If the 16-byte fetched-block contains a jump and crosses a 16-bytes boundary, then the double buffer needs to keep two aligned 16-bytes blocks to generate the fetched-block. The same is true if in the fetched-block the instruction following the jump, crosses a 16-bytes boundary. In the worst case, the delay will be of 2 cycles [49].

The Pentium M has the same mechanism of fetching with one improvement: 16-byte boundaries do not cause delays in jumps. So, it is not important to align subroutine/loop entries. The Core 2 and Nehalem architectures have the same fetch mechanism than the Pentium M.

2.3.1.4 Instruction decode

The instruction decoder is the stage of the pipeline where the instruction is decomposed into micro-operations. For example, and instruction such as `add (%eax),%ecx`, is an instruction that does two operations, the first consists in loading the data from the memory (`(%eax)`), and the second operation is the addition. Each operation in called *micro-operation* or *micro-instruction*. Each micro-operation is executed in one execution unit. The decoder in the Pentium II, III and Pentium Pro architecture is decomposed in three sub-decoders: 4-1-1. The first decoder can handle any instruction of any length and with no more than 4 micro-operations per cycle. The other two decoders only handle instructions that generate no more than one uop each and are not longer than 8 bytes. The decoder receives from the fetcher (ie. instruction length decoder) a fetched-block of 16-bytes. If the fetched-block contains instructions that fit in the 4-1-1 pattern, then this fetched-block is decoded in 1 cycle. Otherwise, it takes more than one cycle to decode the block. We should know that the first instruction of a fetched-block always goes to the first decoder (4) [49]. The example below illustrate the dispatch in the 4-1-1 decoders:

```
mov (%esi),%edi -- 1uop
sub %8,%edi -- 1uop
add %eax,(%ecx) -- 2uops
sub %edi,(%ebx) -- 2uops
jle LABEL -- 1uop
```

The first instruction goes to the first decoder (4). The second instruction goes to the second decoder (1). The third and fourth instructions will wait until the first decoder (4) is free. The last instruction goes to the third decoder (1). It takes 3 clock cycles to decode this group of instructions because the 4-1-1 pattern is broken twice. To optimize the utilization of the pattern 4-1-1, the instructions should be reorganized to fit the pattern.

```
add %eax,(%ecx) -- 2uops
mov (%esi),%edi -- 1uop
sub %8,%edi -- 1uop
sub %edi,(%ebx) -- 2uops
jle LABEL -- 1uop
```

With this new schedule, the group of instructions is decoded in 2 cycles instead of 3 cycles.

In the Pentium M, Core 2, and Nehalem architecture, the decoder has a similar behavior. However in the Core 2 and Nehalem there is one more decoder that decodes one micro-operation. The pattern in Core 2 and Nehalem is a 4-1-1-1 pattern instead of a 4-1-1 pattern. Moreover, the Pentium M, Core 2, and Nehalem use a mechanism called Micro-op fusion to increase the throughput of the decoder. In the Micro-op fusion it is possible to fuse two micro-operations in one micro-operation so it can go in the decoder that only handles instructions of one micro-operation [49]. The other advantage of the micro-fusion is that downstream resources such as: Loops Stream Detector, Reservation Stations, Reorder-Buffer, only needs one slot for the fused micro-ops.

2.3.1.5 Register Renaming

The register renaming is a technique used to increase the number of instructions that can be executed in parallel. This technique is used in the Tomasulo Algorithm [97] and allows to resolve the Write-after-Write and Write-after-Read hazards. In [86], Qian *et al.* proposed a two-phases register renaming scheme used in implementing an x86-compliant processor.

The register renaming in the Pentium II, III, and Pentium Pro architectures, is controlled by the `Register Alias Table (RAT)`. The `RAT` is a unit that maps the architectural registers to the physical registers. The maximum throughput of the `RAT` is 3 micro-operations per clock cycle. It means that the `RAT` can rename up to 3 registers per clock cycle. In Core 2, Nehalem, and Sandy Bridge the register renaming is also controlled by the `RAT` but its throughput is increased to 4 micro-operations per clock cycle [49].

Data Hazards When different instructions uses a location for an operand, by reading it or writing it, executing these instructions in a different order may lead to some hazards such as:

- **Read-after-Write**: when a register is read it returns the value of the last write in the program. These instructions must be executed in order.

- **Write-after-Write**: when a register is written by successive instructions, the last value of the register must the one written during the last write.

- **Write-after-Read**: a read of a register must return the last value written in the register before the read.

2.3.1.6 Reorder-Buffer-read

The Reorder-Buffer-read stage is the stage of the pipeline where the values of the renamed registers are stored. In the Pentium II, III, and Pentium Pro, the RoB-read has 2 input registers and 2 output registers. A register can be read from the RoB-read without stall if it has been modified (ie. written) within 3 clock cycles. If the register is written in more than 3 clock cycles, then the register is read from the Permanent Register File and a stall may occur because the register file has only 2 read ports [49]. The Permanent Register File is an array of processor registers that can be read through dedicated ports and written through other dedicated ports [45, 14].

In Pentium M, Core 2, and Nehalem, the same problem if the number of registers to read happens. However, the number of read ports in the register file to read instructions' operands has increased from 2 to 3 ports.

In Sandy Bridge it seem like the number of read ports in the register file has been increased but it is no longer a bottleneck.

2.3.1.7 Retirement

The retirement station is the stage of the pipeline where the temporary registers used for register renaming are written in the Permanent Register File. At this stage of the pipeline, the instructions must retire in order. It means that if an instruction `I1` is not ready to be retired, none of the instructions following this `I1` can be retired before it. In Pentium II, III, and Pentium Pro the retirement can handle 3 micro-operations per clock cycle. Also a taken jump can only be retired in the first of the three slots of the retirement station. This is significant in a small loop where the number of micro-operations in the loop is not divisible by three. That means that if the last instruction in a taken jump, then the two other slots of the retirement station will be idle [49].

In Core 2, Nehalem, and Sandy Bridge architectures , the retirement station is not a bottleneck anymore.

2.3.2 Out-of-order execution

In the last 6 generations of microprocessors, beginning with the Pentium Pro, the mechanism of Out-of-Order is designed. The concept of Out-of-Order is that when an instruction is not ready to be executed because its operands are not ready yet, then the microprocessor handles later instructions which are ready and which don't have any dependency with the delayed instruction. Instead, in the in-order execution, if the operands of an instruction are not ready yet, then the processor stalls until they are available. The Intel *Atom* processor is in-order.

Typically, two instructions such as an addition and a multiplication, can be executed at the same time if both are independent of the other. Because the addition and the multiplications units are separated, these two operations can be executed in parallel. However, because there is only one multiplication unit, one cannot perform two multiplications at the same time even if they are independent. Figure 2.13 shows an example of two instructions that can be executed in parallel.

In all the microprocessors where the Out-of-Order is designed, the instructions are decomposed into micro-instructions (or micro-operations). For example, in the following instruction that does an integer addition: `add (%rbx),rax`, there is one micro-operation to read the data from the memory at the address `%rbx` and another one to do the addition with the register `%rax`. To increase the throughput, the mechanism of register renaming, explained Section 2.3.1.5, is used. It allows to break some false dependencies to increase the number of instructions that can be executed in parallel.

Everything in these microprocessors is fully pipelined, so every cycle a new instruction can be launched, if no stalls occur and no dependency exists.

In the Pentium II, III, and Pentium Pro architectures, the RoB can hold up to 40 micro-operations and 40 temporary registers. The reservation station holds up

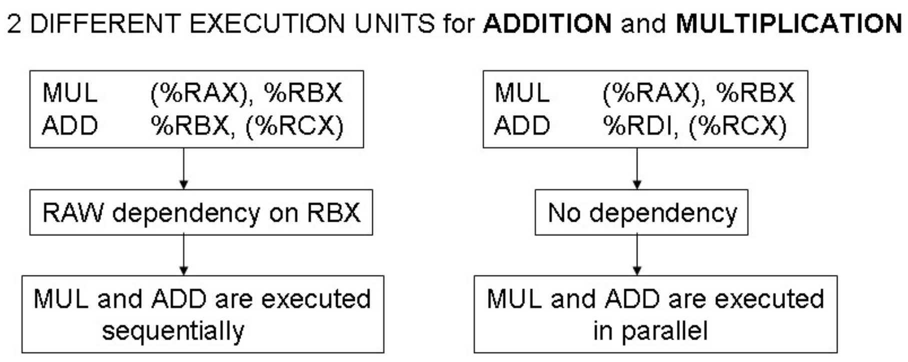

2 DIFFERENT EXECUTION UNITS for ADDITION and MULTIPLICATION
MUL (%RAX), %RBX
ADD %RBX, (%RCX)
RAW dependency on RBX
MUL and ADD are executed sequentially
MUL (%RAX), %RBX
ADD %RDI, (%RCX)
No dependency
MUL and ADD are executed in parallel

Figure 2.13: A couple of instructions that can be executed in parallel if they are independent.

to 20 micro-operations. After the Rob, the micro-operation waits in the reservation station and waits until its operand are ready. When the input data is ready, the instruction is executed. This make the process of Out-of-Order execution possible.

Some operations cannot be executed out-of-order. For example, if a read to an address is executed before a write to it, that is programmatically before the read, then an error is detected when the write computes the address and the read should be re-done.

Figure 2.14 show in the Pentium II, III, and Pentium Pro architectures that there are 5 execution units clustered around 5 ports:

- `P0-P1`: dedicated to arithmetic and logic operations and simple mov instructions.
- `P1`: handles the jump instructions.
- `P2`: dedicated to memory read.
- `P3`: dedicated to address computation for memory write.
- `P4`: dedicated to all memory writes.

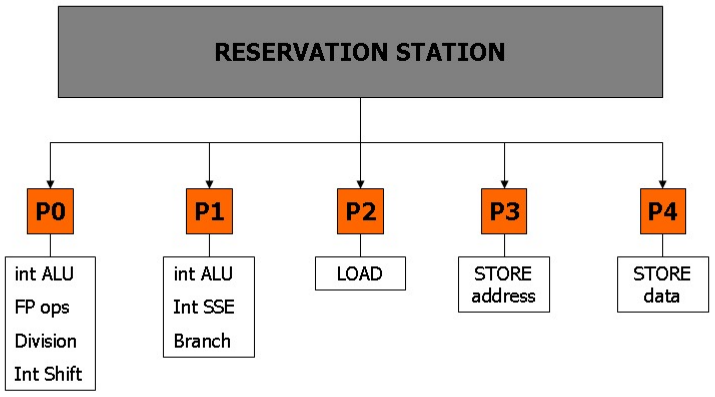


Figure 2.14: The execution units in the Pentium II, III, and Pentium Pro.

Each execution port receives one micro-operation per clock cycle, but the throughput cannot exceed 3 micro-operations per cycle because of the limit of the `RAT` and the retirement station. Some micro-operations take more than one cycle to execute, such as floating-point (fp) addition and multiplication, but the corresponding execution units are fully pipelined. That means that the execution unit can receive one fp instruction per cycle. Some instructions such as a division, are not pipelined at all. If there are 2 divisions successively in a code, the second one waits until the first is completely executed [49].

In the Pentium M, there are also 5 execution units. The throughput can achieve 5 micro-operations per cycle because two fused micro-operations can be considered as a single unfused one. So, 3 fused micro-operations can be considered as 5 unfused micro-operations.

In Core 2 and Nehalem, as shown in Figure 2.15 there are 6 execution units that can have a throughput of 6 micro-operations. All ports in Core 2 and Nehalem support the 128 bits vectors:

- `P0-P1-P5`: dedicated to arithmetic and logic operations and jumps.
- `P2`: dedicated for memory read.
- `P3-P4`: dedicated for address computation and memory write.

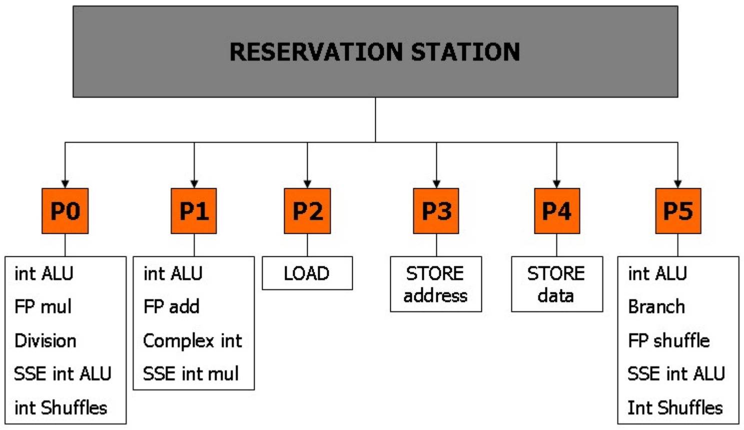


Figure 2.15: The execution units in the Core2 and Nehalem architectures.

In the Sandy Bridge, all ports support the 256 bits vectors. There are 6 execution units but the ports are clustered differently than in the Core 2 as shown in Figure 2.16:

- `P0-P1-P5`: dedicated to arithmetic and logic operations and jumps.
- `P2-P3`: both of them are dedicated for memory read and address computation .
- `P4`: dedicated for all memory writes.

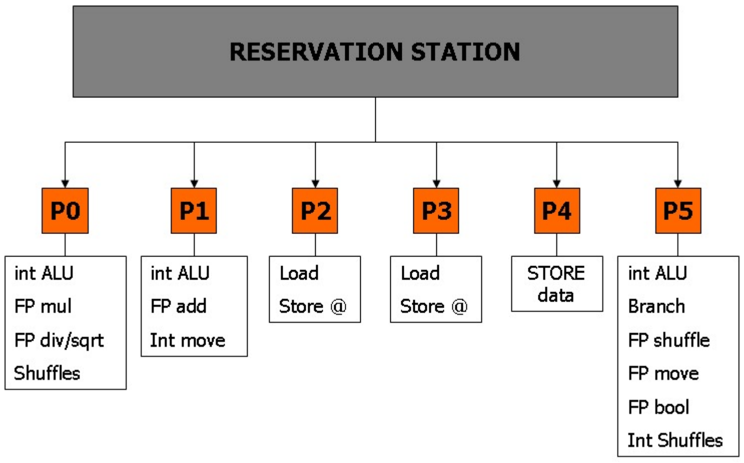


Figure 2.16: The execution units in Sandy Bridge architecture.

2.3.3 Branch prediction

In the latest generation of microprocessors, everything is pipelined. This allows the microprocessors, to handle several operations at the same time. When an instruction `I1` is executed, an instruction `I2` is fetched and decoded at the same time. The main problem with pipelining is branches. When there is a conditional jump in a code, it means that there are 2 possibilities. However, the conditional jump allows the instruction stream to go in just one direction. The microprocessor does not know which flow to feed into the pipeline. If there is one pipeline, and if the wrong flow is taken then the pipeline needs to be flushed and a big amount of time is wasted.

Different implementations of the branch prediction are known such as the saturating counters, or the two-level adaptive prediction [107, 43]. In [38] Alba *et al.* present a path-based loop predictor that allows to unroll loop iterations using a hardware mechanism. There is also the Neural Branch prediction, proposed the first time by Lucian N. Vintan [101] and improved in [39, 65]. Other researches are done on the Neural Network prediction [67, 16].

2.3.4 Loop Stream Detection

The point of this buffer is to, detect when the processor is executing a loop, stop predicting branches (potentially mis-predict the last branch), and stream the instructions out of the LSD.

In Core 2 micrarchitecture, the LSD is situated between the fetch and the decoder and is able to hold 18 instructions. In that case (a loop of 18 instructions), the fetch and branch prediction are disabled. In Nehalem, the LSD is located after the decoder and is able to handle up to 28 micro-operations. The Sandy Bridge eliminates the LSD in favor of a uop-cache.

2.3.5 Conclusion

In this section we presented a description of the last generation of microprocessors. From the Pentium Pro to the Sandy Bridge we see the evolution in complexity of the microarchitectures (ie. longer pipelines, complex mechanisms such as the out-of-order, etc). This increasing complexity, augments the percentage of bottlenecks, and it becomes more difficult to pinpoint the real performance problems of a code running in such microarchitecture.

With the increasing complexity of microarchitectures and memory, it becomes difficult to detect which part of a program causes poor performance. For that reason, several tools to analyze the performance of a code have been developed to help the programmer optimize his code. Some of them perform on source code, others on binary. Some of them have a big overhead, others negligible overhead but with less accuracy.

The next section describes briefly the impact of the compiler on the performance evaluation and why it is better to focus on the binary. After that, we make an overview of the existing performance analysis tools.

2.4 Compiler impact

The compiler transforms the source code written in a programming language to a binary code that can be executed on a target platform. This transformations involves several optimizations that the current version of compiler can apply. This transformation or *tuning* has the purpose to reduce the execution time of the binary. These *tuning* is generally applied on loops, but not only. We focus on loops, because in HPC codes, they are the most-time consuming region.

Analyze the performance of a code in source level is the easiest way to do, because it is simple to insert probes for instrumentation in a target region. However, when the instrumented code is compiled, we do not control the optimizations applied by the compiler and which implies a false interpretation of the results of the instrumentation.

To avoid the compiler-dependency, we focused in the tools presented in this thesis on the binary code. At binary level, we are compiler-independent, and the binary transformations applied by the tools to evaluate the performance are not polluted by the compiler.

2.5 Performance Analysis Tools

2.5.1 Introduction

In this section, we make a brief overview of existing performance analysis tools. We describe the characterization of each, what is the focus if the tool and the results obtained with the tool. We present type of different tools, some of them target loops, others target functions. Some of them perform static analysis and others dynamic profiling.

2.5.2 Static profiling

The static profiling is considered as the first step in the process of performance evaluation. It is a fast process that abstracts the dynamic phenomena. It does not take into account the semantic of the code, since it does not execute the target code and it does not consider the input dataset. One of the tools presented in this thesis is MAQAO's static analysis. The following section presents another static analysis tool proposed by Intel.

2.5.2.1 IACA

Introduction *IACA (Intel Architecture Code Analyzer)* [99] allows the analysis for Intel Advanced Vector eXtension and the previous sets of instructions. It allows to identify the link between the kernel instructions with the processor execution ports. It gives the number of cycles spent by each instruction in its corresponding execution port. It performs a static analysis of the latency cycle counts and detects the critical path.

Analysis *IACA* is a command-line tool with ASCII output. It handles a single code section that is marked for analysis within an executable, a shared library, or an object file. The tool is a static tool, it sees the code as a one block of instructions, and does not follow branches. To evaluate a loop using *IACA*, some *IACA* macros should be placed in the code, as follows:

```
while (condition)
{
      IACA_START
      <loop body>
}
IACA_END
```

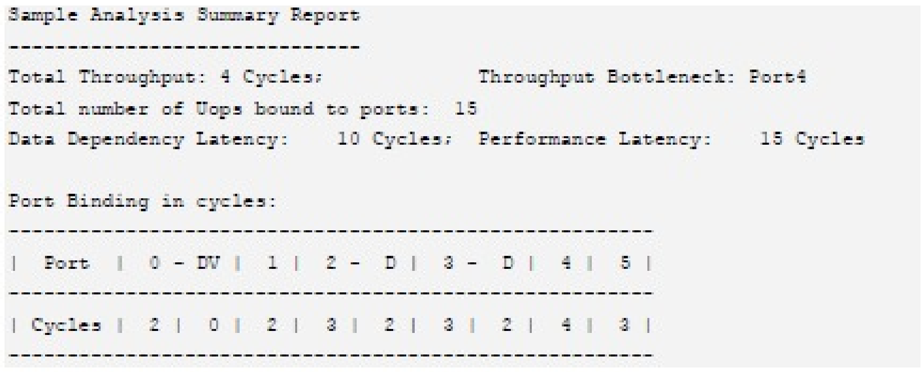
In that case, by placing the first macro at the beginning of the loop and the second one after the end of the loop, it skips the initialization of the loop and keeps the loop branch.

Figure 2.17 below shows the report summary of the *IACA* analysis. It contains:

- The total throughput counted in cycles.
- The bottleneck that limits the throughput.
- The total number of micro-operations.
- The data dependency latency (the time in cycles that takes to execute the data dependency critical path).
- The performance latency (the time in cycles that takes to execute the performance critical path).

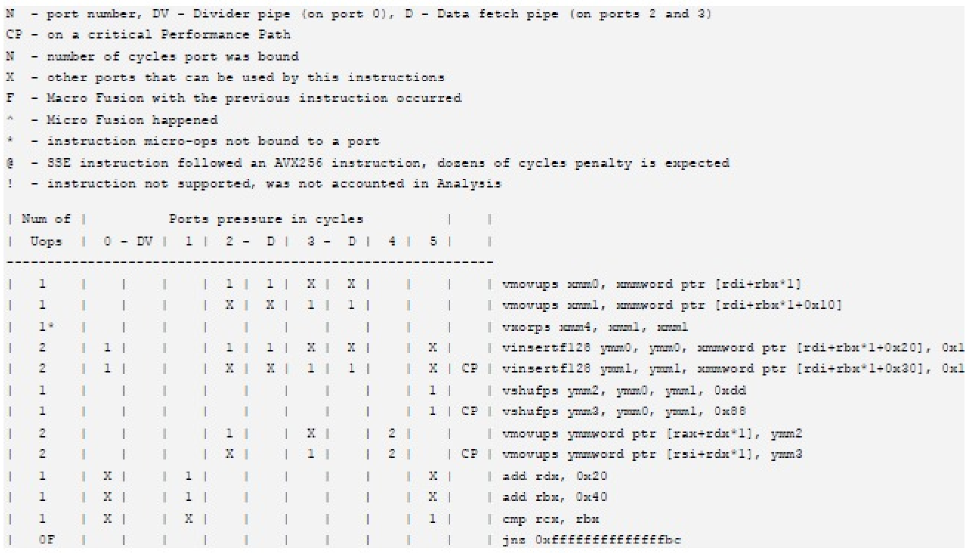
In the *IACA* analysis report, there is a detailed part (Figure 2.18) the gives and analysis information about each instruction:

Conclusion *IACA* is a performance evaluation tool that performs static analysis. It is based on modelisation of the performance of the Intel Advanced Vector Extensions. It also support the Nehalem. *IACA* performs a static analysis on a section of code and gives information related to the bottleneck in case of L1 data. Comparing to our MAQAO's static analysis, *IACA* does not perform on the loop level because it does not detect loops. Moreover, it does not give performance estimations in the memory/cache levels up to L1. More details about what MAQAO's static analysis are given in Chapter 3.



```
Sample Analysis Summary Report
------------------------------
Total Throughput: 4 Cycles;            Throughput Bottleneck: Port4
Total number of Uops bound to ports:  15
Data Dependency Latency:    10 Cycles;  Performance Latency:    15 Cycles

Port Binding in cycles:
-------------------------------------------------------
|  Port   |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |
-------------------------------------------------------
| Cycles |  2 |  0 |  2 |  3 |  2 |  3 |  2 |  4 |  3 |
-------------------------------------------------------
```

Figure 2.17: *IACA* report summary [99].



```
N  - port number, DV - Divider pipe (on port 0), D - Data fetch pipe (on ports 2 and 3)
CP - on a critical Performance Path
N  - number of cycles port was bound
X  - other ports that can be used by this instructions
F  - Macro Fusion with the previous instruction occurred
^  - Micro Fusion happened
*  - instruction micro-ops not bound to a port
@  - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
!  - instruction not supported, was not accounted in Analysis

| Num of |          Ports pressure in cycles          |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|   1    |    |    |    |  1 |  1 |  X |  X |    |    |    | vmovups xmm0, xmmword ptr [rdi+rbx*1]
|   1    |    |    |    |  X |  X |  1 |  1 |    |    |    | vmovups xmm1, xmmword ptr [rdi+rbx*1+0x10]
|   1*   |    |    |    |    |    |    |    |    |    |    | vxorps xmm4, xmm1, xmm1
|   2    |  1 |    |    |  1 |  1 |  X |  X |    |  X |    | vinsertf128 ymm0, ymm0, xmmword ptr [rdi+rbx*1+0x20], 0x1
|   2    |  1 |    |    |  X |  X |  1 |  1 |    |  X | CP | vinsertf128 ymm1, ymm1, xmmword ptr [rdi+rbx*1+0x30], 0x1
|   1    |    |    |    |    |    |    |    |    |  1 |    | vshufps ymm2, ymm0, ymm1, 0xdd
|   1    |    |    |    |    |    |    |    |    |  1 | CP | vshufps ymm3, ymm0, ymm1, 0x88
|   2    |    |    |    |  1 |    |  X |    |  2 |    |    | vmovups ymmword ptr [rax+rdx*1], ymm2
|   2    |    |    |    |  X |    |  1 |    |  2 |    | CP | vmovups ymmword ptr [rsi+rdx*1], ymm3
|   1    |  X |    |  1 |    |    |    |    |    |  X |    | add rdx, 0x20
|   1    |  X |    |  1 |    |    |    |    |    |  X |    | add rbx, 0x40
|   1    |  X |    |  X |    |    |    |    |    |  1 |    | cmp rcx, rbx
|   0F   |    |    |    |    |    |    |    |    |    |    | jnz 0xfffffffffffffbc
```

Figure 2.18: *IACA* detailed report [99].

2.5.3 Dynamic profiling

Dynamic analysis involves the testing and evaluation of a program based on execution. Some dynamic profiling tools perform on source code, others on binary code. Some of them are accurate with a big overhead, others have a negligible overhead but with less details on the profiled region.

In this section we give an overview of the existing dynamic analysis tools, giving information on which code region they perform, how they perform and the time cost.

2.5.3.1 PerfExpert

Introduction PerfExpert [30] is a performance analysis tool that automatically identifies and characterizes intrachip and intranode performance bottleneck, and also suggests solutions to reduce the impact of performance bottlenecks.

Figure 2.19 shows a comparison between two workflows. A typical optimization process that uses a performance evaluation in which only the measurements are automated, and the PerfExpert workflow. In generic profiling tools, an iterative process is followed involving multiple stages. The process is generally conducted manually and the decision making is left to the user performance evaluation and system knowledge. In contrast, most of these stages are automated in PerfExpert, such as the measurement and bottleneck detection.

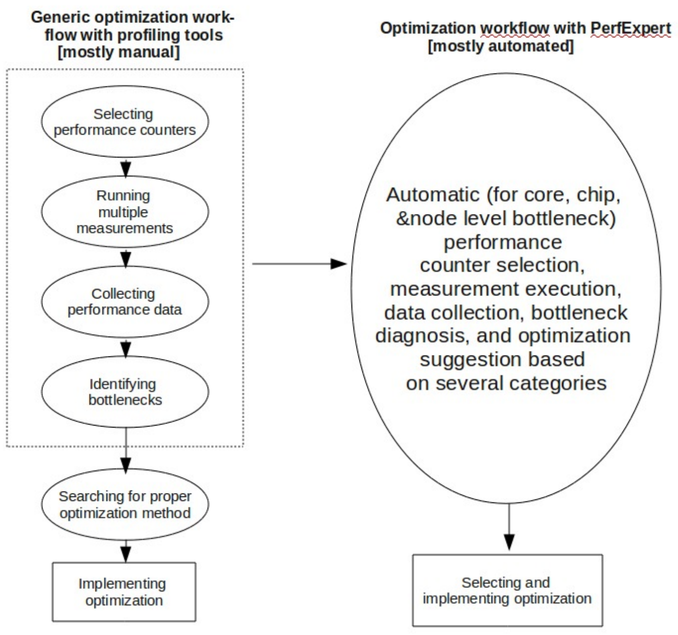

Generic optimization work-flow with profiling tools [mostly manual]
Selecting performance counters
Running multiple measurements
Collecting performance data
Identifying bottlenecks
Searching for proper optimization method
Implementing optimization
Optimization workflow with PerfExpert [mostly automated]
Automatic (for core, chip, &node level bottleneck) performance counter selection, measurement execution, data collection, bottleneck diagnosis, and optimization suggestion based on several categories
Selecting and implementing optimization

Figure 2.19: Profiling and optimization workflow with generic measurement tools (left) and with PerfExpert (right) [30].

PerfExpert combines a user interface with an analysis engine that detects bottlenecks in cores, sockets and nodes on loops and routines. PerfExpert employs the existing measurement tool HPCToolkit [73] to execute a structured sequence of performance counter measurements.

PerfExpert automatically evaluates the core, chip, and node performance. It determines which performance counters to use, analyzes the results, detects potential bottlenecks, and gives some hints for optimization. It presents the *LCPI for Local cycle-per-instruction* metric that combines performance counter measurements with architectural parameters to make the measurements comparable.

PerfExpert metrics The most important metric computed by PerfExpert is the *LCPI* for *Local Cycle Per Instruction*. PerfExpert computes the *LCPI* for each loop and routine and also an upper bound of the latency caused by the measured *LCPI* contribution for the six following categories: data memory accesses, instruction memory accesses, floating-point operations, branches, data Translation Lookaside Buffer (TLB) accesses, and instruction TLB accesses.

Benefit of the metrics The performance metrics of PerfExpert has the following benefits and abilities:

1. PerfExpert performance metric can combine, for example, the information about the hits and the misses for every cache level, into a single metric: data

access LCPI. This reduces the amount of output and summarizes PerfExpert results.

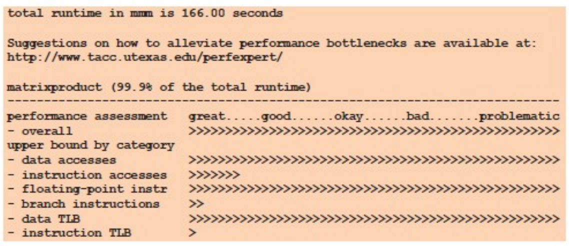
2. PerfExpert metrics can be refined if there are new performance counters available, such as counters for the shared L3 cache. This refinement has a cost. Because to have a new metric, PerfExpert needs a new hardware counter, this increases the number of runs to do to collect all the metrics because.

3. PerfExpert is extensible for different CPU generation and thus for new instructions and events related to these instructions.

The metrics proposed in PerfExpert are based on hardware counters. It uses HPCToolkit to execute a sequence of performance counter measurements. HPCToolkit uses a statistical sampling of hardware performance counters. The drawback of sampling is that it deal with several instructions in flight which leads to unable to assign a cost to the right instruction.

Output and optimization PerfExpert generates an output for each routine and loop. Figure 2.20 shows the output for a matrix-matrix multiplication. The first line of the output shows the name of the measurement file and its runtime. The end of the output corresponds to the performance evaluation.

For each routine and loop, PerfExpert lists the name and the percentage of the total runtime that the routine/loop represents. This is followed by the performance evaluation. The length of the ">" shows how bad/great are the performance. A lot of ">" show bad performance. We can see that the overall performance evaluation is problematic in the matrix-matrix multiplication. The rest of the assessment is the *LCPI* for each of the six categories: data memory access, instruction memory access, floating-point instructions, branch instructions, data TLB access, and instruction TLB access.



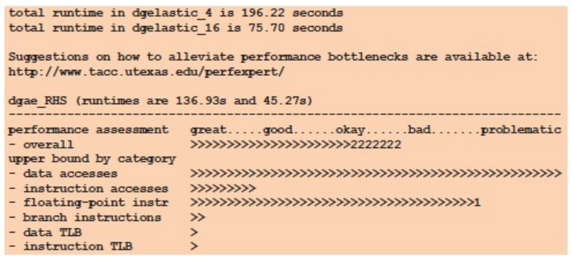
```
total runtime in mmm is 166.00 seconds

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

matrixproduct (99.9% of the total runtime)
---------------------------------------------------------------------------
performance assessment  great.....good......okay......bad.......problematic
- overall               >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses         >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses  >>>>>>>
- floating-point instr  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions   >>
- data TLB              >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction TLB       >
```

Figure 2.20: PerfExpert output for matrix-matrix multiplication [30].

We can see that for the matrix-matrix multiplication, the instruction memory and TLB accesses and the branch instructions are not problematic. However, the data miss in the cache and TLB as well as floating-point instructions are evaluated as problematic.

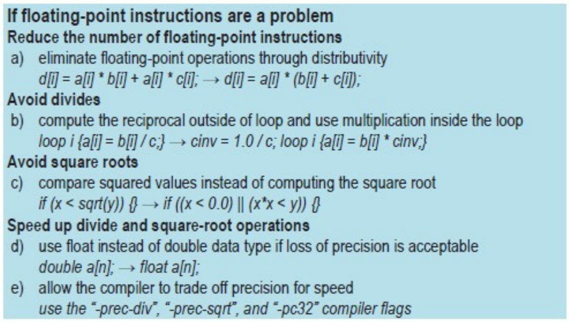
PerfExpert can also generate a combined output between two inputs. For example an input with one thread per chip and another input with four threads per chip. Figure 2.21 shows the output for `DGELASTIC`, a global earthquake simulation code based on the `MANGLL` [29] library with these two inputs.

PerfExpert also provides in the output, the path to proposed optimizations. They are accessible through a web page, that provides classic code transformations and compiler flags for each assessment.



```
total runtime in dgelastic_4 is 196.22 seconds
total runtime in dgelastic_16 is 75.70 seconds

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

dgae_RHS (runtimes are 136.93s and 45.27s)
----------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>2222222
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>1
- branch instructions    >>
- data TLB               >
- instruction TLB        >
```

Figure 2.21: PerfExpert output for DGELASTIC correlating two runs [30].



If floating-point instructions are a problem

Reduce the number of floating-point instructions

a) eliminate floating-point operations through distributivity
*d[i] = a[i] * b[i] + a[i] * c[i]; → d[i] = a[i] * (b[i] + c[i]);*

Avoid divides

b) compute the reciprocal outside of loop and use multiplication inside the loop
*loop i {a[i] = b[i] / c;} → cinv = 1.0 / c; loop i {a[i] = b[i] * cinv;}*

Avoid square roots

c) compare squared values instead of computing the square root
*if (x < sqrt(y)) {} → if ((x < 0.0) || (x*x < y)) {}*

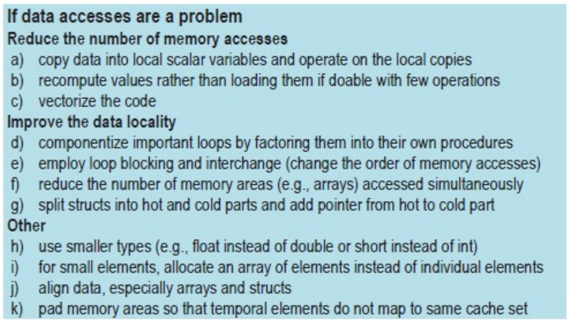
Speed up divide and square-root operations

d) use float instead of double data type if loss of precision is acceptable
double a[n]; → float a[n];

e) allow the compiler to trade off precision for speed
use the "-prec-div", "-prec-sqrt", and "-pc32" compiler flags

Figure 2.22: Simplified list of optimizations with examples [30].

Figure 2.22, shows the examples of code transformations in case the floating-point instructions are the problem, and Figure 2.23, shows the code optimizations proposed when the data access is the problem.



If data accesses are a problem

Reduce the number of memory accesses

a) copy data into local scalar variables and operate on the local copies
b) recompute values rather than loading them if doable with few operations
c) vectorize the code

Improve the data locality

d) componentize important loops by factoring them into their own procedures
e) employ loop blocking and interchange (change the order of memory accesses)
f) reduce the number of memory areas (e.g., arrays) accessed simultaneously
g) split structs into hot and cold parts and add pointer from hot to cold part

Other

h) use smaller types (e.g., float instead of double or short instead of int)
i) for small elements, allocate an array of elements instead of individual elements
j) align data, especially arrays and structs
k) pad memory areas so that temporal elements do not map to same cache set

Figure 2.23: Simplified list of optimizations without examples [30].

Conclusion PerfExpert is a tool for performance evaluation that uses user interface to give some hints for optimization to the tool's user. PerfExpert is based on hardware performance counter measurements using HPCToolkit. When the hardware counter measurements can be useful in routine level, in loop and instruction level it is different. Indeed, the statistical sampling used to collect information from the hardware counters, is not precise when comes to instruction level. It deals with several instructions in flight. Moreover, to increase the number of metrics in PerfExpert, there is a need to measure more hardware counters. In the Core i7, there are up to 1200 hardware counters and getting measurements from them requires a great number of runs.

Likwid (*Like I knew what I am doing*) [12] is a project that contributes to use command line tools for Linux to support programmers in developing high performance multi threaded programs. It contains several tools as Likwid topology, Likwid bench, and Likwid perfCtr.

While there are already a number of tools to measure hardware performance counters (as VTune or PerfExpert) a simple lightweight command line tool was still missing. There are two ways to use Likwid perfCtr: from outside as a wrapper to the target application and without changing the source code. Or with code markers inside the source in combination with the wrapper to configure the events to measure.

Whether we want to perform a fine-grained analysis at loop and instruction level, the hardware performance counter measurements with statistical sampling seems to have limitations (overhead, accuracy at instruction level). They can be used as a first step to detect the most time-consuming function then let more fine-grained tools profile the loop and the instruction levels.

2.5.3.2 TAU

TAU Performance System [90, 89] is a portable profiling and tracing framework for performance analysis of parallel programs written in Fortran, C, C++, Java, Python. The *TAU* framework architecture is organized into three layers: instrumentation, measurement, and analysis. Within each layer there are multiple modules and they can be configured in a flexible manner under user control.

Instrumentation *TAU* supports a flexible instrumentation model that allows the user to insert performance instrumentation calling the TAU measurement API at different stages such as, multiple levels of program code representation, transformation, compilation, and execution. The API also provides selection of profiling groups for organizing and controlling instrumentation.

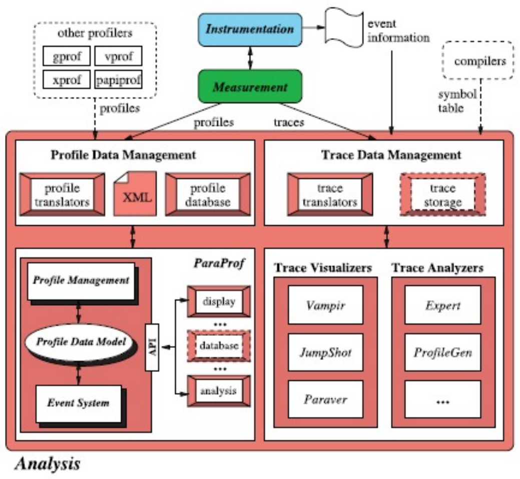


Figure 2.24: Architecture of *TAU* Performance System. Analysis and Visualization [90].

TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of several locations: functions, methods, basic blocks, and statements. All C++ language features are supported including templates and namespaces.

The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (*PDT*), dynamically using *DyninstAPI*, at runtime in the Java virtual machine, or manually using the instrumentation API.

PDT [69] is used to parse the source code and to determine the semantic constructs to instrument. Figure 2.25 shows the purpose of *PDT*. *Dyninst* [3, 28] is used to instrument dynamically the binary (ie. the executable) and permits to insert code snippets for instrumentation.

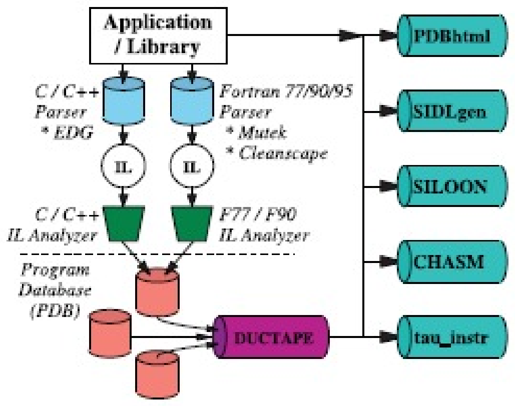


Figure 2.25: Program Database Toolkit *PDT* [90].

Visualization *TAU*'s profile visualization tool, *ParaProf*, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. Figure 2.26 shows the *ParaProf* design. Moreover, *TAU* can generate event traces that are displayed with the Vampir [11], Paraver [84] or JumpShot trace visualization tools.

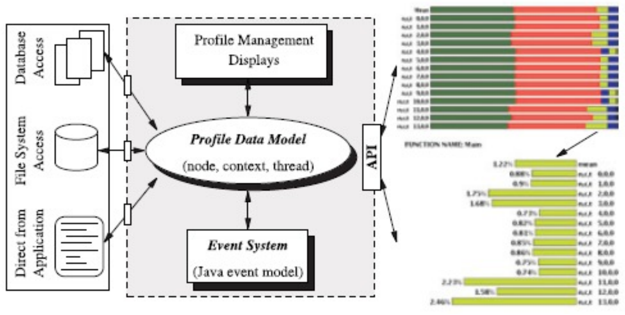


Figure 2.26: ParaProf Architecture [90].

Conclusion The *TAU* framework addresses the performance problems at three levels: the instrumentation, the measurement, and the analysis. It provides the instrumentation's at different levels, and it performs tracing on parallel programs. It allows an interactive performance analysis and data management. *TAU* framework is the product of many years (20 years) of development and it is available on HPC platforms and supports mots of the parallel programming methodologies. However, when goes to loop profiling, *TAU* is more grosgrain. It does not detect loops, and actually to instrument loops, we need to insert some probes explicitly in the source code. *TAU* can be used as a first step in a performance analysis process, to detect the most executed routine.

Other tools for parallel applications exit such as: KOJAK [75], a *Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks*, Paradyn [74], a performance measurement tool for parallel and distributed program, based on Dyninst [3]. Parallel Performance Wizard [95], is a performance analysis system for PGAS [8] and MPI application analysis.

Active Harmony [36, 33] is an automated runtime tuning system. It requires adaptation of the application and is mostly concerned with distributed resource environments.

Acumem AG [1] sells the commercial products *ThreadSpotter* (multithreaded applications) and *SlowSpotter* (singlethreaded applications), which capture information about data access patterns and offer advice on related losses, specifically latency exposed due to poor locality, competition for bandwidth, and false sharing. SlowSpotter and ThreadSpotter also recommend possible optimizations.

2.5.3.3 LoopProf-LoopSampler

Introduction Moseley *et al.* proposed in [76] the concept of loop-centric profiling to give the programmer a more complete view of where time is spent in a program. This concept is based on loop profiling which can be performed by two approaches. An instrumentation-based approach that gathers interesting information about the loop behavior performed with *LoopProf.* It uses a dynamic stack-based algorithm to detect and account for loops. The second approach is a sampling approach with *LoopSampler* that achieves similar results than *LoopProf* with negligible overhead.

LoopProf *LoopProf* algorithm, was presented the first time in [77]. It is based on the Pin [71] binary instrumentation. That means that *LoopProf* works transparently on different architectures binaries (e.g.,Intel ARM, IA32, EM64T, and Itanium architectures).

LoopProf is based on binary instrumentation and is compiler independent. It does not assume a known Control Flow Graph. To detect loops, *LoopProf* bases itself on the Pin definition of a BBL (ie. Basic Block). To discover loops, *LoopProf* traces the BBLs during the execution, and a stack of BBLs is kept to represent the execution path. Each time a BBL is encountered, the BBL is pushed on the stack. If a BBL has already been encountered, then it will be marked as a head of a loop.

To avoid detecting recursion with the *LoopProf*, the tool keeps track of the function call stack as well and the detection of loops is limited to the BBLs of the same frame.

Concerning the instruction accounting, this is done at each end of an iteration of the loop before the BBL is popped off.

In case of multithreaded programs, *LoopProf* maintains a separate context for each thread at runtime, and, at the end of the execution, per-thread and aggregate statistics are reported.

This approach of loop detection (ie *LoopProf*), because it is compiler independent, has some limitations such as an overhead of over 70 times slower than the original execution.

LoopSampler *LoopSampler*, like *LoopProf*, also uses the Pin runtime. However, instead of using the dynamic instrumentation, it uses Pin's "probes" to run an application and takes control of the application only when certain events occur.

LoopSampler periodically interrupts the application and reads the call stack to determine which function or loop is active. *Loop Sampler* needs a periodic interrupt to read the contents of the call stack. One solution consists in implementing it directly in the kernel, but this would not be portable. A second approach consists in using the Unix function, `setitimer`.

When an application is initialized, *LoopSampler* registers a signal handler for the specified signal. Then, the signal is sent back at a specific frequency.

When an application is running, *LoopSampler* registers a callback with Pin to be notified when a new binary image is loaded. Each time, a new image is loaded, this binary image is parsed, a partial Control Flow Graph is built, and loops are detected using a traditional dominator analysis (unlike *LoopProf* that has no knowledge of the CFG).

LoopSampler is good at identifying hot outer loops, but lacks some of the detailed information that the, slower *LoopProf* offers.

Results on SPEC Both *LoopProf* and *LoopSampler* are tested on a set of the SPEC2000 benchmarks [93]. Figure 2.27, shows the overhead of *LoopProf* when performed on the SPEC2000, comparing to three cases: native run, Pin running the program without instrumentation and a basic block counting Pintool. According to the author, as most of programs do not require iterative profiling, the overhead of *LoopProf* is reasonable for users that want more details about the loops.

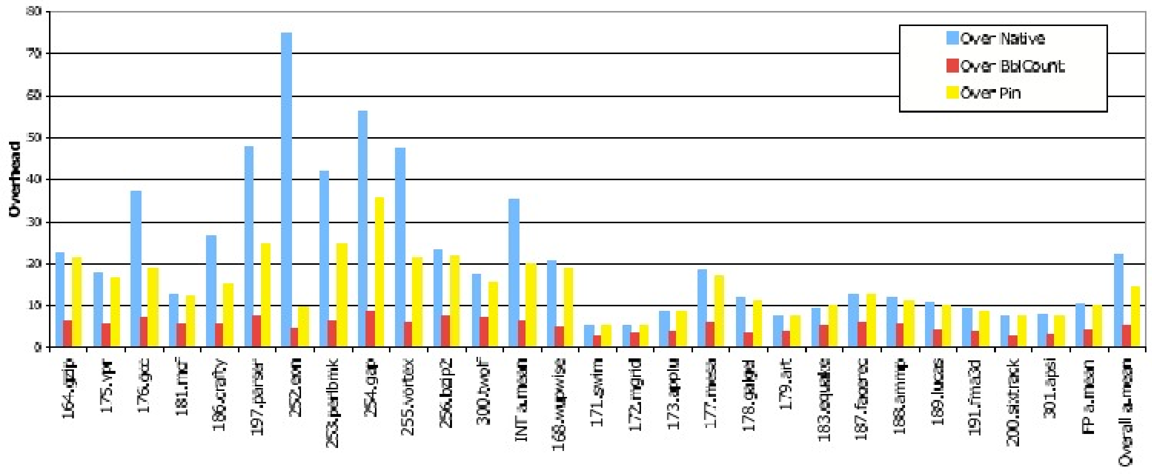


Figure 2.27: The runtime overhead of the *LoopProf* tool when compared to applications run natively, executed with no instrumentation under Pin, and instrumented by the BblCount Pintool [76].

Conclusion Loops are common target for parallelization but without enough information about the loop behavior, it would be difficult to know which loop to parallelize. *LoopProf* and *LoopSampler* are two techniques to collect the most information about the loop behavior to make easier the choice of which loop to parallelize. The instrumentation-based profiling performed with *LoopProf*, gives a lot of details about the loops but is limited because of an overhead of over 70 times slower then the original execution. This overhead can be avoided while using *LoopSampler* but with a less detailed information about the loops. The user should make a tradeoff between execution time and accuracy.

Another known tool for instrumentation, Valgrind [10]. It is an instrumentation framework for building dynamic analysis tools. Valgrind uses just-in-time compilation to enable dynamic instrumentation. When using Valgrind, the original instructions never run on the host processor. Instead, they are converted to an intermediate representation so that other tools can easily and directly manipulate them, and then is recompiled to the target architecture. Valgrind includes several sub-tools, including a memory error detector, cache profiler, heap profiler, and more. Cachegrind is the cache profiler. It is based on the simulation of configurable L1I, L1D and L2 caches. It identifies the number of cache misses for each line of the source code, with per-function, per-module and whole-program summaries.

Since *LoopProf*, *LoopSample* and Valgrind perform on binary codes, another tool on which the two first tools are based on: *Pin*. It is a tool for dynamic instrumentation of programs. It does not rewrite the code, it injects code so the application can keep on working. This method of instrumentation allows to avoid alteration of the semantic of the code. However, injecting code in flight, has a significant overhead noticed in the instrumentation-based tool, *LoopProf*.

2.5.3.4 VTune

VTune Performance Analyzer [4] is a tool developed by Intel also based on sampling. It runs on x86 architectures and allows a programmer to profile code and extract various performance data about the code's behavior. VTune operates in three main modes: event-based sampling, counter monitoring and call-graph profiling. Event-based sampling is a popular, fast and transparent mode to analyze a program using performance counters. In this mode, the VTune analyzer collects data from the processor using regular timer interrupts. These interrupts are issued when a threshold of events is reached (e.g., instructions retired). This number depends on the sampling rate specific to the hardware event and determines the analysis' precision.

2.5.3.5 PTU

Performance Tuning Utility [15] is an Intel tool developed to target some traditional features such as identifying the hottest modules and functions in a whole application, tracking call sequences. PTU also offers the processor hardware counters for in-depth analysis of the memory system and architectural tuning. PTU also associates performance issues with the source code. If no symbol sources for an analyzed application are found, PTU represents data with basic block granularity and provides a graph of the function execution flow (control flow graph) to navigate the disassembly.

	Target Code	**Granularity**	**Vectorization**	**Front- Back-end**	**Perf. Estimation**	**Semantic**	**Overhead**	**GUI**
Maqao	Binary	Loop	+	+	+	-	+	+
IACA	Source	Marked code section	-	+	-	-	+	-

Table 2.1: A comparison between Maqao's static analysis features and *IACA*'s features.

	Target Code	**Granularity**	**Loads/Stores**	**Cache misses**	**Semantic**	**Overhead**	**GUI**
Decan	Binary	Loop/Instruction	+	-	-	+	-
PerfExpert	Binary	Function/Loop	-	+	+	-	-
VTune	Binary	Function/Loop	-	+	+	-	+
LoopProf	Binary	Loop	-	-	+	-	-
Pin	Binary	Function/Loop	-	-	+	-	-

Table 2.2: A comparison between Decan features and some dynamic profiling tools' features.

2.6 Comparison of tools

In this section we gather the tools performing the same analysis in a table to make a comparison with our tools developed in this thesis.

Table 2.1 makes a comparison between Maqao's static analysis and the Intel static analysis tool *IACA*. The advantage of Maqao is that it performs on loop level on binary codes. There is no need to change the source code to extract performance data. Moreover, Maqao's static analysis gives information about the degree of vectorization of the loop, which is an important data, since the vectorization is a special case of parallelism. *IACA* has an ASCII output when Maqao is based on web interface. However, Maqao can be used in command line for scripting for expert users. Both *IACA* and Maqao are static tools, they do not consider the dataset input, thus the semantic.

Table 2.2 make a comparison between Decan, our decremental analysis tool and some of the dynamic analysis tools described previously. The table shows that there are no tools which are focusing on load/store instruction level. Moreover, we can notice that Decan does not have overhead comparing to others tool based on statistical sampling or dynamic instrumentation. Even if Decan alterates the semantic of the code, it is all about gathering more performance data. Decan is a command line tool, that can be scripted in case of a big amount of binaries to analyze.

This comparison shows that there are many useful performance tools that target different sides of the performance analysis. Our tools Maqao's static analysis and Decan are two tools that bring more new concepts for more information about the program behavior.

Out tools are not considered as a panacea, they are additional tools that can give additional performance information to apply the best optimization.

2.7 Conclusion

In this chapter we showed how the increasing complexity of the microarchitecture and the memory leads to about more complex and dedicated performance analysis tools.

The purpose of all these performance evaluation tools is to detect where, in a code, is the bottleneck. Such a bottleneck, can be anything in the code: memory saturation, vectorization, bad memory pattern access, branch misses, etc. Numerous tools and techniques exist, but no single tool is a panacea; instead, different tools have different strengths. Therefore, an assortment of performance tuning utilities and strategies are necessary to best utilize scarce resources (e.g., bandwidth, functional units, cache).

The next chapters present the performance analysis tools developed during my thesis. The first one, is MAQAO static analysis, a performance tool the performs on binary and allows to predict the performance of a code, on Core2 architecture. It is based on the performance model of the Core 2 architecture and gives information about the quality of the code. The second tool, DECAN, is a new approach to performance evaluation, that also performs on binary, and allows to give the impact on performance of the memory accesses to pinpoint the delinquent instruction.

Chapter 3

MAQAO Static Analysis Tool

3.1 Introduction

Modern processors rely on many complex hardware mechanisms in order to reach high levels of performance. In particular, the use of the levels of parallelism and the appropriate use of the memory hierarchy to hide large memory latencies are both required to obtain the full computing capacity of processors. This road to high performance is paved with many complex compiler optimizations. It uses, according to the code, prefetching mechanisms, vectorization, loop transformations for better cache usage or data layout restructuring. While many optimizing compilers are able to perform all these transformations, they have a poor knowledge of the application context and must be conservative in their transformations. Failing to find the best optimization sequence for a given application code, leads compiler to generate programs with poor performance, or with inappropriate code.

The performance tuning process therefore guides the compiler, through pragmas, compilation flags, or source to source restructuring, to generate a better code. Many approaches to performance tuning have been proposed, getting feedback from the application either by collecting execution traces through instrumentation (with Dyninst [3] or Pin [9] for single processors, with Scalasca [106] for multi-node systems) or hardware counters values (such as Intel Vtune [4] or PTU [15] for instance). Hardware counter-based techniques show how the architecture behaves with the considered code and input set. However, it is difficult to make the connection between hardware event counts and source code, since both source code and compiler optimizations have an impact on the resulting hardware events. Moreover, there is no direct link between hardware counters and the quality of the compiler generated code. To have feedback from the compilation process, it is, generally, necessary to analyze performance from the assembly generated code.

In this chapter, we describe how Maqao [41] (*Modular Assembly Quality Analyzer and Optimizer*) handles static performance analysis on real-life programs. Our target architecture is the Core2, but the tool can be easily retargeted to other x86 architectures essentially by changing the performance model used. As Maqao has a plugin construction, targeting other architectures requires modifying the concerned plugins: an earlier version of Maqao targeted IA64 architectures which are very different from X86. The static performance evaluation provides hints on how to improve the compilation process, and assess the amount of performance that could be obtained through optimization. This estimate is performed on the sequential codes (number of threads equal to 1). Improving unicore performance (both in the sequential and the parallel part of the programs) contributes to improving global performance and efficiency of the code. We show in particular how static performance evaluation is achieved on the Core 2 architecture.

This chapter is organized as follows: Section 3.2 gives an overview of the Maqao framework. Section 3.3 describes how a target code is restructured to be analyzed.

Section 3.4 presents the performance model implemented. Section 3.5 applies the MAQAO's static analysis on the Numerical Recipes. Finally, Section 3.6 applies the MAQAO's static analysis to real-life applications.

3.2 MAQAO Framework

MAQAO, a Modular Assembly Quality Analyzer and Optimizer, is a tool for analyzing and optimizing assembly code. Its principles and its general organization are fairly generic and are capable of supporting a large number of target processors and compilers. At this point, MAQAO supports the Itanium (`IA64`) [55] and the Pentium (`x86`) [61] architectures, with `ICC`, the Intel C Compiler [62],`IFORT`, the Intel Fortran Compiler [62], and the `GCC` compiler [50].

MAQAO handles performance analysis and memory tracing for OpenMP programs. It combines static analysis of compiler-generated binary code with the analysis of execution traces and binary instrumentation. Static performance evaluation provides hints on how to improve the compilation process, and assess the amount of performance that could be obtained through optimization. This static analysis corresponds to the design of the performance model of the Core 2 architecture. It is performed on sequential programs (both in the sequential and the parallel part of the programs). Focusing on the unicore performance, and improving the sequential performance has an impact on the global performance of a program. For instance, dynamic thread-wise traces, in particular compact memory traces, show how to improve interactions between threads, and detect false sharing situations. MAQAO provides the following features:

- Intelligent navigation and flexible automated analysis (either predefined or user defined) of binary code.
- Quality assessment of the code generated and detection of potential inefficiencies of the assembly code either statically or dynamically (value profiling). To predict performance, MAQAO has a performance model to identify the deliverable performance of the application and the individual contributions of several factors to the performance degradation.
- Generation of hints and guidelines to drive the optimization process.
- Building of an evolving database of known performance issues on the target architecture (currently Itanium 2 and Pentium).

MAQAO is composed of different modules. Figure 3.1 shows the MADRAS, MAQAO core, plugins (developed in Lua script language [6]) and the user web interface modules.

- MADRAS [100] is a library for disassembling and instrumenting binary files. MADRAS uses a grammar associating binary expressions to assembly instructions, similar to `yacc` grammars, and generates a corresponding disassembler, using a linear-sweep method (similar to `objdump`). This disassembler for `x86` is then used by MAQAO.
- MAQAO core is a program that restructures the list of instructions built by MADRAS into a call graph and builds the control flow graphs of the different functions exposed.

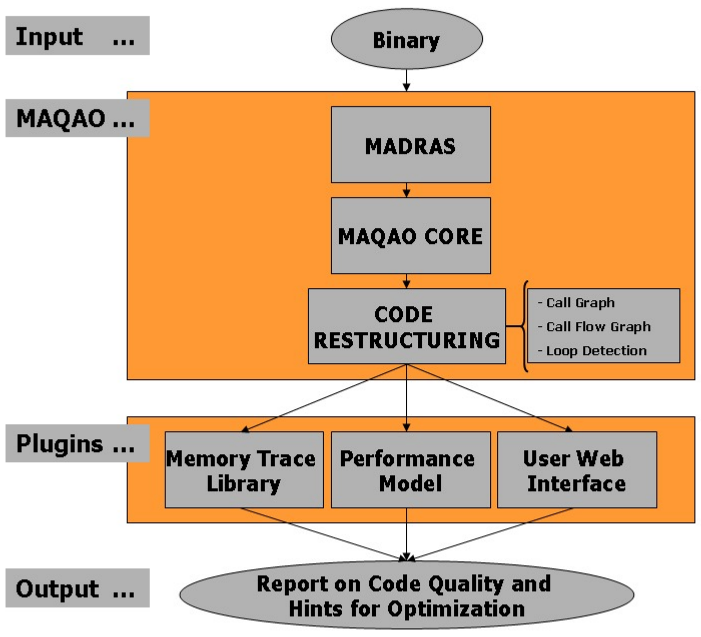


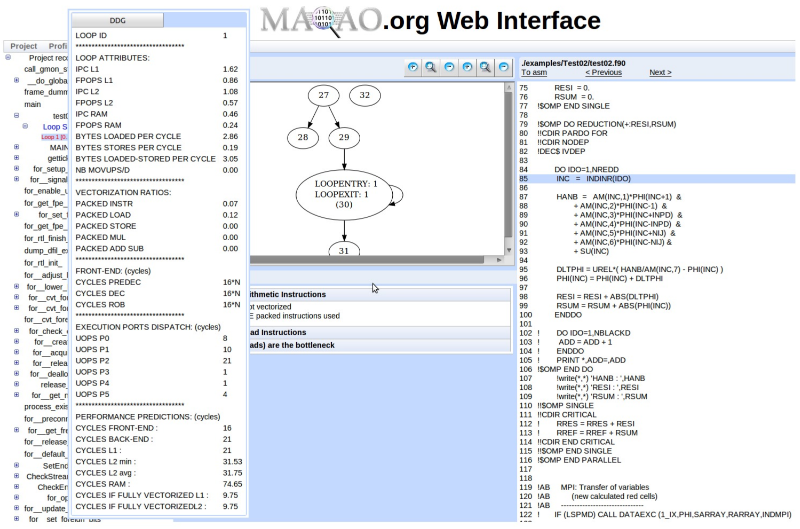
Figure 3.1: MAQAO framework with the connexion between different modules.

- The plugins are a convenient mechanism for the rapid prototyping of assembly analysis. With an API to access the structures built by MAQAO core and instrumentation capabilities of MADRAS, this mechanism extends MAQAO capabilities through scripting. There are many scripts available, ranging from batch analysis to a web service for the user web interface. The plugins are written in LUA. The majors plugins are:
  - Performance Model: this consists in a set of plugins which evaluate the performance of inner loops and provides performance improvement hints. This is the work done in the thesis as a first contribution.
  - Memory trace library: this plugin instruments multithreaded codes to trace memory accesses and analyzes them.
- A user web interface is proposed for a more intuitive interaction with MAQAO. Requiring no installation, this interface enables the analysis of codes uploaded on remote servers.

Figure 3.2 shows an example of the user web interface.

3.3 Code Restructuring

MAQAO exploits static binary rewriting for reading and instrumenting executables. Static binary rewriting refers to post-link time manipulation of binary executables. This approach has the advantage, compared to approaches requiring compiler interaction (analysis of assembly code) or inclusion of libraries, to obviate the need of recompiling or relinking. The API for reading and manipulating static binary files is defined by MADRAS [100]. The disassembled binary code is then restructured: call graphs, control flow graphs, loops, and dependence graphs on registers are built.



```
LOOP ID                                1
***********************************
LOOP ATTRIBUTES:
IPC L1                                 1.62
FPOPS L1                               0.86
IPC L2                                 1.08
FPOPS L2                               0.57
IPC RAM                                0.46
FPOPS RAM                              0.24
BYTES LOADED PER CYCLE                 2.86
BYTES STORES PER CYCLE                 0.19
BYTES LOADED-STORED PER CYCLE          3.05
NB MOVUPS/D                            0.00
***********************************
VECTORIZATION RATIOS:
PACKED INSTR                           0.07
PACKED LOAD                            0.12
PACKED STORE                           0.00
PACKED MUL                             0.00
PACKED ADD SUB                         0.00
***********************************
FRONT-END: (cycles)
CYCLES PREDEC                          16*N
CYCLES DEC                             16*N
CYCLES ROB                             16*N
***********************************
EXECUTION PORTS DISPATCH: (cycles)
UOPS P0                                8
UOPS P1                                10
UOPS P2                                21
UOPS P3                                1
UOPS P4                                1
UOPS P5                                4
***********************************
PERFORMANCE PREDICTIONS: (cycles)
CYCLES FRONT-END :                     16
CYCLES BACK-END :                      21
CYCLES L1 :                            21
CYCLES L2 min :                        31.53
CYCLES L2 avg :                        31.75
CYCLES RAM :                           74.65
CYCLES IF FULLY VECTORIZED L1 :        9.75
CYCLES IF FULLY VECTORIZEDL2 :         9.75
```

```
          RESI  = 0.
          RSUM  = 0.
!$OMP END SINGLE

!$OMP DO REDUCTION(+:RESI,RSUM)
!!CDIR PARDO FOR
!!CDIR NODEP
!DEC$ IVDEP

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

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

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

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

!         DO IDO=1,NBLACKD
!          ADD = ADD + 1
!         ENDDO
!         PRINT *,ADD=,ADD
!$OMP END DO
           !write(*,*) 'HANB : ',HANB
           !write(*,*) 'RESI : ',RESI
           !write(*,*) 'RSUM : ',RSUM
!!$OMP SINGLE
!!CDIR CRITICAL
!         RRES = RRES + RESI
!         RREF = RREF + RSUM
!!CDIR END CRITICAL
!!$OMP END SINGLE
!$OMP END PARALLEL


!AB    MPI: Transfer of variables
!AB        (new calculated red cells)
!AB    -------------------------------
!       IF (LSPMD) CALL DATAEXC (1_IX,PHI,SARRAY,RARRAY,INDMPI)
```

Figure 3.2: MAQAO user web interface.

3.3.1 Call graph

Figure 3.3 shows a call graph built by MAQAO. A call graph is a directed graph that represents the calls to routines in a program. Each node in a call graph corresponds to a routine/function/procedure. An edge between two routines `e(f,g)` means that the routine `f` calls the routine `g`. If a cycle is detected in a call graph, that means that there is a recursive call. In MAQAO, the call graph construction uses labels found in the binary, if any.

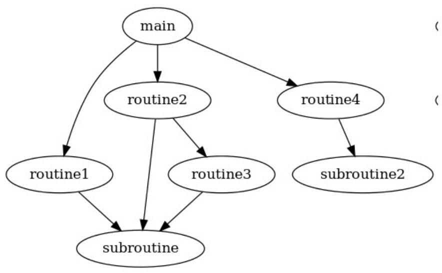


Figure 3.3: A MAQAO call graph for a basic program.

3.3.2 Control flow graph

Figure 3.4 shows a control flow graph built by MAQAO. A control flow graph is a graph representation of all paths that might be taken in a program during its execution. Each node in a control flow graph corresponds to a *basic block*. In MAQAO the building of the control flow graph is limited by indirect jumps that may prevent from finding a correct control flow.

3.3.3 Data dependence graph

Figure 3.5 show a data dependence graph built by MAQAO. The data dependence graph is a graph representation of the dependency between objects. In MAQAO it represents the dependency between registers. Each node of the data dependence

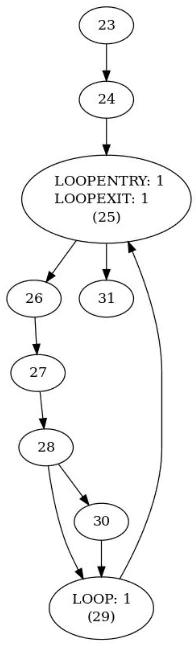


Figure 3.4: A MAQAO control flow graph for a basic program.

graph corresponds to an instruction. Two nodes are connected with an edge, if any register dependency exists between the two nodes.

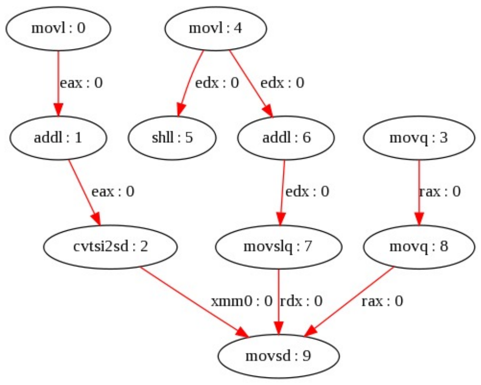


Figure 3.5: MAQAO data dependence graph for a basic program. A node represents an instruction. An edge corresponds to a register dependency between two nodes. Each edge is tagged with a register that is the subject of the dependency, and the dependency distance.

3.3.4 Loop detection

MAQAO's loop detection is based on the dominance algorithm of Cooper *et al.* [35]. It is a technique for computing dominators on a control flow graph. The algorithm builds on the well-developed and well-understood theory of iterative data-flow analysis.

Considering that the dominance has been computed, the loop detection algorithm can be easily implemented. MAQAO implements the natural loop detection algorithm detailed by S. Muchnick in his book [81].

3.4 Performance Model [24, 66]

Computer modeling is now a well-known approach to simulate an *abstract model* of complex systems. It is used to explore the behavior of these complex systems and to estimate/predict their performance.

The performance model of MAQAO computes performance estimates based on the assembly code. It evaluates the cycles required for executing innermost loops. The reason for considering only the innermost loops is that they usually constitute the most time consuming part of the code. The x86-Core2 architecture model we consider takes into account the front-end pipeline (decoding, permanent register file allocation, special microcoded instructions), the different ports for the execution units [49], and the latencies of memory instructions. For memory instructions, several latencies are considered, according to the location of the data in the memory hierarchy. The evaluation provides an optimistic bound, meaning that the real code may execute itself in more cycles due to some extra latency not taken into account by our static model.

MAQAO's performance model (ie. the static analysis) collects several metrics for a target program, in order to predict its performance on a Core2 architecture.

3.4.1 The Core 2 microarchitecture

The Intel Core2 microarchitecture is an x86 microarchitecture and an extended design of the Pentium M microarchitecture. There are two or more CPU cores in Core2 with a shared L2 cache and separate L1 caches. The Core2 has an *out-of-order* execution, the pipeline has been expanded to handle 4 micro-operations per clock cycle, and the execution units have been expended from 64 bits to 128 bits. The reorder-buffer has 96 entries and the reservation station has 32 entries. Figure 3.6 illustrates these features [49].

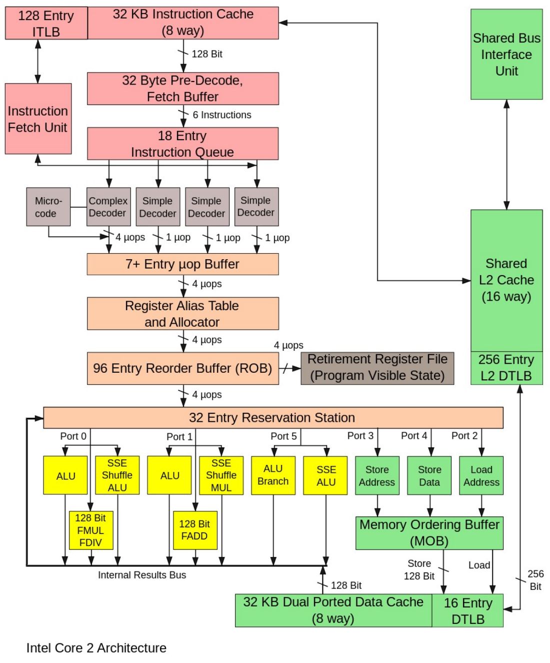


Figure 3.6: Intel Core2 microarchitecture. [2]

The mechanism of out-of-order consists in delaying the execution of an instruction if its input data is not ready yet and to find a later instruction that can be executed first if its input data is available. This can be done if the later instruction does not need the output of the delayed instruction. If all the instructions have such dependencies, then there is no opportunity for an *out-of-order* execution.

In the Core2, the instructions are fetched and decoded *in order*, then they wait in the reservation station queue. When the input data of an instruction is ready, the instruction is executed. After the execution, the result/output of the instruction is queued. The result is not written back in the register file (ie. retired) until all the older instructions have their results written back in the register file.

In previous *in-order* processors, stalls occur due to some dependencies that cannot be circumvented. With an *out-of-order* architecture, some mechanisms are added to allow dynamic scheduling of instructions and avoid the stalls:

- Instruction split
- Register renaming

Instruction split In an *out-of-order* microarchitecture, the instructions are translated into micro-instructions to better exploit the *out-of-order* mechanism. For example, the instruction `MUL [MEM],RAX`, can be split into two micro-operations. One for reading the memory and another one for the multiplication. For the instruction `ADD RAX,[MEM]`, three micro-operations are generated. One for reading the memory, a second one for the addition and a third one for writing the result in the memory. The benefit of this splitting, is that the micro-operations can be executed *out-of-order*. For example:

```
i     R1 <– R1 + MEM1
i+1   R1 <– 5 + R1
i+2   R1 <– R1 * MEM2
```

The instruction *i* is split into two micro-operations. One for fetching MEM1 and a second one for the addition. Instruction *i+2* is split into two micro-operations for fetching MEM2 and for the multiplication. With the *out-of-order* mechanism, fetching MEM2 can be done at the same time as the instruction *i+1* or even before if both MEM1 and MEM2 are not in the cache (they can be fetched at the same time).

Register renaming The register renaming mechanism allows to avoid some dependencies such as write-after-read (WAR) and write-after-write (WAW). Breaking such dependencies tends to exploit the *out-of-order* mechanism. In the following example:

```
i     R1 <– MEM1
i+1   R1 <– 6 + R1
i+2   MEM2 <– R1
i+3   R1 <– MEM3
i+4   R1 <– 3 * R1
i+5   MEM4 <– R1
```

The three last instructions are completely independent from the three first. The microprocessor detects this *independence* and replaces `R1` in these three last instructions with a temporary register `R20`. The result is as follows:

```
i    R1 <– MEM1
i+1  R1 <– 6 + R1
i+2  MEM2 <– R1
i+3  R20 <– MEM3
i+4  R20 <– 3 * R20
i+5  MEM4 <– R20
```

The Independence between the instructions is now clear and the mechanism of *our-of-order* is exploited. That means that if MEM1 and MEM3 are not in the cache, they can be fetched at the same time before the the addition *i+1*.

Pipeline When an instruction is executed in a Core2 processor, it crosses the following stages of the pipeline:

1. Fetch/Predecoder
2. Decoder
3. Reorder-Buffer read stage
4. Execution Units

The behavior of these previous stages has been implemented in our performance model. As they are, most of the time, considered as a bottleneck, their behavior has been implemented for performance prediction.

Algorithm 1 MAQAO Performance Model Algorithm.

```
Require: Binary = The target binary to analyze
Require: Asmfile = The disassembled binary
 1: MADRAS disassembles Binary and generates Asmfile
 2: for (each Function in Asmfile) do
 3:   for (each Loop in Function) do
 4:     Tpredec = Predecoding(Loop)
 5:     Tdec = Decoding(Loop Predecoded)
 6:     Trob = RoB-read(Loop Decoded)
 7:     Tbend = Back-End(Loop)
 8:     Tmetrics = Other-Metrics(Loop)
 9:     Gather the different times Ti with i in {predec, dec, rob, bend, metrics}
10:     Generate a performance analysis report for Loop
11:   end for
12: end for
```

Algorithm 1 shows the performance model implemented on MAQAO. The performance model is applied on each loop of each function of the target binary. For each loop, the behavior of each stage of the pipeline is *simulated.* First, the loop is predecoded (decomposed on blocks of full instructions), then the predecoded blocks are sent to the decoder that decodes the instructions of each predecoded block. After decoding, the reorder-buffer read stage, is the stage where the renamed registers (the mechanism of register renaming is applied after decoding) are stored. In the RoB-read stage, some stalls can occur depending on the minimum number of registers that can be read from the register file (see Chapter 2, Section 2.3.1.6. When an instruction has its operands ready, it is dispatched in its corresponding execution port.

The computation of all the different metrics, is independent from the front-end (predec, dec, RoB-read) and the back-end computation.

The following sections describe each function of the performance model.

3.4.2 Predecoding

The purpose of the predecoder is to detect where each instruction begins. The maximum throughput of the predecoder in the Core2 microarchitecture is 16 bytes per clock cycle (about 6 instructions). When a loop is predecoded, it is split into blocks of 16 bytes. Any instruction that crosses the 16-byte boundary is unhandled until the next 16-byte block is processed. Between the predecoder and the decoder there is a queue of 64 bytes called the *loop buffer*. If a loop is completely contained in the loop buffer, the throughput of the predecoder is up to 32 bytes per clock cycle. The predecoder takes as input a target loop and generates a list of blocks of instructions. Each block is a 16-byte or a 32-byte block depending on the size of the loop. The number of blocks corresponds to the number of cycles spent in the predecoder. Algorithm 2 explains the process of predecoding depending on the size of the input loop [49].

Algorithm 2 Predecoder Algorithm.

```
Require: Asmfile = The disassembled binary using MADRAS
 1: for (each Function in Asmfile) do
 2:   for (each Loop in Function) do
 3:     if (size of Loop > 64 bytes) then
 4:       Split Loop into blocks of 16 bytes max
 5:       Each 16-byte block contains full instructions
 6:     else
 7:       Split Loop into blocks of 32 bytes max
 8:       Each 32-byte block contains full instructions
 9:     end if
10:     Tpredec = number of 16-byte or 32-byte blocks
11:   end for
12: end for
```

3.4.3 Decoding

There are 4 decoders organized in a 4-1-1-1 pattern. The first decoder decodes 4 uops and the others one uop each per cycle. The input of the decoder is the predecoded blocks of 16 or 32 bytes. If the 16-byte block contains 4 instructions of 2 uops each, the maximal throughput of the decoder is 2 uops per clock cycle. The instructions that generate more than 4 uops use microcode ROM and takes several cycles to decode. Experimentally, we have found 2 cycles. The output of the decoder is a list of blocks of instructions where each block fit in a 4-1-1-1 pattern, decoded in 1 cycle [49]. Algorithm 3 illustrates the process of decoding implemented in MAQAO.

In the Core2 microarchitecture, there is a mechanism called *micro-operation* and *macro-operation* fusion. The *micro-operation* fusion consists in the fusion of two *micro-operations* of different instructions. The *macro-operation* fusion corresponds to the fusion of two instructions. These two mechanisms, when performed, allows to increase the throughput of the decoder [49].

These two concepts are not implemented in out algorithm, because there are not enough documentation about the list of the instructions that will be fused.

Algorithm 3 Decoder Algorithm.

```
Require: Lpredec = The list of 16 or 32 bytes predecoded blocks
Require: Ldec = The list of blocks decoded
1: for (each Block in Lpredec) do
2:   if (Block does not fit in 4-1-1-1 pattern) then
3:     Split Block into instructions blocks that fit in the 4-1-1-1 pattern
4:     Ldec = Ldec + number of blocks generated from the previous split
5:   else
6:     Ldec = Ldec + 1
7:   end if
8: end for
9: Tdec = Ldec
```

3.4.4 Reorder-buffer-read stage

The Reorder-buffer-read stage can not read more than 3 different registers from the permanent register file per clock cycle. The first two register read ports can read registers for instructions operands, base pointers, and index pointers. The third read port can only read registers for index pointers. If there are more than 2 registers for base pointers or instruction operands, a read-stall is generated. Registers that have been written recently can be read from the ROB-read if they have not yet passed the ROB-writeback stage. It takes approximately 5 clock cycles for a uop to pass from the ROB-read stage to the ROB-writeback stage. A register can be read without problem if it has been modified within the last 5 clock cycles [49]. Algorithm 4 shows how the process described below, has been implemented in MAQAO.

3.4.5 Back-end

Figure 3.7 shows the execution units in the Core2 microarchitecture. Each port is dedicated to an execution unit. In the Core2, the ports are dedicated to [49]:

- P0-P1-P5: computation and branches
- P2: memory accesses (loads)
- P3: memory accesses (addresses stores)
- P4: memory accesses (data stores)

Each instruction has its corresponding execution ports depending of the type of the instruction (fp or int) and the type of its operands (register, memory, or address).

For each execution port, MAQAO computes an estimation of the number of cycles spent on each port. The performance estimate takes into account the special case of instructions that are split into different micro-operations to be executed on multiple ports [49]. When an instruction (or a micro-operation) can be executed on different ports (a common example is simple integer instructions which can be assigned indifferently to P0, P1 and P5), the less saturated port is chosen. Since

Algorithm 4 Reorder-Buffer read Algorithm.

```
Require: Ldec = The list of instructions blocks decoded
Require: Tstall = The number of stalls occurred
 1: for (each Block in Ldec) do
 2:   for (each Register in Block) do
 3:     if (Register is a base register that is not written within 5 cycles ) then
 4:       Base-Stall = Base-Stall + 1
 5:     else
 6:       if (Register is an index register that is not written within 5 cycles)
          then
 7:         Index-Stall = Index-Stall + 1
 8:       end if
 9:     end if
10:   end for
11: end for
12: if (Base-Stall > 2) then
13:   Tbstall = Tbstall + (Base-Stall/2) + (Base-Stall mod 2)
14: end if
15: if (Index-Stall > 1) then
16:   Tistall = Tistall + Index-Stall
17: end if
18: Tdec = Ldec + max(Tbstall,Tistall)
```

every port can operate in parallel, this metric is essential to measure the amount of parallelism exploitable between the key functional units: add, multiply, load and store units. This provides a first estimate of a best performance case (assuming all memory operands are in cache level 1) and also of the potential imbalance between the port usage. For example, depending on the number of cycles spent in each port, this information allows to detect whether the code is memory bound (P2, P3-P4) or computation bound (P0-P1-P5), and by how much. The number of cycles spent on each port gives us an accurate ranking on the potential bottlenecks of the code.

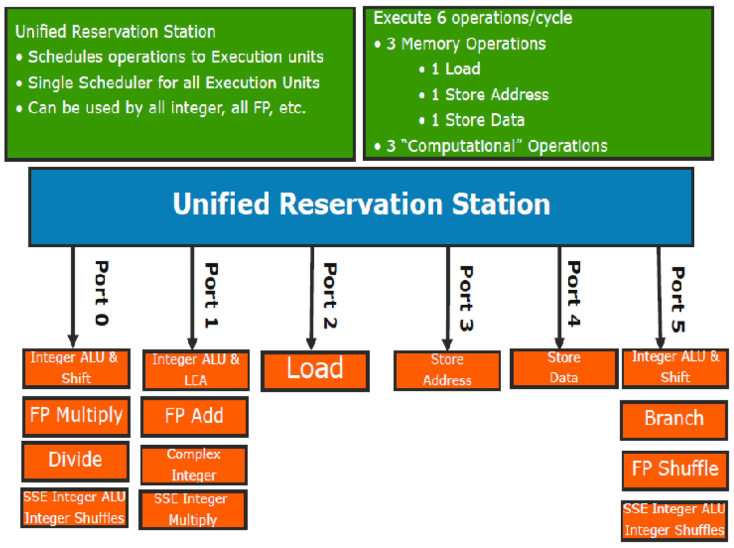

Unified Reservation Station
• Schedules operations to Execution units
• Single Scheduler for all Execution Units
• Can be used by all integer, all FP, etc.
Execute 6 operations/cycle
• 3 Memory Operations
• 1 Load
• 1 Store Address
• 1 Store Data
• 3 "Computational" Operations
Unified Reservation Station
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
Integer ALU & Shift
FP Multiply
Divide
SSE Integer ALU Integer Shuffles
Integer ALU & LEA
FP Add
Complex Integer
SSE Integer Multiply
Load
Store Address
Store Data
Integer ALU & Shift
Branch
FP Shuffle
SSE Integer ALU Integer Shuffles

Figure 3.7: Core2 execution unit overview [94].

The different execution ports P0 to P5 in the Core2 architecture correspond to (Fig. 3.7).

3.4.6 Vectorization ratios

Vectorization is an instruction level of parallelization. It processes computations not just on one scalar operand, but on a whole vector, 128 bits vector on modern architecture, which represents floating-point operands. MAQAO's static analysis provides individual load, store, add, and multiply reports on vector instruction usage: for example a vector ratio of 1 for multiply operations means that all of the multiply operations have been vectorized. This ratio is computed taking into account only floating point operations and full length packed vector operations. These metrics are essential to evaluate the quality of the vectorizing capabilities of the compiler and possibly to overcome some of its deficiencies by inserting appropriate pragmas. The vectorization ratios computed consist in: global vectorization (`GV`), load vectorization (`LV`), store vectorization (`SV`), addition/multiplication vectorization (`AV,MV`). These vectorization ratios show the quality of the code in term of vectorization. The user can immediately see if its code is well vectorized or not, without dealing with the assembly code. These ratios are computed, for a target loop, as follows:

```
GV = Number of Packed Instr  / Total Number of SSE Instructions
LV = Number of Packed Loads  / Total Number of SSE Loads
SV = Number of Packed Stores / Total Number of SSE Stores
AV = Number of Packed Add    / Total Number of SSE Additions
MV = Number of Packed Mult   / Total Number of SSE Multiplications
```

In the SSE instructions, there are two types of instructions: the `packed` and the `scalar` instructions. The `scalar` instructions that are dealing with 64 bits of the whole 128-bit vector, when the `packed` instructions are the ones dealing with the whole 128 bits vector.

The portion of code below has been analyzed with MAQAO. This code corresponds to a DAXPY [104]. The DAXPY has been compiled with Intel C compiler. One time with *-O1* flag, and a second time *-O3*.

- DAXPY compiled with the *-O1* flag is not vectorized

```
                                B1:
                                movsd (%rdi,%rax,8),%xmm1
                                mulsd %xmm0, %xmm1
  for (i=0 ; i<size ; i++)      addsd (%rsi,%rax,8), %xmm1
  y[i] += alpha*x[i]            movsd %xmm1, (%rsi,%rax,8)
                                incq %rax
                                cmpq %r8, %rax
                                jb B1
```

- MAQAO's vectorization report:

```
Global         Vectorization   =  0.00
Load           Vectorization   =  0.00
Store          Vectorization   =  0.00
Multiplication Vectorization   =  0.00
Add-Sub        Vectorization   =  0.00
```

- DAXPY compiled with the *-O3* flag is not vectorized

```
for (i=0 ; i<size ; i++)
y[i] += alpha*x[i]
```

```
B1:
movaps (%rdi,%rax,8),%xmm1
mulpd %xmm0, %xmm1
addpd (%rsi,%rax,8), %xmm1
movaps %xmm1, (%rsi,%rax,8)
addq $16, %rax
cmpq %r8, %rax
jb B1
```

- Maqao's vectorization report:

```
Global         Vectorization    =  1.00
Load           Vectorization    =  1.00
Store          Vectorization    =  1.00
Multiplication Vectorization    =  1.00
Add-Sub        Vectorization    =  1.00
```

These ratios give a first information about the quality of the code, the code is fully vectorized. This information allows the user to know that there is no need to make efforts to vectorize the code. On the other hand, if the ratios were all equal to 0, that means that the first optimization to apply to this code is vectorization to increase the `ILP` (ie. instructions per cycle).

3.4.7 Performance prediction

Performance prediction L1 Taking into account all of the limitations of the pipeline front-end and of the pipeline back-end, Maqao's static analysis provides us with an estimate of the cycles necessary to execute one loop iteration assuming all the operands are in L1. The limitations that we are taking into account are: instruction predecoding, instruction decoding, permanent register file allocation, special microcoded instructions. The front-end can be a limitation in these cases:

- Since the throughput of the predecoder is 16 bytes per cycle, the longer is the loop, the more cycles are spent in predecoding.

- The instruction decoding can be a limitation if the instructions are not well scheduled (not in a 4-1-1-1 pattern) to optimize the throughput of the decoder. If the decoder is a limitation, then the only solution is to change the scheduling of the instructions so they can fit in the 4-1-1-1 pattern. This should be done taking into account the dependencies.

- The permanent register file allocation can be a limitation because only a limited number of registers can be read. In Core2 there is a limitation of 3 registers to read from the permanent register file. This bottleneck depends on the microarchitecture. In the Sandy Bridge, there is no limit to the number of the registers to read from the permanent register file [49].

- The microcoded instructions, are instructions with more than 4 micro-operations and can not be decoded. These instructions, use the microcode ROM and takes multiple cycles to decode, in the Core2.

In most cases, the front-end bound is only useful as a lower bound.

Performance prediction L2/RAM Relying on memory access patterns detected at the assembly level and microbenchmarking results on the same memory patterns, MAQAO's static analysis computes an estimate for the execution time of a loop iteration, assuming all operands are in a given level of the memory hierarchy which means, the data reside in L1, L2 or RAM, and are accessed with stride 1. The stride of an array corresponds to the number of locations in memory between successive elements of an array. The memory patterns used for the pattern matching have previously been determined by systematic hierarchical microbenchmarking: first a simple "Load X" (resp. "Store Y") kernel, performing a single read stream through an array X, resp. a simple writing stream through an array Y, is measured under various conditions (unrolling, instruction used, etc .). Then more complex patterns "Load X Store Y", "Load X Load Y", "Load X Load Y Store Z", ... are measured to quantify the interaction between Load streams and Stores streams. Therefore this simple set of patterns is used for our performance prediction [64].

There are 9 patterns built and measured for the prediction in L2 and RAM. These patterns are small portions of code built in assembly. They are called *microbenchmarks*:

- Lx : 1 load on vector X
- Sx : 1 store on vector X
- LxLy : 2 loads on 2 different vectors (X/Y)
- LxSy : 1 load and one store on 2 different vectors (X/Y)
- LxLySz : 2 loads and 1 store on 3 different vectors (X/Y/Z)
- LyLxSy : 1 load on vector X and 1 load/store on vector Y
- LxLyLzSt : 3 loads and 1 store on 4 different vectors (X/Y/Z/T)
- LxLySzSt : 2 loads and 2 stores on 4 different vectors (X/Y/Z/T)
- LxLyLzStSu : 3 loads and 2 stores on 5 different vectors (X/Y/Z/T/U)

These microbenchmarks have been built with two kinds of assembly instructions: aligned & unaligned instructions.

- Aligned instruction : data must be aligned on a 16Bytes boundary.
- Unaligned instruction : data don't have to be aligned on a 16Bytes boundary.

The different microbenchmarks used for the prediction and other microbenchmarks are described in Appendix A.

The L2 estimate constitutes a reasonable performance objective while the RAM estimate is a stride 1 worst case. The drawback of both of these estimates is that they ignore the stride issue which in RAM will be essential, and, second, that they do not take into account the prefetching, and the mixture of hits and misses which is typical for real applications. However, it should be noted that microbenchmarking already accounts for some typical mixture of hits/misses resulting from spatial locality usage. For stride 1 memory accesses, microbenchmarking does not distinguish between primary misses which are occurring for the first word access to a cache line, and secondary misses/hits that occur when subsequent words in the cache line are requested. It provides an estimate of the average time for accessing a memory location in a stride 1 access pattern which means that the array is stored in contiguous memory. To improve the accuracy of the performance prediction, we plan to enrich the data base of microbenchmarks, so larger patterns can be recognized in the target loop, this will ensure more accurate performance predictions.

The pattern matching can also be done with multicore microbenchmarks. Indeed, multicore microbenchmarks allow to measure the effect of shared resources. For example, while *n* cores are *n* times more powerful in term of computation, the memory bandwidth is still a shared resource. Perform the pattern matching with multicore microbenchmarks gives important information on the behavior of the analyzed code.

Performance prediction for full vectorization In cases where the code is partially or not vectorized, MAQAO's static analysis computes performance estimations assuming full vectorization. This is performed by replacing the scalar operations by their vector counterparts and updating the timing estimate due to the use of these instructions. This is particularly useful to guide the optimization process and avoid useless efforts: for example, indirect access to arrays cannot be vectorized . However, in most loops, these indirect accesses are followed by floating point operations (adds or multiplies) which could be vectorized. The MAQAO performance projection gives us quickly an estimate of whether trying to vectorize these operations should pay off or not.

3.4.8 Others metrics

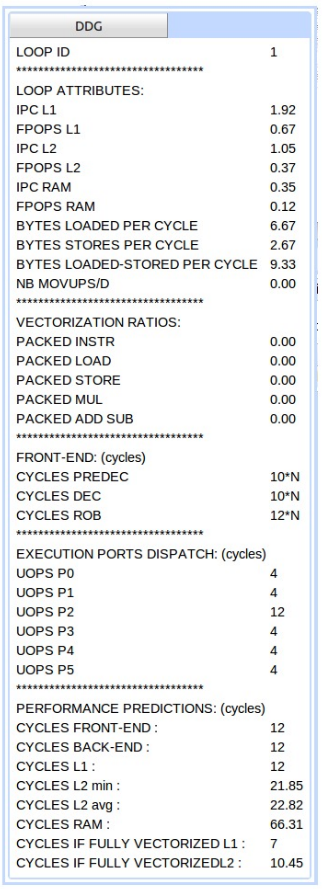
MAQAO's static analysis computes other metrics that give more information about the quality of the code analyzed. These metrics are :

```
Number of Bytes loaded per cycles:
LD Bytes/Cycle = Total number of Bytes loaded /
                 Total number of Cycles

Number of Bytes stored per cycles:
ST Bytes/Cycle = Total number of Bytes stored /
                 Total number of Cycles

Number of XMM registers used (to check spilling)

Number of instructions per cycle in L1, L2, and RAM:
IPC = Total number of Instructions /
      Total number of Cycles

Number of floating-point instructions per cycle in
L1, L2, and RAM:
FPOPS  = [(4*PACKED S) + (2*PACKED D) + SCALAR(S/D)] /
               Total Number of Cycles

with:
PACKED S = Packed Simple precision instruction
PACKED D = Packed Double precision instruction
```

```
SCALAR S = Scalar Simple precision instruction
SCALAR D = Scalar Double precision instruction
```

All the metrics described in the performance model Section 3.4, are computed by MAQAO's static analysis and displayed as shown in Figure 3.8. This figure shows how all the metrics described previously are shown in the web interface. There are first some loop attributes, such as, the number of instructions per cycle *IPC*, in L1, L2, and RAM ; the number of floating-point instructions per cycle *FPOPS* ; and the number of bytes loaded and stored per cycle. Then we have the vectorization ratios with a ratio of 1 meaning that the loop is fully vectorized. The third part corresponds to the metrics about the front-end. The fourth part is the dispatch on the execution ports, and finally, the performance prediction in cycles. The `N` corresponds to the number of iterations of the loop which is not known statically.

These metrics, as shown, give static information about the quality of the code, if the code is memory bound or compute bound, and a prediction of performance in the different cache levels.



```
LOOP ID                              1
**********************************
LOOP ATTRIBUTES:
IPC L1                               1.92
FPOPS L1                             0.67
IPC L2                               1.05
FPOPS L2                             0.37
IPC RAM                              0.35
FPOPS RAM                            0.12
BYTES LOADED PER CYCLE               6.67
BYTES STORES PER CYCLE               2.67
BYTES LOADED-STORED PER CYCLE        9.33
NB MOVUPS/D                          0.00
**********************************
VECTORIZATION RATIOS:
PACKED INSTR                         0.00
PACKED LOAD                          0.00
PACKED STORE                         0.00
PACKED MUL                           0.00
PACKED ADD SUB                       0.00
**********************************
FRONT-END: (cycles)
CYCLES PREDEC                        10*N
CYCLES DEC                           10*N
CYCLES ROB                           12*N
**********************************
EXECUTION PORTS DISPATCH: (cycles)
UOPS P0                              4
UOPS P1                              4
UOPS P2                              12
UOPS P3                              4
UOPS P4                              4
UOPS P5                              4
**********************************
PERFORMANCE PREDICTIONS: (cycles)
CYCLES FRONT-END :                   12
CYCLES BACK-END :                    12
CYCLES L1 :                          12
CYCLES L2 min :                      21.85
CYCLES L2 avg :                      22.82
CYCLES RAM :                         66.31
CYCLES IF FULLY VECTORIZED L1 :      7
CYCLES IF FULLY VECTORIZEDL2 :       10.45
```

Figure 3.8: MAQAO display of the performance model metrics computed for a target loop.

codelet__28	MAQAO Analysis (NR : svdcmp)											
	Loop ID	FP_per_cycle	BYTES_LOADED_per_cycle	BYTES_STORED_per_cycle	Ratio_Vect	Execution ports dispatch (in cycles)						XMM Reg
						P0	P1	P2	P3	P4	P5	
No Unroll	Loop1	0,5	2	2	0	2	1	1	1	1	1	2
Unroll(2)	Loop1	0,5	2	2	0	4	2	2	2	2	1	3
Unroll(3)	Loop1	1	4	4	0	3	3	3	3	3	2	4
	Loop2	0,5	2	2	0	2	2	1	1	1	1	2
Unroll(4)	Loop1	0,67	2,67	2,67	0	6	2	4	4	4	1	5
	Loop2	0,5	2	2	0	2	1	1	1	1	1	2
Unroll(5)	Loop1	1	4	4	0	5	4	5	5	5	3	6
	Loop2	0,5	2	2	0	2	1	1	1	1	1	2
Unroll(6)	Loop1	0,5	2	2	0	12	1	6	6	6	1	7
	Loop2	0,5	2	2	0	2	1	1	1	1	1	2
Unroll(7)	Loop1	1	4	4	0	7	5	7	7	7	4	8
	Loop2	0,5	2	2	0	2	1	1	1	1	1	2
Unroll(8)	Loop1	0,8	3,2	3,2	0	10	2	8	8	8	1	9
	Loop2	0,5	2	2	0	2	1	1	1	1	1	2
Default	Loop1	0,5	2	2	0	4	2	2	2	2	1	3

Table 3.1: MAQAO's static analysis on the `svdcmp` codelet from Numerical Recipes. The `svdcmp` performs a Singular Value Decomposition [85]. In this case the codelets are characterized according to two static metrics: $Ratio - vect = O$ and $FP\_per\_cycle \neq 0$

3.5 MAQAO's Static Analysis on Numerical Recipes

MAQAO's static analysis has been used on some codelets of the Numerical Recipes [85] to help the prediction of the best unroll factor. The term *codelet* is used for a piece of code having a simple regular loop at source level. MAQAO's static analysis is applied on different codelets extracted from 18 Numerical Recipes. The purpose is to use the information provided by MAQAO's static analysis to help extract rules to build an algorithm for unroll factor prediction [82].

MAQAO's static analysis has been used with the Numerical Recipes codelets to characterize the codelets depending on different metrics/information extracted from these codelets. This charaterization helped to predict the best unroll factor [82].

Table 3.1 illustrates the MAQAO's static analysis applied on one of the codelets of the Numerical Recipes. Depending on the codelet, some static metrics, from the MAQAO's static analysis, are extracted and used to understand what happens when the code is unrolled.

Tables 3.2 and 3.3 show that MAQAO-s static analysis on two other codelets from the Numerical Recipes. They show how the static metrics help to characterize the codelets in order to gather information for an unroll factor prediction.

3.6 MAQAO's Static Analysis on Real-Life Applications

To illustrate the interest of the metrics described in Section 3.4, we performed a static analysis using MAQAO on two high performance codes from the ParMA project [7]: `AIOLOS` from RECOM [87], and `ITRLSOL` from Dassault-Aviation [40, 7]. Two code fragments are shown in Figure 3.9 are the hottest loops in the hottest subroutine in both `AIOLOS` and `ITRLSOL`. The Intel C and Fortran Compilers (ifort and icc v11.0) are used to generate the binary codes analyzed by MAQAO.

3.6.1 3D Combustion simulation code

The `AIOLOS`[87] application provided by RECOM builds a 3D model of industrial-scale furnaces, and in particular, helps solve problems due to the corrosion of the walls of such a furnace at high temperatures. The most time-consuming subroutine in `AIOLOS` is `RBgauss`, which implements a red-black iterative solver. The choice of the red-black algorithm allows for easy parallelization with, for example, OpenMP.

codelet__7	MAQAO Analysis (NR : gaussj)							
	Loop ID	Execution ports dispatch (in cycles)						XMM Reg
		P0	P1	P2	P3	P4	P5	
No Unroll	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	1	1	1	1	2
	Loop3	2	1	1	1	1	1	2
Unroll(2)	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	2	2	2	1	3
	Loop3	2	1	1	1	1	1	2
Unroll(3)	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	1	1	1	1	2
	Loop3	2	1	1	1	1	1	2
Unroll(4)	Loop1	2	1	1	1	1	1	2
	Loop2	4	1	4	4	4	1	5
	Loop3	2	1	1	1	1	1	2
Unroll(5)	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	1	1	1	1	2
	Loop3	2	1	1	1	1	1	2
Unroll(6)	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	1	1	1	1	2
	Loop3	2	1	1	1	1	1	2
Unroll(7)	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	1	1	1	1	2
	Loop3	2	1	1	1	1	1	2
Unroll(8)	Loop1	2	1	1	1	1	1	2
	Loop2	8	1	8	8	8	1	9
	Loop3	2	1	1	1	1	1	2
Default	Loop1	2	1	1	1	1	1	2
	Loop2	2	1	2	2	2	1	3
	Loop3	2	1	1	1	1	1	2

Table 3.2: MAQAO's static analysis on the `gaussj` codelet from Numerical Recipes. The `gaussj` performs a Gauss-Jordan Elimination [85]. The interesting loops are the ones that have been transformed : execution ports dispatch different from the original no-unroll loop dispatch.

codelet__1	MAQAO Analysis (NR : toeplz)							
	Loop ID	Execution ports dispatch (in cycles)						XMM Reg
		P0	P1	P2	P3	P4	P5	
No Unroll	Loop1	4	4	3	0	0	3	5
Unroll(2)	Loop1	5	5	6	0	0	4	10
Unroll(3)	Loop1	6	6	9	0	0	4	15
	Loop2	4	4	3	0	0	3	5
Unroll(4)	Loop1	8	8	12	0	0	6	16
	Loop2	4	4	3	0	0	3	5
Unroll(5)	Loop1	10	10	15	0	0	4	16
	Loop2	4	4	3	0	0	3	5
Unroll(6)	Loop1	12	12	18	0	0	4	16
	Loop2	4	4	3	0	0	3	5
Unroll(7)	Loop1	14	14	21	0	0	4	16
	Loop2	4	4	3	0	0	3	5
Unroll(8)	Loop1	16	16	34	2	2	6	16
	Loop2	4	4	3	0	0	3	5
Default	Loop1	5	5	6	0	0	4	10

Table 3.3: MAQAO's static analysis on the `toeplz` codelet from Numerical Recipes. The `svdcmp` builds Toeplitz matrices [85]. In this case, the loops with a number of XMM registers above 16 which is the maximum number of XMM registers in the Core 2 architecture. That shows that there is no need to unroll more because of register spilling.

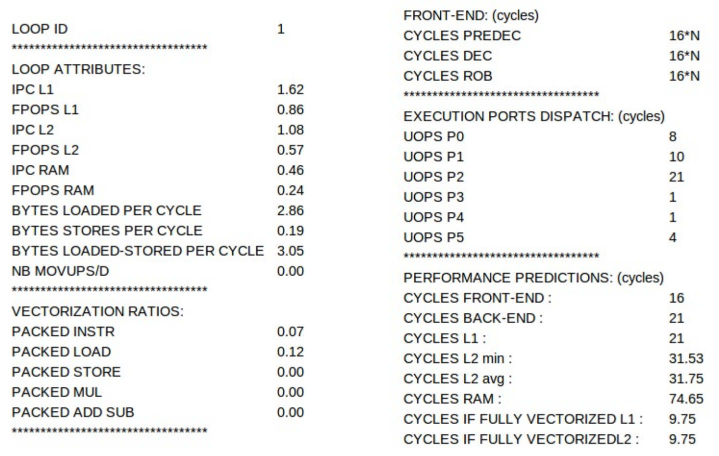
```
DO IDO=1,NREDD
  INC   =   INDINR(IDO)
  HANB  =   AM(INC,1)*PHI(INC+1)  &
  + AM(INC,2)*PHI(INC-1)  &
  + AM(INC,3)*PHI(INC+INPD)  &
  + AM(INC,4)*PHI(INC-INPD)  &
  + AM(INC,5)*PHI(INC+NIJ)  &
  + AM(INC,6)*PHI(INC-NIJ) &
  + SU(INC)
  DLTPHI = HANB/AM(INC,7)-PHI(INC)
  PHI(INC) = PHI(INC) + DLTPHI
  RESI = RESI + ABS(DLTPHI)
  RSUM = RSUM + ABS(PHI(INC))
ENDO
```

(a) AIOLOS analyzed code fragment

```
DO cb=1,ncbt
 igp = isg isg = icolb(icb+1) igt = isg - igp
 DO ig=1,igt
  e = ig + igp i = nnbar(e,1) j = nnbar(e,2)
   DO k=1,ndof
    DO l=1,ndof
     vecy(i,k) = vecy(i,k) + ompu(e,k,l)*
                             vecx(j,l)
     vecy(j,k) = vecy(j,k) + ompl(e,k,l)*
                             vecx(i,l)
    ENDO
   ENDO
 ENDO
ENDO
```

(b) ITRLSOL analyzed code fragment

Figure 3.9: Two examples of codes. The code (a) corresponds to the hottest loop of the hottest subroutine of the AIOLOS application. It implements the red black iterative solver [87]. The code (b) corresponds to the hottest loop of the hottest subroutine of the ITRLSOL application. It implements a sparse matrix-vector product [40, 7].



```
LOOP ID                                 1
**********************************
LOOP ATTRIBUTES:
IPC L1                                  1.62
FPOPS L1                                0.86
IPC L2                                  1.08
FPOPS L2                                0.57
IPC RAM                                 0.46
FPOPS RAM                               0.24
BYTES LOADED PER CYCLE                  2.86
BYTES STORES PER CYCLE                  0.19
BYTES LOADED-STORED PER CYCLE           3.05
NB MOVUPS/D                             0.00
**********************************
VECTORIZATION RATIOS:
PACKED INSTR                            0.07
PACKED LOAD                             0.12
PACKED STORE                            0.00
PACKED MUL                              0.00
PACKED ADD SUB                          0.00
**********************************
```

(a) Loop attributes & Vectorization

```
FRONT-END: (cycles)
CYCLES PREDEC                           16*N
CYCLES DEC                              16*N
CYCLES ROB                              16*N
**********************************
EXECUTION PORTS DISPATCH: (cycles)
UOPS P0                                 8
UOPS P1                                 10
UOPS P2                                 21
UOPS P3                                 1
UOPS P4                                 1
UOPS P5                                 4
**********************************
PERFORMANCE PREDICTIONS: (cycles)
CYCLES FRONT-END :                      16
CYCLES BACK-END :                       21
CYCLES L1 :                             21
CYCLES L2 min :                         31.53
CYCLES L2 avg :                         31.75
CYCLES RAM :                            74.65
CYCLES IF FULLY VECTORIZED L1 :         9.75
CYCLES IF FULLY VECTORIZEDL2 :          9.75
```

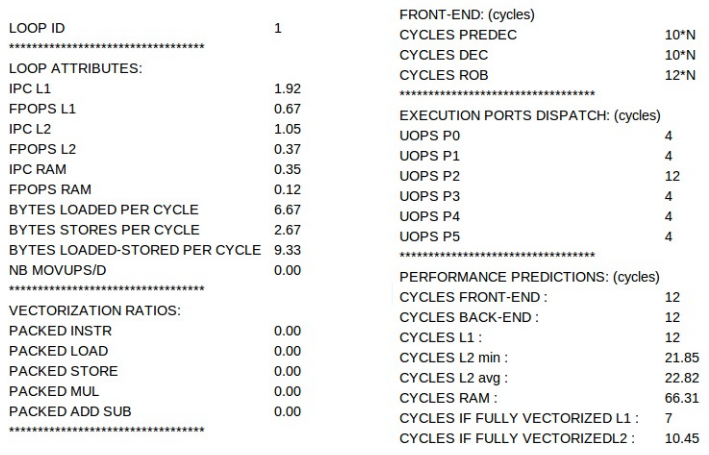
(b) Front-End & Back-End & Performance prediction

Figure 3.10: MAQAO's static analysis performed on RBgauss. Different metrics are computed: front-end, back-end, vectorization ratios, and performance predictions.

The RBgauss subroutine contains two loops (denoted *Red* and *Black* loop) with a communication between them using MPI. The two loops consist of :

- Red loop: it is an iterating loop over half of the AM array elements (*r*ed elements) to update with the other half of the AM array elements (*b*lack). It means that each *red* element depends on its four immediate *black* neighbors (Figure 3.9 (a)).
- Black loop: it has the same structure as the red loop but it updates the black elements with the red ones (computed in the Red loop).

The static analysis with MAQAO is performed on the *R*ed loop as both loops are the same. The global set of metrics computed for the *R*ed loop is shown in



```
LOOP ID                             1
*********************************
LOOP ATTRIBUTES:
IPC L1                              1.92
FPOPS L1                            0.67
IPC L2                              1.05
FPOPS L2                            0.37
IPC RAM                             0.35
FPOPS RAM                           0.12
BYTES LOADED PER CYCLE              6.67
BYTES STORES PER CYCLE              2.67
BYTES LOADED-STORED PER CYCLE       9.33
NB MOVUPS/D                         0.00
*********************************
VECTORIZATION RATIOS:
PACKED INSTR                        0.00
PACKED LOAD                         0.00
PACKED STORE                        0.00
PACKED MUL                          0.00
PACKED ADD SUB                      0.00
*********************************
```

(a) Loop attributes & Vectorization

```
FRONT-END: (cycles)
CYCLES PREDEC                       10*N
CYCLES DEC                          10*N
CYCLES ROB                          12*N
*********************************
EXECUTION PORTS DISPATCH: (cycles)
UOPS P0                             4
UOPS P1                             4
UOPS P2                             12
UOPS P3                             4
UOPS P4                             4
UOPS P5                             4
*********************************
PERFORMANCE PREDICTIONS: (cycles)
CYCLES FRONT-END :                  12
CYCLES BACK-END :                   12
CYCLES L1 :                         12
CYCLES L2 min :                     21.85
CYCLES L2 avg :                     22.82
CYCLES RAM :                        66.31
CYCLES IF FULLY VECTORIZED L1 :     7
CYCLES IF FULLY VECTORIZEDL2 :      10.45
```

(b) Front-End & Back-End & Performance prediction

Figure 3.11: MAQAO's static analysis performed on `EUFLUXm`. Different metrics are computed: front-end, back-end, vectorization ratios, and performance predictions.

Figure 3.10. Below a summary of the values:

- Vectorization report: the global vectorization ratio is equal to 6%. The compiler has not vectorized the loop.
- Execution units usage (the format is PORT_NUMBER:CYCLES_SPENT): P0:8 / P1:10 / P2:21 / P3:1 / P4:1 / P5:4. This shows that the loop is bounded by the loads.
- L1 prediction: 21 cycles. This corresponds to maximum value between: P0, P1, P2, P3, P4, and P5, and it is considered as the optimal bound.
- L2 prediction: 31.53 cycles.
- RAM prediction: 74.65 cycles.
- Vectorization prediction (assuming data in L1): 9.75 cycles.

Thanks to the static analysis of MAQAO, we can notice that the code is memory bound on Core 2, since it takes 21 cycles to execute all the read instructions. This corresponds to the largest number of cycles on any given port. It also shows that even if the code is not vectorized, it is not worth it to force the vectorization of arithmetic operations, because the code is memory bounded.

3.6.2 Iterative solver for the Navier-Stokes equation

The `ITRLSOL`[40, 7] application, developed by Dassault-aviation, solves the Navier-Stokes equation, through the use of Computational Fluid Dynamics (CFD), with the help of an iterative solver. The most time-consuming subroutine in `ITRLSOL` is `EUFLUXm` (Figure 3.9 (b)), which implements a sparse matrix-vector product. The `EUFLUXm` subroutine contains two groups of quadruply nested loops (2 identical quadruply nested loops in each group).

For the considered 4-level loop nest in this code, the global set of metrics computed is shown in Figure 3.11. Below is a summary of the values:

- Vectorization report: all the ratios of vectorization are equal to 0%. The compiler has not vectorized any of the loops.
- Execution units usage (the format is PORT_NUMBER:CYCLES_SPENT): P0:3 / P1:3 / P2:6 / P3:2 / P4:2 / P5:2
- L1 prediction: 6 cycles.
- L2 prediction: 13.05 cycles.
- RAM prediction: 37.29 cycles.
- Vectorization prediction (assuming data in L1): 3 cycles.

The static analysis with MAQAO shows that the code is dominated by memory accesses (6 cycles to execute all load instructions) (max(P0, P1, P2, P3, P4, P5) = P2, the port of loads) and not vectorized (vectorization ratio = 0). This information gives a first impression on the code quality.

3.7 Search for accuracy

MAQAO's static analysis is a tool used to gather data about the quality of the code, quickly. Indeed, it does not have any knowledge about dynamic phenomena. It analyzes the binary generated, offline, without execution. MAQAO is able to give estimations about the number of cycles spent in the different stages of the pipeline. MAQAO's static analysis is based on a x86 performance model that mimics the behavior of the processor. The first version of MAQAO was implemented for the IA64 architecture. The in-order behavior of this architecture and the instruction bundles make it easier to implement the performance model. However, for x86, it is not that easy. The introduction of out-of-order mechanism, the micro-operations, the limitation of the register file ports makes the understand and the implementation of the x86 performance model, in MAQAO, more complicated. Having an accurate performance model was one of the objectives of this thesis. This performance model, gives a more accurate room of improvement. Indeed, the detailed model goes further then just giving an estimation of GFLOPS, which gives to the programmer a false impression of how much the performance can be increased. Moreover, the pattern matching with microbenchmarks, used, gives an important information about the upper bound that can be achieved depending on the data location. An useful information about the data location is then provided which can give a first idea for optimization. Indeed, if the performance of the target code are similar to the performance of a RAM microbenchmark, then probably, the prefetching can be considered as an optimization.

A real research and engineering works have been done, to achieve this objective. The performance model implemented for the x86 architectures and precisely for the Core 2, is the first contribution of this thesis. The model implemented can easily be extended to the other new x86 architecture such as the Nehalem or the Sandy Bridge.

3.8 Limitations

MAQAO's static analysis is a tool for a binary code inspection. It is the first step in a comprehensive software quality-control regime. The main drawback of a static analysis, is that it does not consider any dynamic phenomenon. MAQAO's static analysis is able to give an estimate of a lower bound that can be achieved but this lower bound does not take into account any of the known dynamic behaviors (ie. memory access pattern, 4K-aliasing, etc.)

3.9 Conclusion

In this chapter, we address MAQAO's static analysis [66, 24], a part of MAQAO which is a tool for performance tuning that relies on both the static analysis of binaries and on the data collected through instrumentation. The static analysis is combined with the hint mechanism of MAQAO in order to help the user locate easily in the application's source code the code fragments that exhibit poor performance. Moreover, this analysis provides a rough estimate of possible performance gains that could be extracted by an efficient vectorization. The static analysis is considered as a first step in the process of quality-control. It gives a first conclusion about the quality of the code, potential bottlenecks, and first potential optimizations. In the next chapter, we address the second contribution in this thesis, which is DECAN, a Decremental Analysis tool.

Chapter 4

DECAN: Decremental Performance Analysis Tool

4.1 Introduction

The world of High Performance Computing is dynamic and constantly evolving, achieving the petaflop performance and attaining exaflop in the near future. Nonetheless, peak performance is often achieved only for a few applications and after a rigorous performance optimization process for a target architecture. Unfortunately, current optimizing compilers and run-time systems often fail to keep pace with the complexity of the available hardware and applications, thus leaving many programs underperforming while wasting expensive computing resources and energy. Therefore, users are still forced to resort to the tedious, repetitive, and time-consuming process of understanding the behavior of a large number of legacy, or even new, applications and optimizing them for constantly evolving architectures. Such approaches are expensive, and can considerably slow down the time to the market for new HPC systems.

In past decades, iterative feedback-directed compilation became a popular black-box approach to empirically improve static compilers and adapt code to any given architecture [103, 34, 26, 98]. This technique helps find the best performing variant of a program by automating the exploration of the available optimization space of this program. However, such approaches are usually slow, limited to a set of available program transformations, and rarely provide clear insight about performance problems and bottlenecks. They also offer no assistance in the case of algorithmic changes or when manual optimizations are needed. Therefore, reliable dynamic performance analysis becomes critical to quickly detect performance anomalies and bottlenecks to help users drive manual or automatic optimization processes or even suggest possible hardware improvements to improve the utilization of all available resources.

Multiple tools and techniques have been developed to perform dynamic program and system analysis that differ in scope, precision, and overhead. Most of these techniques are based on instrumentation [52, 71], statistical sampling [4, 80, 70], simulation [37] or a combination thereof [91, 10, 89, 48, 73]. Architecture and system simulation tools can provide accurate information about program execution, but most of the time they induce several orders of magnitude slowdowns and require a detailed simulator that is rarely available, particularly for rapidly evolving hardware.

Our initial experience with performance tools has been that they are overly complex, not easily portable, and often inaccurate when applied to large industrial HPC applications. We seek a simple and portable alternative solution to detect instructions associated with performance anomalies. During the program analysis and traditional feedback-directed optimization, we often run the same program with the same input multiple times while ignoring the output. Naturally, during production runs, the optimized program should be semantically equivalent to the original program and should produce the same output, but we reason that there is no strict requirement on semantics during performance analysis.

In this chapter we present DECAN, an automated tool for decremental performance analysis. The motivations of DECAN are threefold:

• Evaluate room for improvement: because any optimization effort can be long and tedious, in order to budget it wisely, it is necessary to know in advance the upper limit of the potential gains to get from this effort.

• Correlate with the source code: because dealing with memory optimizations means that the data layout may have to be changed, to speed-up the code refitting task DECAN informs which data structure has to be reconsidered.

DECAN deletes or replaces x86 SSE (Streaming SIMD Extensions) instructions inside regular loops to detect bottlenecks. This replacement, done directly at the binary level, generates a set of modified binaries that will be associated to performance. For the evaluation time of all of these versions DECAN performs a kernel execution that allows to run the hot code region within its original execution environment without running the whole application. This means that only the kernel of interest with and without modification is being run and not the entire program.

Due to the deletion/replacement of an instruction, the semantics of the original program is modified. Despite the theoretical risk of program crashes and register overflows, empirically DECAN's approach has been found to be robust enough to handle most regular, or even with indirections, HPC loops. It appears that changing the semantics of the program allows to extract more precise information about the behavior of the hot code region. This can be compared to a debug process. When a programmer has a bug, he adds or removes some expressions in his code to detect the bug. The concept of DECAN is similar, but the bug in that case is the bad performance.

The DECAN technique opens up many performance analysis and optimization opportunities. As DECAN is doing fine-grained analysis compared to existing tools, it may be considered as an additional tool that can be combined with higher level profiling tools, in order to have a precise code analysis.

This chapter is organized as follows: Section 4.2 gives a brief overview of the DECAN process. Section 4.3 describes a motivating example. Section 4.4 presents the decremental analysis concept and infrastructure. Section 4.5 is about the methodology used for performance measurements in DECAN. Section 4.6 gives the experimental evaluation of DECAN.

4.2 Overview

The concept of DECAN is similar to the debug process that every programmer follows. When a bug occurs in a program, the developer generally modifies the code by removing and/or transforming some source code lines in the program and runs it to check if the bug still occurs or not. DECAN follows a similar process. Bad performance is considered `a bug`. DECAN performs via binary patching of SSE memory access instructions for detection of delinquent instructions responsible of the poor performance. DECAN is a fine-grained bottleneck detection tool. Figure 4.1 shows the overall process of DECAN.

This figure shows that DECAN performs on the most time-consuming region of an application. This region is identified using the profiling. The profiling can be done

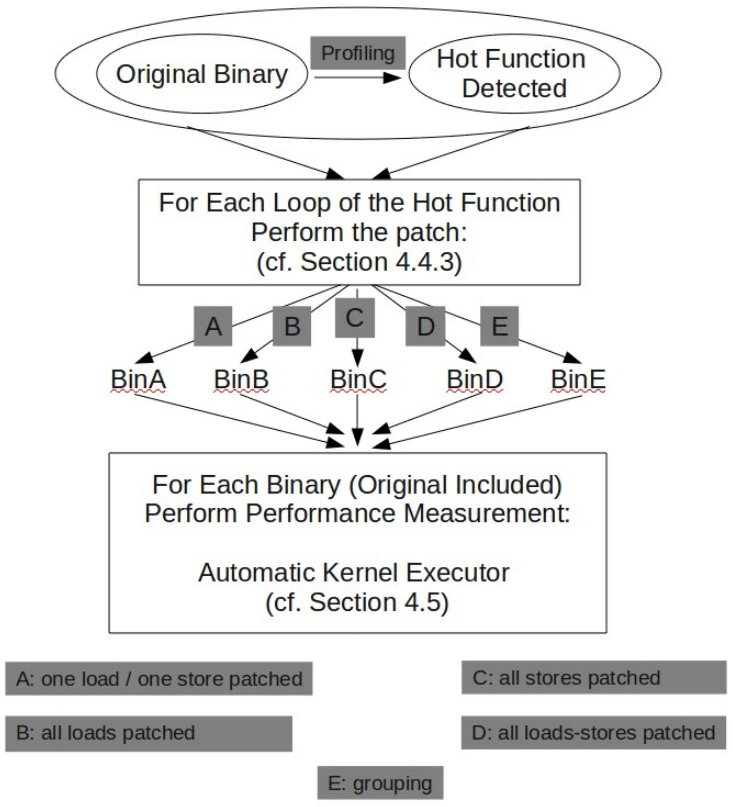

Original Binary
Profiling
Hot Function Detected
For Each Loop of the Hot Function Perform the patch: (cf. Section 4.4.3)
A
B
C
D
E
BinA
BinB
BinC
BinD
BinE
For Each Binary (Original Included) Perform Performance Measurement:
Automatic Kernel Executor (cf. Section 4.5)
A: one load / one store patched
C: all stores patched
B: all loads patched
D: all loads-stores patched
E: grouping

Figure 4.1: Decremental performance analysis infrastructure (DECAN).

automatically using transparent profiling tools such as OProfile [70] and VTune [4] or manually if the user already has knowledge of the application's behavior. When the hot function is detected, then DECAN is used to patch instructions in different combinations:

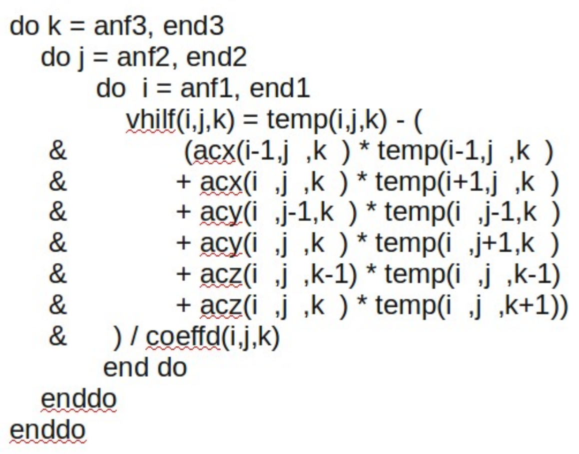
- all loads at the same time
- all stores at the same time
- all loads-stores at the same time
- one load or one store at a time
- grouping: patch a group of dependent instructions

All these combinations are detailed in Section 4.4.3. These substitutions lead to the generation of new binaries that are run, and the original binary also, using the kernel executor that runs the target region in its original execution environment without running the whole application. These executions are evaluated to determine the impact of each substitution/patch. The process of patching is described in Section 4.4 and the automatic kernel execution is presented in Section 4.5.

4.3 Motivation

4.3.1 Decoupling semantic from analysis

The general idea of DECAN is based on the observation that there is a lot in common between functional debugging and code optimization. As a matter of fact, code optimization is often about performance debugging. This is particularly true when the analyst has to deal with memory behavior, where a single instruction can lead to dramatic performance variation.



```
do k = anf3, end3
   do j = anf2, end2
        do  i = anf1, end1
          vhilf(i,j,k) = temp(i,j,k) - (
    &           (acx(i-1,j  ,k  ) * temp(i-1,j  ,k  )
    &          + acx(i  ,j  ,k  ) * temp(i+1,j  ,k  )
    &          + acy(i  ,j-1,k  ) * temp(i  ,j-1,k  )
    &          + acy(i  ,j  ,k  ) * temp(i  ,j+1,k  )
    &          + acz(i  ,j  ,k-1) * temp(i  ,j  ,k-1)
    &          + acz(i  ,j  ,k  ) * temp(i  ,j  ,k+1))
    &     ) / coeffd(i,j,k)
         end do
   enddo
enddo
```

Figure 4.2: Example of a loop that shows a 4K-aliasing problem. Code extracted from `Matvec` routine provided by MAGMA Giebereitechnologie GmbH.

Figure 4.2 shows a false dependency between loads of `acx(i-1,j,k)`, `temp(i-1,j,k)` and the store of `vhilf(i,j,k)` leads to the serialization of these memory accesses. This considerably lengthens the critical path. The purpose of DECAN is to diagnose the source of the problem, not the cause of it. Here we suspect that the false dependency is due to a 4K-aliasing (ie. `acx`, `temp`, and `vhilf` have the same addresses modulo 4K). Despite their complete independence, these three instructions are considered by the processor as targeting the same address. Indeed, on certain architectures and in 64 bits addresses, the processor considers only the twelve first bits of the address. If two addresses have the same twelve first bits then the processor considers that these addresses might be the same and serializes the operations on these addresses. If we consider the following C kernel:

```
for (i=0 ; i<SIZE ; i++)
    a(i) = b(i-offset)
```

If we have addresses such as:

```
address(a)%4KB = address(b)%4KB
(the same low 12 bits)
```

With `offset` $= 1$, there is a conflict between **the store a(i)**, at iteration **i** and **the load b((i+1)-1)** at iteration **i+1**. This is known as the 4K-aliasing problem.

To detect that these three instructions bear a performance problem, DECAN breaks the semantics of the code! Exactly as an engineer trying to understand which assembly instruction is causing the problem, DECAN replaces memory instructions by `nops`. A `nop`, is an assembly instruction that does nothing. In Intel64 architecture, it has different sizes, from 1 byte to 9 bytes and is issued in the P0, P1, or P5 execution port.

If removing a single instruction leads to an unexpected performance improvement then this instruction is considered a delinquent instruction. While breaking the semantics of the code seems dangerous, in practice for a loop body with a control flow of reasonable complexity, we find this method extremely efficient.

In the case of Matvec, it is difficult to detect the 4K-aliasing problem. Using DECAN with Matvec allows to pinpoint precisely the delinquent instructions. Figure 4.3 shows the impact on performance of the store instruction. The X axis corresponds to the number of cycles per iteration and the Y axis corresponds to the new binaries generated by DECAN when patching one memory access or all memory accesses. One of the main advantages of DECAN is to provide an immediate estimation of what can be gained by optimizing the loop body: The difference between the original binary and the one without any Load/store is 55 cycles minus 22 i.e 33 cycles. The white bar on the top corresponds to the original version. The black bar on the bottom corresponds to a binary with all load and stores nopped. Each light transparent gray bar corresponds to a version where a specific load instruction has been changed in a `nop`. The white bar with diagonal large stripes corresponds to the version without any load instructions. The bar with diagonal tight stripes corresponds to a version without any store. The transparent light gray bar corresponds to a version where one store has been transformed into a `nop`.

The white bar on the top (the long one) corresponds to the performance of the original Matvec code and the transparent light gray bar at the bottom (the small one) corresponds to the Matvec code with a `nop` patched store. Patching the store provides a speedup of 2. In this case, the store instruction is considered as the delinquent instruction.

4.3.2 Alteration and preservation

Once a routine has been selected for optimization DECAN proceeds to its isolation (cf. Section 4.5). The isolation mechanism in DECAN is based on the alteration of the execution flow and on the preservation of the run-time context. This means that, in the first stage, the application is executed but at the first call of the selected function, the application is stopped, using a `gdb` break, and its memory is dumped into a file. In the second stage, still in a fully automated way, the binary of the application is patched in order that the `main` function now branches directly to a loader that loads the dumped memory context and calls the hot function. In DECAN, the alteration of the execution flow answers the first need of reducing the execution time of the benchmark program, while the preservation of the data context allows accurate analysis. The details of the technique is given in Section 4.5.

4.4 Concept and Infrastructure

4.4.1 Decan's algorithm

The key concept of the DECAN approach is the automatic identification of sets of instructions responsible for increased computation latency. It performs this through

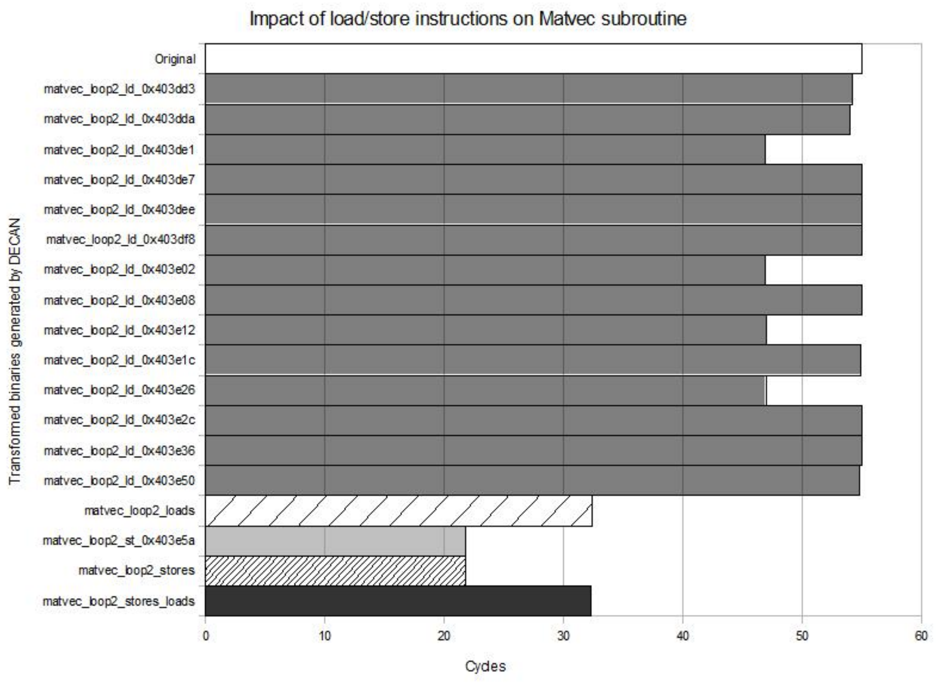


Figure 4.3: Performance in cycles per iteration when applying the `nop` transformation on Matvec subroutine.

systematic modification of individual instruction behavior in the hottest region of a program.

Algorithm 5 illustrates DECAN's algorithm. DECAN focuses on a memory access instruction patching technique, on regular loops.

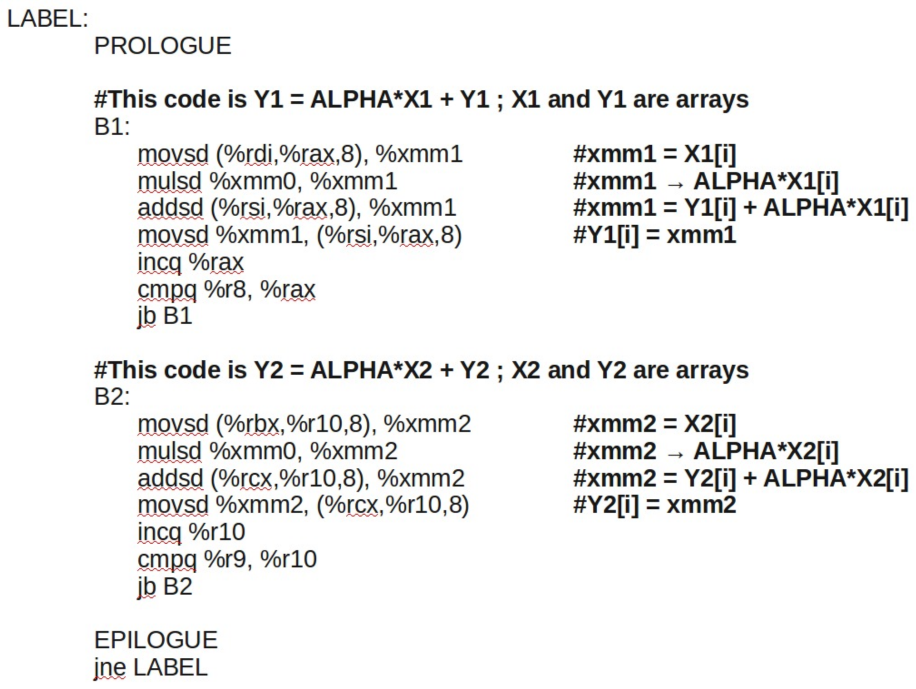
Algorithm 5 Decremental Performance Analysis Algorithm.

```
Require: Binary = The target binary to analyze
Require: Asmfile = The disassembled binary
 1: MADRAS disassembles Binary and generates Asmfile
 2: for (each Function in Asmfile) do
 3:   for (each innermost Loop in Function) do
 4:     Detect groups of dependent instructions
 5:     Perform Grouping
 6:     for (each Instruction in Loop) do
 7:       if (SSE Instruction and memory access Instruction) then
 8:         if (Instruction is a Load) then
 9:           Perform Load patching
10:         else
11:           //Instruction is a Store
12:           Perform Store patching
13:         end if
14:       end if
15:     end for
16:   end for
17: end for
```

4.4.2 Instruction detection

Decan performs on the hot routine of an application. It patches the memory access instructions of the innermost loops of this routine. Decan uses Maqao libcore [41] to detect the innermost loops in a routine and to iterate on the different instructions of each loop. When iterating on the loop's instructions, if a SSE (Streaming SIMD Extensions) memory instruction is detected, then it is selected for patching.

Figure 4.4 is portion of assembly code that corresponds to two juxtaposed loops nested in an outer loop. Each loop is a `daxpy` [104]. If we suppose that this code corresponds to the most time-consuming region of an application, then Decan detects the memory access instructions in each of the two juxtaposed loops by using the Maqao libcore.



```
LABEL:
        PROLOGUE

        #This code is Y1 = ALPHA*X1 + Y1 ; X1 and Y1 are arrays
        B1:
            movsd (%rdi,%rax,8), %xmm1          #xmm1 = X1[i]
            mulsd %xmm0, %xmm1                  #xmm1 → ALPHA*X1[i]
            addsd (%rsi,%rax,8), %xmm1          #xmm1 = Y1[i] + ALPHA*X1[i]
            movsd %xmm1, (%rsi,%rax,8)          #Y1[i] = xmm1
            incq %rax
            cmpq %r8, %rax
            jb B1

        #This code is Y2 = ALPHA*X2 + Y2 ; X2 and Y2 are arrays
        B2:
            movsd (%rbx,%r10,8), %xmm2          #xmm2 = X2[i]
            mulsd %xmm0, %xmm2                  #xmm2 → ALPHA*X2[i]
            addsd (%rcx,%r10,8), %xmm2          #xmm2 = Y2[i] + ALPHA*X2[i]
            movsd %xmm2, (%rcx,%r10,8)          #Y2[i] = xmm2
            incq %r10
            cmpq %r9, %r10
            jb B2

        EPILOGUE
        jne LABEL
```

Figure 4.4: Portion of assembly code that corresponds to two juxtaposed DAXPY [104].

4.4.3 Instruction removal

4.4.3.1 How to remove

Madras [100], a multi architecture tool and library which disassembles and patches ELF executables, is used to patch the binary code. Decan transforms each SSE (Streaming SIMD Extension) memory access instruction in the set and patches the binary, using Madras, in the following way: for any kernel containing n SSE memory instructions Decan generates n versions of the kernel. Each version corresponding to the removal of one of the memory access instruction. Decan also generates 3 additional versions, one without any loads, one without any stores, and one without any memory instructions. Then Decan generates a last set of versions based on the grouping policy: memory instructions are then removed by group considering their base address.

DECAN detects the SSE memory access instructions and substitutes them with `nop` instructions. For this substitution, three rules have to be enforced:

- First, in order to avoid artificial pressure to the targeted instruction execution port, a single `nop` instruction has to be generated (and not multiple `nop`s). By generating a list of `nop`s, the pressure is increased on the ports P0, P1 and P5, which are the ports targeted by the `nop` instruction.

- Second, in order to keep instruction alignment unchanged, the size of the `nop` has to match exactly the width of the suppressed instructions (this is possible with the multi-Byte x86 instruction).

- Third, in order to avoid side effects on the cycle count, the latency of the suppressed instruction has to be taken into account. In x86 assembly, it is possible for an arithmetic instruction to load its operand from memory. Therefore removing the memory access also removes the arithmetic instruction. In the case of a divide instruction which is 17 cycles, this has an impact on the overall loop body cycle count. In the current version of DECAN presented in this thesis, this impact is not detected. In a future version, DECAN will be able to have a more precise way of patching. Indeed, the floating-point instructions that have a memory operand, will be patched in a way that keeps the floating-point operation but deletes the memory access. For example:

```
   mulsd 4(%rdi),%xmm1
becomes
   mulsd %xmm1,%xmm1
```

4.4.3.2 Illustrating example

Consider the following simple vector addition DAXPY/SAXPY code [104] a stride of 2 written in C, and its corresponding x86 assembly code produced using the ICC, to illustrate the decremental performance analysis concept. Similar code exists in many real industrial applications, mainly in HPC applications based on the linear algebra codes. Two examples of real-life HPC applications are given in Section 4.6.

```
for (i=0 ; i<size ; i+=2)
y[i] += alpha*x[i]
```

```
B1:
movsd (%rdi,%rax,8),%xmm1
mulsd %xmm0, %xmm1
addsd (%rsi,%rax,8), %xmm1
movsd %xmm1, (%rsi,%rax,8)
addq $16, %rax
cmpq %r8, %rax
jb B1
```

Using MADRAS, DECAN detects the SSE memory access instructions and replaces them with `nop` instructions. `nop` instructions with variable length are used to match the original x86 instruction width [5]. These `nop` instructions with a size larger than 1 byte may have a register or a memory operand. The patching is performed on the DAXPY/SAXPY code, as follows:

- Only one SSE memory access instruction is patched (load or store):

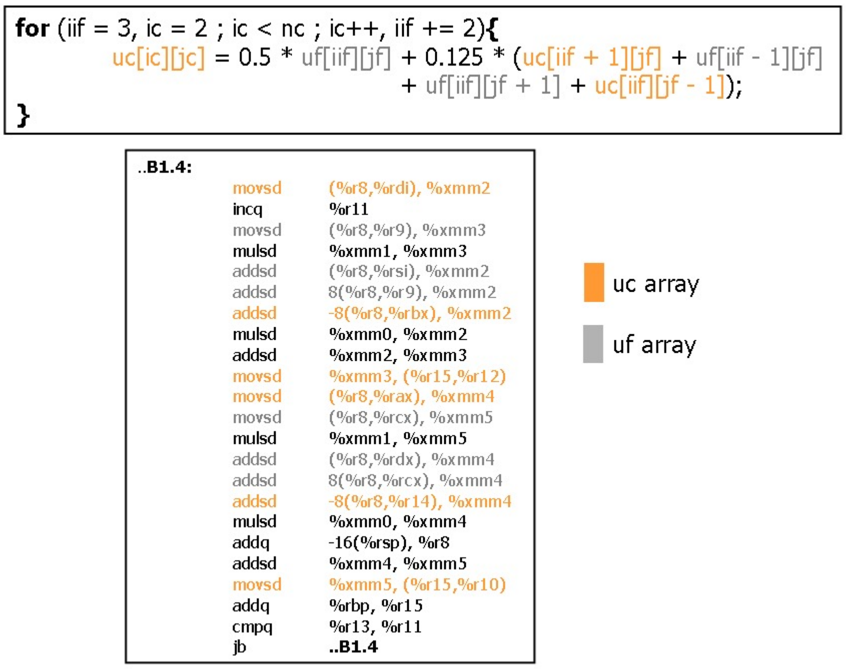
```
B1:  #One load is patched
     nop                    operand
     mulsd                  %xmm0, %xmm1
     addsd                  (%rsi,%rax,8), %xmm1
     movsd                  %xmm1, (%rsi,%rax,8)
     addq                   $16,%rax
     cmpq                   %r8, %rax
     jb                     B1

B1:  #One load is patched
     movsd                  (%rdi,%rax,8),%xmm1
     mulsd                  %xmm0, %xmm1
     nop                    operand
     movsd                  %xmm1, (%rsi,%rax,8)
     addq                   $16,%rax
     cmpq                   %r8, %rax
     jb                     B1

B1:  #One store is patched
     movsd                   (%rdi,%rax,8),%xmm1
     mulsd                   %xmm0, %xmm1
     addsd                   (%rsi,%rax,8), %xmm1
     nop                     operand
     addq                    $16, %rax
     cmpq                    %r8, %rax
     jb                      B1
```

- All the loads are patched to generate a new binary and all the stores are patched to generate another binary:

```
B1:
     nop    operand
     mulsd  %xmm0, %xmm1
     nop    operand
     movsd  %xmm1, (%rsi,%rax,8)
     addq   $16, %rax
     cmpq   %r8, %rax
     jb     B1

B1:
     movsd  (%rdi,%rax,8),%xmm1
     mulsd  %xmm0, %xmm1
     addsd  (%rsi,%rax,8), %xmm1
     nop    operand
     addq   $16, %rax
     cmpq   %r8, %rax
     jb     B1
```

- All the loads and stores instructions are patched to generate one binary:

```
B1:
     nop    operand
     mulsd  %xmm0, %xmm1
     nop    operand
     nop    operand
     addq   $16, %rax
     cmpq   %r8, %rax
     jb     B1
```



```
for (iif = 3, ic = 2 ; ic < nc ; ic++, iif += 2){
        uc[ic][jc] = 0.5 * uf[iif][jf] + 0.125 * (uc[iif + 1][jf] + uf[iif - 1][jf]
                                      + uf[iif][jf + 1] + uc[iif][jf - 1]);
}
```

```
..B1.4:
        movsd   (%r8,%rdi), %xmm2
        incq    %r11
        movsd   (%r8,%r9), %xmm3
        mulsd   %xmm1, %xmm3
        addsd   (%r8,%rsi), %xmm2
        addsd   8(%r8,%r9), %xmm2
        addsd   -8(%r8,%rbx), %xmm2
        mulsd   %xmm0, %xmm2
        addsd   %xmm2, %xmm3
        movsd   %xmm3, (%r15,%r12)
        movsd   (%r8,%rax), %xmm4
        movsd   (%r8,%rcx), %xmm5
        mulsd   %xmm1, %xmm5
        addsd   (%r8,%rdx), %xmm4
        addsd   8(%r8,%rcx), %xmm4
        addsd   -8(%r8,%r14), %xmm4
        mulsd   %xmm0, %xmm4
        addq    -16(%rsp), %r8
        addsd   %xmm4, %xmm5
        movsd   %xmm5, (%r15,%r10)
        addq    %rbp, %r15
        cmpq    %r13, %r11
        jb      ..B1.4
```

Figure 4.5: Code example to illustrate the concept of grouping. The instructions in the assembly code with the same color (gray and orange), correspond to an access to the same source array.

4.4.3.3 Grouping version of patching

DECAN generates a set of code versions based on the grouping policy. The idea of performing the group policy comes from a code that has the following behavior:

```
B(i) = A(i) + A(i+1)
```

If we assume that `A` is coming from the memory, the `A(i)` generates a miss followed by a hit with `A(i+1)`. If `A(i)` is replaced by a `nop` instruction then `A(i+1)` generates the hit, and if `A(i+1)` is *nopped* then `A(i)` generates a hit. This is why replacing both accesses is more significant.

In DECAN, are considered as a group all memory instructions that are accessing to the same array, that means that these memory instructions are using the same base address. Patching a group of memory instructions that are considered "dependent" can allow to pinpoint precisely the bottleneck. Instead of dealing with all the memory instructions, we deal with a more significant set of memory instructions. Figure 4.5 shows a code example that illustrates the grouping. The memory access instructions with the same color in the assembly code correspond to a group of dependent instructions that use the same base address. That means that they are accessing to the same array, the array has the same color in the source code. In Section 4.6, we show how grouping is used to pinpoint precisely the delinquent memory accesses.

4.4.3.4 Performance measurement

The new binaries generated by DECAN are measured using the automatic kernel executor. The performance obtained correspond to the execution time of the hottest routine. The evaluated performance are divided by the number of iterations of the loop (in the hottest routine) to display the performance per iteration. The iteration bound of the loop is determined with value profiling. The performance measurements obtained are gathered in

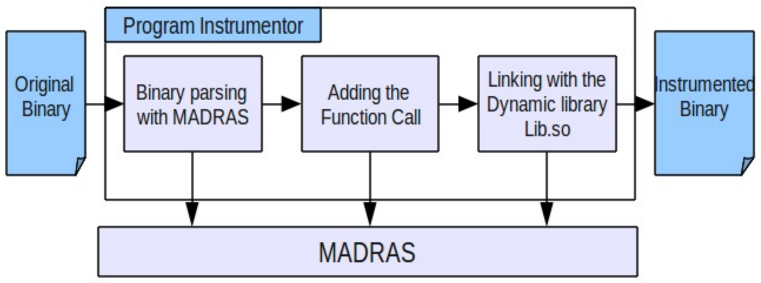


Figure 4.6: FPEC program instrumentor.

a Comma-Separated Values file that contains the name of the binary and its corresponding performance. The format of the binary name is the following:

```
 file_name = <function_name>_loop<loop_number>_<instruction_type>_
             <instruction_offset>_<source_line_number>;

loop_number        = sequential number of the loop in the function;
instruction_type   = loads (all)| stores (all)|
                     ld (individual)|st (individual)|
                     stores_loads (all);
instruction_offset = offset of the instruction in the binary;
source_line_number = number of the source line that corresponds to
                     the instruction;
```

For example, in `rbgauss_load3_ld_0x402f4c_line80`: the load instruction at 0x402f4c address in loop 3 of rbgauss function has been patched.

4.4.3.5 Error handling

Obviously any version leading to a crash is considered by DECAN as an error, and the version is removed from the set of programs to analyze. Any version with an execution time larger than the original code is considered as an error (removing instruction should not increase the execution time), and the version is also removed from the analysis. Also, another drawback that can occur is the generation of *Floating Point Exceptions(FPE)* that are not present in the original binary. Hence, the appearance of new FPE can affect the execution time. In order to tackle this problem, a small tool to detect and count the FPEs is needed. This helps us to count FPEs in the original binary and compare it with the number of FPEs in the generated binaries. The tool allows to detect and count FP exceptions at binary level, which means that the targeted code doesn't have to be recompiled. It acts by instrumenting the binary. For this purpose, it uses MADRAS. The tool is called *FPEC* for *Floating-point Exception Counter.* It is composed of two components:

- A program instrumentor: Instruments the original binary by adding a function call at the beginning of the main function (prior to any instruction of the program) and linking with an external dynamic library as shown in Figure 4.6.
- A dynamic library: Figure 4.7 illustrates the role of the dynamic library. When called, the added function does the following actions:
  - Parse the instrumented binary with MADRAS and extract of all instructions' addresses with their respective sizes. These are stored in a static table, in order to be used later.
  - Activate the desired interrupts as initially it depends on compilation directives.

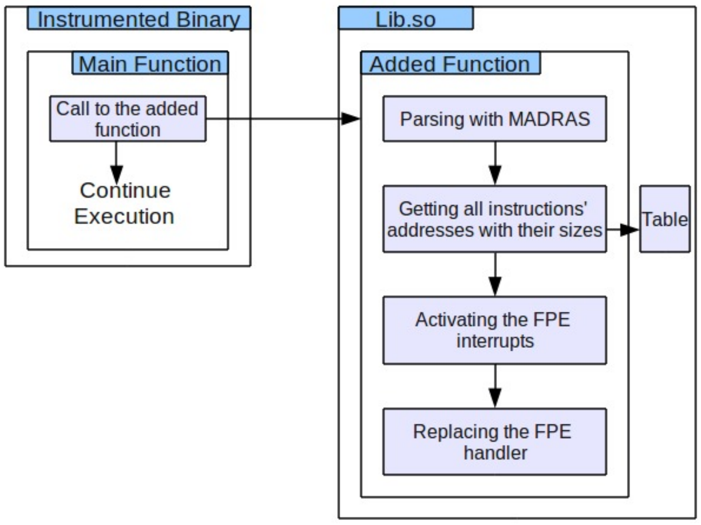

Instrumented Binary
Main Function
Call to the added function
Continue Execution
Lib.so
Added Function
Parsing with MADRAS
Getting all instructions' addresses with their sizes
Table
Activating the FPE interrupts
Replacing the FPE handler

Figure 4.7: FPEC dynamic library.

– Replace the default handler with the modified one.

When an FPE is encountered during the execution of the program the new handler is called and two structures are passed to it : a *siginfo_t* structure which contains information about the exception (address, type, etc.), and a *ucontext_t* structure which contains the context of the original program (general registers, eflags and RIP) and which is used to restore the original context. The problem with exceptions is that if we resume execution, the instruction that causes the FPE is executed again and the exception will be raised again. We don't have this kind of problem in the case of Traps for example, if we resume from a Trap the execution continues with the instruction that follows the one that causes it. Hence, to avoid iterating indefinitely on the exception we use the table of instructions' addresses extracted with MADRAS to get the size of the current instruction (the address to search is extracted from the RIP field in the *ucontext_t* structure). Once the correct size obtained, it is added to the RIP of the *ucontext_t* structure to jump to the next instruction when the context of the original program is restored.

The supported exceptions include :

1. Division by zero.
2. Overflow.
3. Underflow.
4. Invalid result.
5. Inexact result.

The number of each type of exception is kept updated each time it is captured in order to be reported at the end of execution.

4.5 Automatic Kernel Executor

Magee *et al.* [72] systematically evaluates in SPEC2006 the discrepancy between the execution time of each benchmark and the individual execution time of its hottest routine. In the particular case of 433.milc, executing the whole benchmark program takes 175,000 times more time than executing a single occurrence of its critical routine (mult_su3_mat_vec).Therefore, if the performance problem can be understood from this single occurrence a huge amount of time can be saved. In other words, considering that

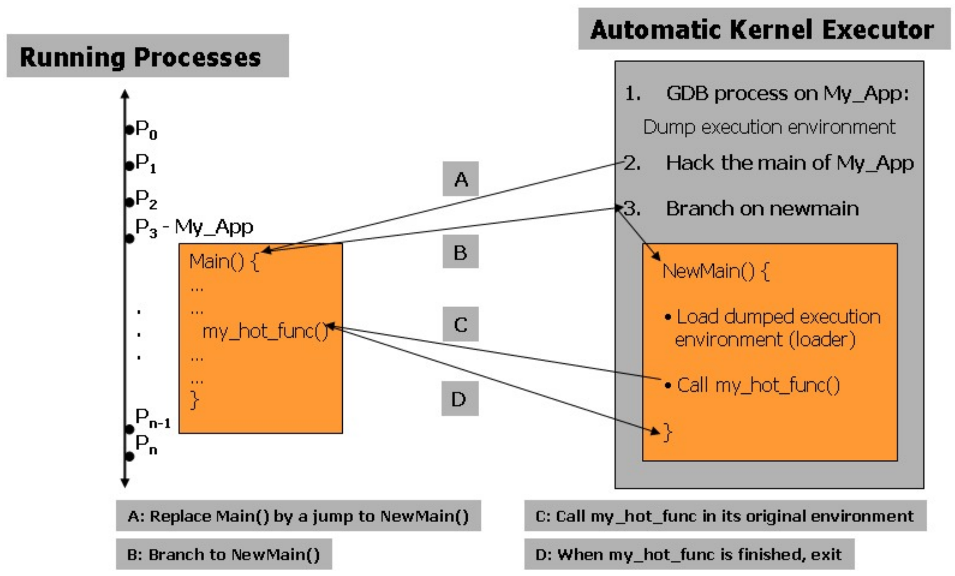

Running Processes
Automatic Kernel Executor
P0
P1
P2
P3 - My_App
Pn-1
Pn
Main() {
...
my_hot_func()
}
1. GDB process on My_App:
Dump execution environment
2. Hack the main of My_App
3. Branch on newmain
NewMain() {
• Load dumped execution environment (loader)
• Call my_hot_func()
}
A
B
C
D
A: Replace Main() by a jump to NewMain()
B: Branch to NewMain()
C: Call my_hot_func in its original environment
D: When my_hot_func is finished, exit

Figure 4.8: The Automatic Kernel Execution process. It describes the different steps of the process from memory context dumping to the branch to the loader. The purpose is to run the target function in its original environment without running the whole application.

code optimization is an iterative process, the overhead of dealing with the whole application instead of focusing on the hot function dramatically reduces the ability to explore the optimization space.

The automatic kernel executor allows the critical routine in an application to be evaluated without evaluating the whole original application. It automatically patches the binary of the application so that the `main` function branches to a loader which executes the routine in its original execution environment. The automatic kernel execution consists in:

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

The two first steps are performed by a GDB script (GNU debugger) [51] then the information provided by GDB is used to build the loader and patch the application's binary.

Figure 4.8 illustrates the process of the automatic kernel execution that consists in running a target function in its original environment without running the whole application. The dump of the memory context and the build of the loader are processes done before running the original application. When the loader is built, then the original application is executed, and during the execution, the loader patches the binary of the application to make it jump to a new main function. This new main function loads the dumped memory context and call the critical routine. The execution ends when the application returns from the critical routine. The end of the application is not run.

4.5.1 The GDB process

The GNU debugger is used to extract useful information about the memory context and the parameters of the routine. First, we set a breakpoint at the entrance of the performance critical routine. To dump the memory context the `dump binary memory` command is used. It dumps the memory allocated before starting the routine. According to the calling convention, a set of registers is dedicated to pass the arguments to the function, this set is dumped alongside the memory context. Additionally, and still according to the calling convention, the stack can be used to pass extra-parameters. Consequently, a large fraction of the stack is also dumped. At the end of the GDB process, the whole execution environment of the routine is collected. After that, the system maps this memory context and passes the parameters in order to call the routine. This GDB process is fully scripted using `expect` a Unix automation tool for interactive application [68]. The overall process of automatic kernel extraction is based on shell and `expect` scripts. This GDB script can handle both C and Fortran codes and there is no need to have the debug information in the binary file.

4.5.2 Building the loader

The loader maps the memory context dumped by the GDB process, using the C library function `mmap`, and passes the parameters to the hot function in the same way they were passed in the original application (register and stack parameters passing). Original register parameters are remapped to the registers, and original stack parameters are set back on the stack. The loader corresponds to a driver that sets the memory context of the routine, sets the parameters of the routine and calls it. The last step makes the binary of the application jump to the loader and runs the critical routine.

4.5.3 Branch to the loader

In this step the purpose is to patch the binary code of the application in order to bypass the `main` function and, instead, to directly call our loader. This, in turn, directly calls our target function. This patching is done by replacing the first bytes of the `main` by a `jump` to the loader.

4.5.4 Conclusion

Considering that code optimization is an iterative process, the overhead of dealing with the whole application instead of focusing on the hot function dramatically reduces the ability to explore the optimization space. The Automatic Kernel Executor allows to reduce this overhead by only running the target function.

One function can be called more than once (ie. several occurrences) in different places in an application. One should mention, that in the automatic kernel execution process, we are only handling the first occurrence of the function (ie. the first call). The more-than-once-called problem is detected but not implemented in our process.

4.6 Decan and Real-Life Applications

In this section, we demonstrate Decan analysis for two real-world industrial applications from independent software vendors (ISVs): Recom Services [87] and Dassault-Aviation [40, 7].

4.6.1 3D Combustion simulation code

The `AIOLOS`[87] application provided by RECOM builds a 3D model of industrial-scale furnaces, and in particular, helps solve problems due to the corrosion of the walls of such a furnace at high temperatures. The most time-consuming subroutine in the Recom application is RBgauss, which implements a red-black iterative solver. The RBgauss subroutine contains two loops, denoted the *Red* and the *Black* loop, with communications between them using MPI. The two loops include:

- A Red loop that updates half of the AM array elements using the other half. It means that each *red* element depends on its four immediate *black* neighbors (Figure 4.9).
- A Black loop has the same structure as the red loop but updates the black elements with the red ones.

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

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

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

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

Figure 4.9: The most time-consuming loop in RBgauss. There are 3 different arrays: AM(2D), and SU(1D) are read-only, PHI(1D) is read and written.

Figure 4.10 shows cycles consumed by the transformed RBgauss for each associated instruction when using nop instruction by modifying individual or all of the memory accesses. There are important bars which define the performance range: the white bar which corresponds to the original binary and the black bar which corresponds to the transformed binary without any load and store instructions. We can see that removing one memory access slightly improves the performance (transparent light gray bars) and removing all of the memory accesses (loads) improves the performance much more (large stripped bar), achieving the lower-bound of 21 cycles. This lower-bound was estimated statically based on instruction dispatch, execution port and other micro-architectural features of the processor. Such static performance estimation is computed by the MAQAO tool [41] for Woodcrest, and Tigerton processors (cf. Chapter 3). Obviously, this performance improvement is due to less cache misses caused by the loads.

From the performance achieved when removing all the memory accesses, we conclude that there is a set of memory loads instructions that have a significant impact on performance. All these instructions access the same array AM, the same base address, so we transformed all the memory accesses in RBgauss that are accessing AM array at the same time using nops. These results, presented in Figure 4.11, confirm that access to AM array is the bottleneck in RBgauss.

4.6.2 Iterative solver for the Navier-Stokes equation

The ITRLSOL[40, 7] application, developed by Dassault-aviation, solves the Navier-Stokes equation, through the use of Computational Fluid Dynamics (CFD), with the help of an

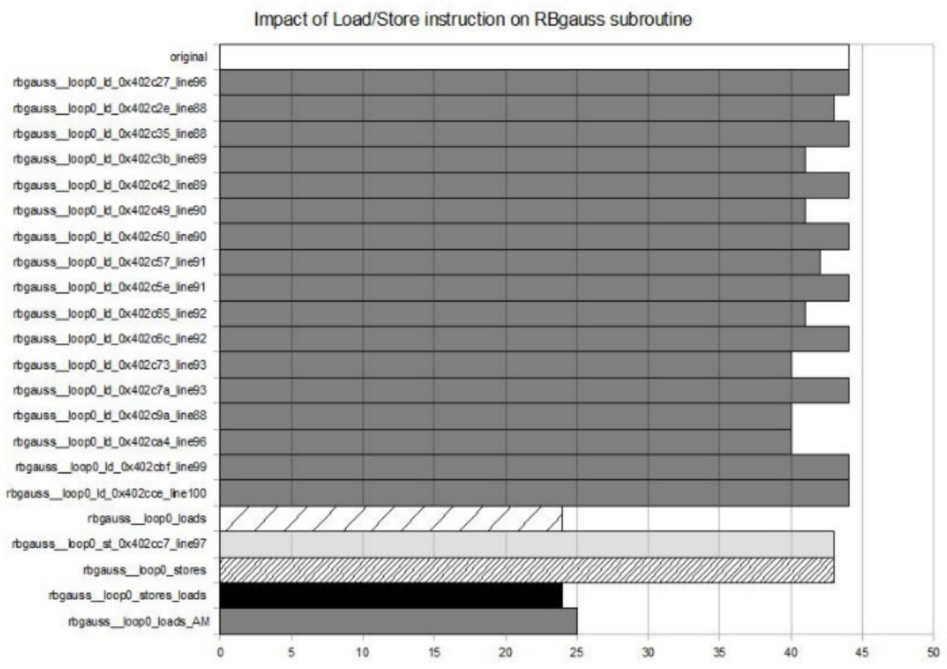


Figure 4.10: Performance in cycles per iteration when applying the `nop` on `RBgauss` subroutine. The X axis corresponds to the number of cycles per iteration and the Y axis corresponds to the different version of binaries generated by DECAN when patching one memory access (load or store) or all the memory accesses (all the loads, all stores and all the loads/stores).

iterative solver. The most time-consuming subroutine in `ITRSOL` is `EUFLUXm`, which implements a sparse matrix-vector product. The `EUFLUXm` subroutine contains two groups of quadruply nested loops, Figure 4.12 shows 2 identical quadruply nested loops in each group.

Figure 4.13 shows cycles consumed by the transformed `EUFLUXm` for each associated instruction when using `nop` instructions to modify individual or all of the memory accesses. In Figure 4.13, the white bar represents the original binary and the black bar represents the version without any load or store instructions. These define the performance range. These two experiments define the maximal performance gain which can be obtained by tuning this specific loop. We can see that removing one memory access slightly improves the performance as shown by the transparent light gray bars, and removing all of the memory loads improves the performance much more as illustrated by the large stripped bar, achieving a lower-bound of 6 cycles, this is confirmed with MAQAO. Again, the performance improvement is due to less cache misses caused by the loads.

These results suggest that there is a set of memory accesses that have a significant effect on the performance. We transformed all loads that access to the same arrays (`ompl` and `ompu`), leading to access the same base address, this means one base address for `ompl` and another for `ompu`. The results presented in Figure 4.14 show that when modifying this set of loads the lower-bound is achieved, thus indicating that the access to `ompl` and `ompu` is the bottleneck in `EUFLUXm`.

4.7 Limitations

DECAN is a new approach in performance analysis. It is considered as a fine-grained analysis tool that pinpoints the delinquent instruction or group of instructions responsible of the bad performance. DECAN, as any tool, has some limitations:

- Dealing with side effects when the instructions are replaced with `nop` instruction. The

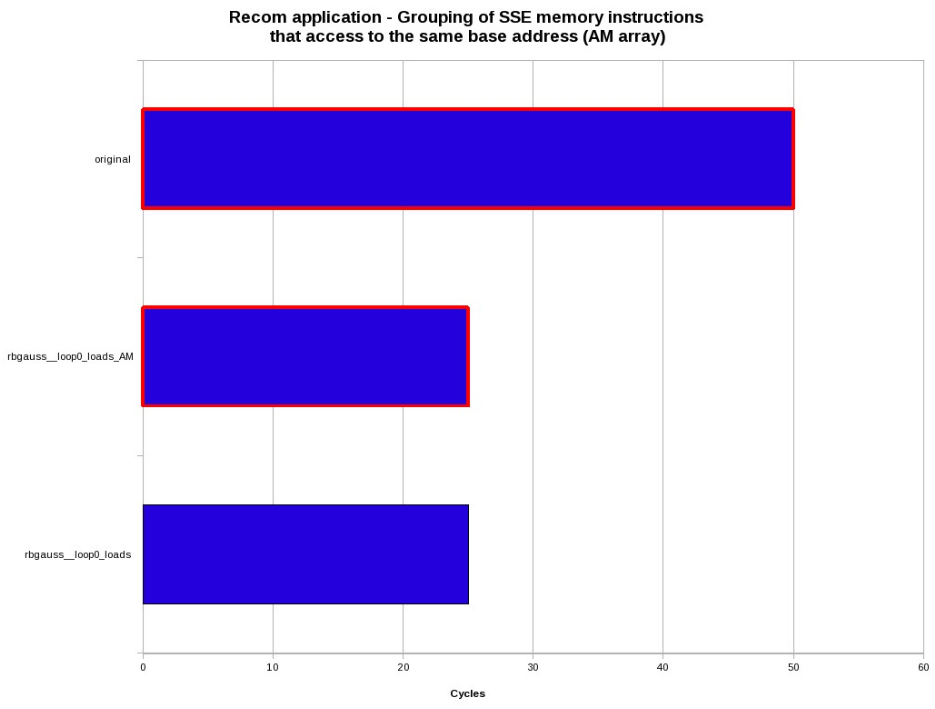


Figure 4.11: Performance in cycles per iteration when applying the `nop` transformation on a set of instructions accessing to the same base address, on `RBgauss` subroutine. The X axis corresponds to the number of cycles and the Y axis to the original binary, grouping version and all the loads version of binaries.

```
do cb=1,ncbt
   igp = isg
   isg = icolb(icb+1)
   igt = isg + igp
   do ig=1,igt
      e = ig + igp
      i = nnbar(e,1)
      j = nnbar(e,2)
      do k=1,ndof
         do l=1,ndof
            vecy(i,k) = vecy(i,k) + ompu(e,k,l)*vecx(j,l)
            vecy(j,k) = vecy(j,k) + ompl(e,k,l)*vecx(i,l)
         enddo
      enddo
   enddo
enddo
```

Figure 4.12: The most time-consuming quadruply nested loop in `EUFLUXm`. There are 4 different arrays: vecx(2D), ompu(3D), and ompl(3D) are read-only, vecy (2D) is read and written.

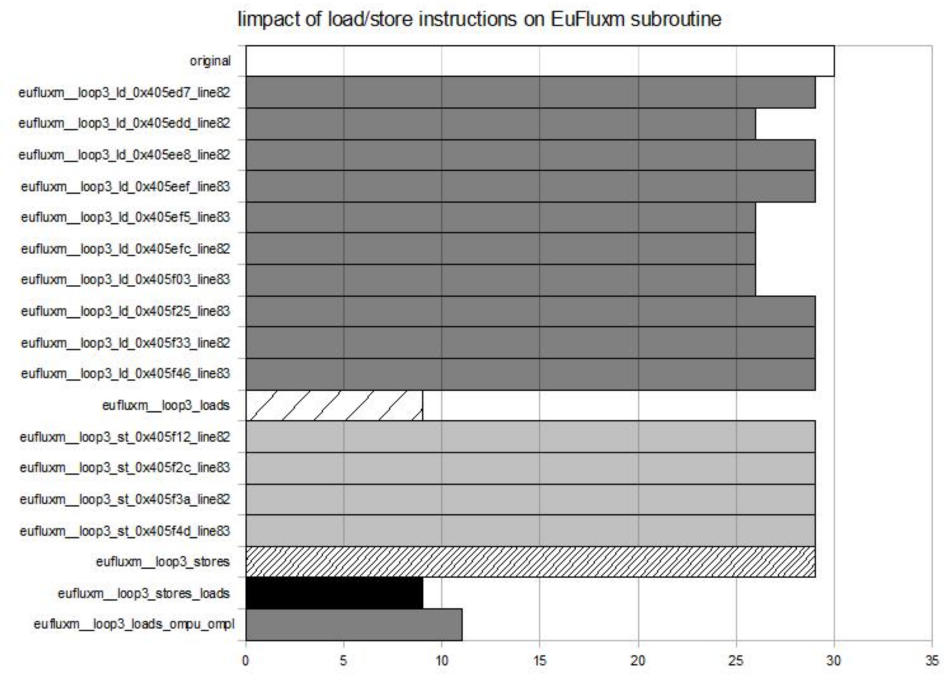


Figure 4.13: Performance in cycles per iteration when applying the `nop` transformation on `EUFLUXm` subroutine. The X axis corresponds to the number of cycles per iteration and the Y axis corresponds to the different versions binaries generated by Decan when patching one memory access (load or store) or all the memory accesses (all the loads, all stores and all the loads/stores).

patching must be done in a more clean and precise way. For example, replace the SSE memory access instructions with a `xor` instruction on the same register. This deletes the memory access and keeps a valid value, `zero`, in the register.

- Semantics is lost: This is considered as a limitation in some cases when Decan generate invalid binaries that crashes. However, from a performance point of view, this is not considered as a limitation.
- Irregular loops: Decan does not handle a loop with a complex control flow in it. Patching the instruction responsible of the branch may give aberrant behavior.
- Decan is a fine-grained tool that performs on the loop level. It needs to be coupled with a profiling tool to detect the critical routine to analyze.

4.8 Conclusion

In this chapter we introduce Decan, a tool for automatic decremental performance analysis to simplify the optimization process and trim down its time consumption. Decan pinpoints precisely the instruction responsible for a performance anomaly without any prior knowledge of the architecture by running the transformed code on a given platform. It performs on small, hot parts of the code to isolate the important instructions for the developer. Decan substitutes x86 SSE memory access instructions with `nop`s to detect memory access bottlenecks. Obviously, as for any other destructive methods, in some cases the side effects of breaking semantics may hinder the optimization.

To conclude, the Decan approach is based on the fact that alleviating the semantics constraints of a kernel allows a deeper insight of its performance behavior. While this method is routinely applied to debug assembly code we are not aware of any tool that systematizes it. Using Decan analysis and further manual optimizations, we achieved

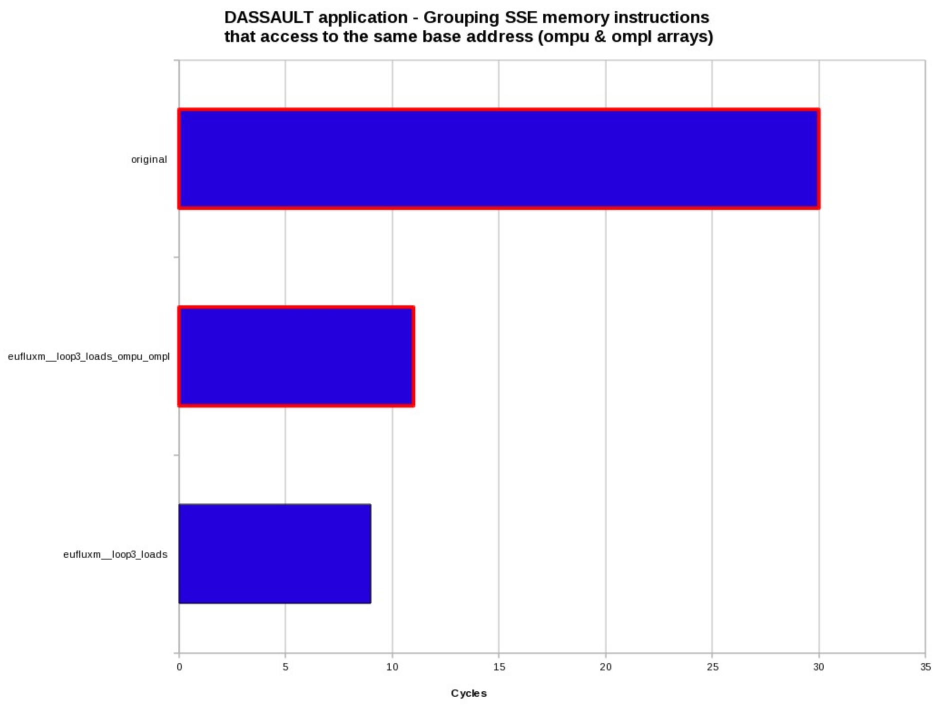


Figure 4.14: Performance in cycles per iteration when applying the `nop` transformation on a set of instructions accessing to the same base address, on `EUFLUXm` subroutine. The X axis corresponds to the number of cycles and the Y axis to the original binary, grouping version and all loads version of binaries. DECAN allows in one glance to determine the large room for improvement possible at the load level.

speedups of up to 2.5 times on industrial applications. These results are described in Chapter 5.

Chapter 5

Approach to Application Performance Tuning

5.1 Introduction

In high performance computing, there is a constant need for more resources such as CPU, memory, or I/O. With finite limits on these resources, it is the responsibility of the programmer, interacting with the compiler, to optimize an application for peak performance. Optimization consists of gathering data about a program's behavior, diagnosing the problem by identifying the resources that are saturated and the costly instructions, and prescribing a solution which entails applying a change to the code's algorithm, structure, or data layout.

The first step, which is data collection, involves an array of different analysis techniques to examine different aspects of application performance. Typically, a code is deemed optimal if it approaches the peak numerical throughput of the processor; this implies that only an algorithmic change could further improve performance. Most often, memory system effects such as the working set, stride, bandwidth, and alignment are the key factors. Once those are solved, it is important to identify the best instruction scheduling and unrolling factors to keep the pipeline full and balanced. After a particular resource has been identified as a bottleneck, it is necessary to characterize the specific cause of the problem by pinpointing the specific instructions involved and how they are suboptimal. Using this information, it is finally the responsibility of the programmer to enact a solution.

In practice, this process is extremely tedious because of the difficulty to understand a program's behavior and the complexity of modern processors. As a result, many tools and methodologies exist to analyze a code and guide the optimization. However, the provided results are essentially raw data, requiring an experienced programmer to perform analysis and synthesis of the information. Furthermore, many existing methodologies do not cover the entire process of analysis and perform a subset of the whole analysis. For example, the analysis can only be static.

Hardware performance monitoring (HPM) can provide great insight into a program's performance bottlenecks. However, there are hundreds of performance counters, but only two at a time can be counted on a Core 2. Arcane event descriptions make most performance counters useful only for true micro-architecture experts. Given these problems, we have invested significant time into identifying which performance counters are understandable and correlate well with performance. Using the HPM data, we can identify potential bottlenecks and move toward pinpointing a region of code that has potential for optimization.

In this chapter, we address optimization of HPC code, specifically CPU- and memory-bound applications. We describe a semi-automated methodology to analyze performance and guide the optimization process. Both static analysis (with Maqao and visual inspection) and dynamic analysis (of memory access) of the code are performed. Information from this analysis guides us to the regions of code furthest from the peak performance. Using this information, we introduce a new approach to identify the specific set of instructions that are responsible for a potential increased computation latency: decremental analysis, Decan. Decan involves systematically changing instructions' behavior in a particular region to identify the runtime contribution of each instruction or set of instructions. Since it uses static, dynamic, and decremental types of analysis, our semi-automated methodology is called "balanced".

This methodology has been applied on two industrial HPC codes: `RBgauss` from RE-

COM Services and ITRLSOL from Dassault-Aviation. Our modifications to the latter code achieved an improvement in sequential and parallel performance by a factor of 2.

Section 5.2 presents the tools and techniques in our analysis approach. Section 5.3 deals with two case studies of HPC codes on which the methodology was applied with significant performance improvement.

5.2 Toward a Better Evaluation Process

The process of performance analysis is a key factor for a good optimization. Having a good methodology of performance analysis allows to focus the effort in each step of the process of the performance evaluation and not to lose time in thinking about what to do next. Spending time to develop a systematic process for performance optimization has a good return on investment. Indeed, if it takes 5 weeks to optimize a HPC application and to try at the same time to extract a systematic process to achieve a good optimization, for the next application to analyze, the period of investigation can take 2 weeks.

A good methodology of performance evaluation is not just a *beautiful* theoretical concept. It gives a solid and systematic way of work to achieve the bottleneck detection in the minimum time.

The performance optimization is not always about decreasing the runtime execution. Sometimes the target can be different. As the energy consumption becomes a very important feature to target, the performance optimization in that case is to find the best optimization to reduce the energy consumption. This shows that performance optimization is a kind of equations that involve several criteria such as the execution time and the power consumption.

When an application should be optimized for a better execution time, this involves the execution platform for which the application has to be optimized. In our MAQAO static analysis, a Core 2 performance model has been implemented to evaluate the performance of an application on a Core 2 platform. However, the plugin framework of MAQAO makes it easy to add new plugin/performance models of the new architectures.

The following performance analysis techniques are intended to identify the root cause of a performance bottleneck. It is assumed that the targets of optimization are significant contributors to program execution time as reported by tools such as gprof [53] or Intel PTU [15].

This section provides a high-level description of each step of the methodology. Figure 5.1 shows a diagram of the flow through the process. This diagram shows the process we followed to evaluate the performance of two real-life programs.

First, a profiling is performed on the application to detect the hot routine. The 90/10 rule that says that 90% of the execution time is spent in 10% of the code, a portion of code that will have an impact on performance if it is optimized. When the hot routine is detected, a MAQAO's static analysis is performed to give a first view about that code quality. It gives information of the vectorization, the dispatch on the different execution ports of the architecture and some estimation of the execution time in the different level of caches of the Core 2 architecture. Then, we go further, by applying DECAN and the HPM simultaneously, to try to understand where precisely is the problem in the code, and which memory access is responsible of the bad performance. If a bottleneck is not detected than we can apply again the evaluation process on the new binaries generated by DECAN, to extract further information. At each iteration of the process, we go deeper in the understanding of the code behavior.

5.2.1 Static analysis using MAQAO

Static analysis is a method of program debugging done by examining the code without executing the program. It provides a comprehension of the structure of the code. The static analysis gives a view of the quality of the code and is considered as a first step in a complete software quality-control process.

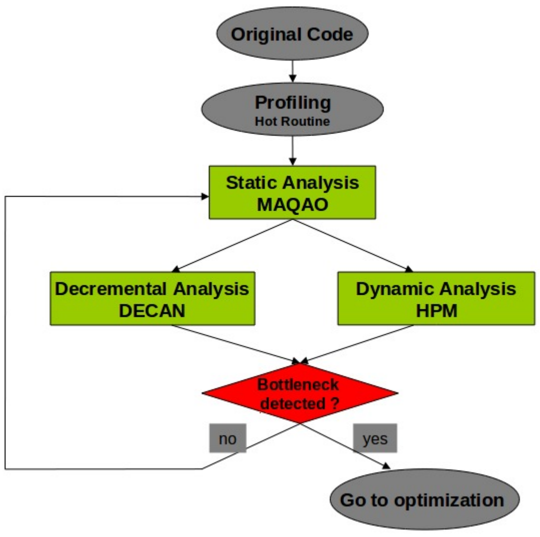


Figure 5.1: Evaluation process diagram.

Automated tools can assist programmers in performing static analysis. The sophistication of the analysis varies from a tool to another. Some tools perform on the source code, others on the compiled object code. On another front, some of them consider individual statements and others deal with the whole code.

In the performance analysis technique presented in this chapter, Maqao [41] performs a static analysis (cf. chapter 3). It deals with object code and generates different metrics that quantify the code-quality.

5.2.2 Hardware counters and memory traces

5.2.2.1 Hardware counters

Tools such as Intel's PTU[15], PerfMon[44], PAPI[80], and others make gathering HPM information relatively easy. However, even though hundreds of events can be monitored through hardware counters, most of the counters give information that is either too arcane or too esoteric to be useful [78]. Moreover, only a few events can be monitored at once on most processors. For example, on the Intel Core 2 processors, only 2 different configurable events can be monitored at a given time in most cases and up to 4 events in very particular cases, such as the monitoring of different variations of a same event. Hence, the first issue to be solved is to identify a limited set of performance counters which should have the following characteristics:

- A small number of counters to avoid numerous reruns which are costly in time. This suppose the program's behavior is repeatable.
- Detect the source of the problem AND the performance impact: for example the `LOAD_BLOCK.OVERLAP_STORE` hardware counter that counts the 4K-aliasing problem, does not detect the cost of the problem it just detects it. Figure 5.2 illustrates the performance cost detection. It shows the 4K-aliasing problem on the `a(i)=b(i-offset)` body loop when varying the **offset**. We can see a sensible impact up to **offset** $= 10$ for the HPM counter and up to **offset** $= 4$ for the runtime cost. The hardware counter detected the problem but not the cost and what we care about is the performance impact.
- Easy to understand and to correlate with performance. On top of the documentation problem, performance counters often refer to low level details of the architecture which are hard to interpret correctly. Understanding the true meaning of many performance counters involves an intimate knowledge of the microarchitecture. Through

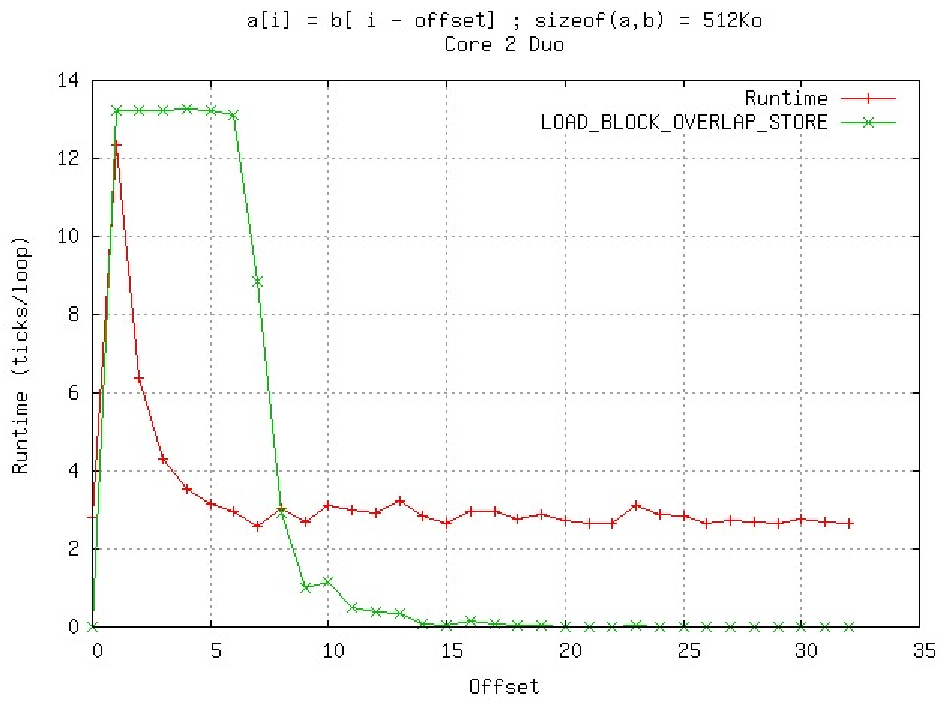


Figure 5.2: 4K aliasing problem and Hardware Performance Counter on Intel Core 2.

a painstaking exploration process, we have identified a set of counters that we find to be understandable and correlate well with the performance of our target applications. The following hardware counters have been identified to be key indicators of performance for our applications.

The INTEL documentation gives the following definitions for the selected counters:

- **L1D_REPL:** Counts the number of lines brought into the L1 data cache.
- **L2_LINES_IN.SELF.ANY/DEMAND/PREFETCH:** Counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can also count demand requests and L2 hardware prefetch requests together (ANY) or separately (DEMAND/PREFETCH).
- **CPU_CLK_UNHALTED.TOTAL_CYCLES:** Counts the total number of core cycles, while it is running code and while it is halted, as long as it not in a sleep state.

Various simple assembly kernels have been evaluated using various hardware performance counters. The purpose is to select hardware performance counters for understanding and optimizing memory performance behavior. This is how, the listed hardware counters have been identified to be indicators of performance.

Work has been done to select hardware performance counters for understanding and optimizing memory performance behavior. For this purpose, we used various very simple assembly kernels: simple streaming loads and stores with various constant striding patterns and different variants essentially varying the unrolling degree. For all of these codes, we performed two tests using different data set sizes: 512 KB (fitting in the L2) to test the L1 versus L2 behavior, and 8 MB (exceeding L2 cache size) to test the L2 versus RAM behavior. For each of these codes, the miss pattern is easily predictable, however clearly

	F1	F2	F3	F4	F5	F6	F7
Maqao	X	X	X	-	-	-	-
Decan	-	-	-	-	X	X	-
`Hardware Performance Monitoring`	-	-	-	X	-	-	-
`Memory Traces`	-	-	-	-	-	-	X

Table 5.1: The features targeted in our performance evaluation process and the tool responsible of each feature. F1: Vectorization / F2: Dispersal on execution ports / F3: Estimation bound in L1, L2, and RAM / F4: Cache misses / F5: Load-Store impact / F6: 4K-aliasing / F7: Memory access patterns.

the behavior of the hardware prefetchers is unknown and hard to predict accurately. However, the codes are simple enough so the hardware prefetchers should be triggered. We monitored systematically various performance counters using PTU [15] and we tried to correlate performance behavior with the behavior of these various performance counters. The basic underlying assumption is that if we are unable to understand the behavior of performance counters on simple load/store kernels, there is little chance that we can use them efficiently on more complex kernels. This work about the selection of the hardware performance counters and all the measurements can be found in the deliverable D2.6.1_b [63], which was done for the ParMA project [7].

5.2.2.2 Memory traces

Binary-level instrumentation is used to collect memory traces to analyze memory access patterns. The memory traces provide a stride report for targeted instructions. Using this stride report, for example, instructions with a longer stride than operand size can be identified as potential performance bottlenecks [24]. An example using this type of information is given in Section 5.3.

5.2.3 Decremental analysis using Decan

When memory behavior is identified as a problem with a loop, given the imprecise nature of performance counters, it is often still difficult to know the specific delinquent instructions. The decremental analysis using Decan is performed to quickly identify such instructions.

Chapter 4 presents the concept as a simple one: first measure the original version of the code, and then measure a version of code modified by removing one or more expressions or instructions such as memory access instructions. Once an instruction is removed, the program is again profiled to account for the contribution of the removed instruction. Timing differences and deltas in L1 and L2 miss rates indicate an individual instruction's effect on a loop's overall performance.

5.2.4 Tools and targeted features

Table 5.1 lists the different features targeted in our performance evaluation process and shows which used tool, between the tools used, targets these features or some of them. This table shows that if we want to target all the features, different tools should be used, since different tools have different strengths.

5.3 Experimental Results

Here we show how the methodology previously described was applied to real-life applications. In the remainder of this section, we examine two codes, developed by Recom Services and Dassault-Aviation.

5.3.1 Experimental setup

The experimental platform consists of a computation node equipped with four Xeon X7350 Tigerton. Each Xeon processor is a quad-core chip clocked at 2.93 GHz, equipped with two 4 MB L2 caches, two cores share one L2 cache, and 32 kB L1 data cache, private to each core. There are 48 GB of RAM available on this node.

The Intel C and Fortran Compilers, icc and ifort v11.0, are used to generate all of our assembly codes. They also are used to generate OpenMP parallel regions when appropriate.

Intel's Performance Tuning Utility *PTU* is used to access hardware counters and perform part of the dynamic analysis.

5.3.2 3D Combustion simulation code

Brief description The most time consuming subroutine in `AIOLOS` is `RBgauss`, which implements a red-black iterative solver [87]. The `RBgauss` subroutine contains two loops (denoted *Red* and *Black*) with a communication between them using MPI. Figure 5.3 shows this loop.

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

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

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

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

Figure 5.3: The most time-consuming loop in `RBgauss`. There are 3 different arrays: AM(2D), and SU(1D) are read-only, PHI(1D) is read and written.

Static analysis According to MAQAO's static analysis, there are no vector instructions SSE instructions generated by ifort. However, looking at the execution port usage report, it becomes apparent that the main bottleneck is memory accesses, specifically loads from memory. The `P2` port that is dedicated to memory loads, shows a much higher access number than the other ports.

Looking at the source code, the explanation for not using vector instructions in the loop is obvious: the `AM` array is accessed indirectly through the `INDINR` index array, which prevents the compiler from knowing whether the data accessed is correctly aligned.

Dynamic analysis As both Red and Black loops are singly-nested loops, the iteration count and the loop bounds are one and the same for each loop. Memory tracing indicates that the `AM` and `PHI` arrays are accessed with a stride 2 basis, with some gaps from time to time. As the static analysis shows, the most time consuming operations in this loop are memory loads. Looking at the source code and loop attribute profiling, it is clear that this routine is memory-bound. As `AM` is accessed with a "stride 2" pattern, half of the memory bandwidth is wasted: only half of the bytes pulled into the cache are actually useful for performing computations, doubling the number of cache misses. Looking at performance counters confirms this.

In a multicore context, dynamic analysis shows a constant amount of L1D and L2 cache line consumption, independent of the number of threads used. Thus, the amount of cache misses per thread is constant, but the amount of CPU cycles increases, thus reducing the overall speedup to 4 with 16 threads. This is mainly due to memory bandwidth limitations which prevent memory bound programs from getting more than a speedup of four. When the memory bandwidth limit is reached it is no longer the latency of the memory accesses that limits the execution speed, but the number of accesses to main memory that the application causes.

Decremental Analysis Removing the memory access instructions has an impact on the analysis given by MAQAO by decreasing load pressure on port P2. However, since MAQAO cannot distinguish between different strides, its added-value remains limited in this particular case. Hence, when iterating the dynamic analysis with the modified loop, memory behaviors become more apparent: accesses to `PHI` occur almost always in cache, whereas accesses to `AM` are always RAM-based. The dynamic analysis is applied after the decremental analysis to account the effect of the removed instruction. DECAN was essential in this case to identify which memory accesses are causing the contention on the memory bus. Using DECAN with the grouping option allows to pinpoint precisely the blocking memory access in the `RBgauss` subroutine. Grouping consists in detecting the memory access instructions that are accessing to the same base address. This means that these memory accesses are accessing to the same source array. The group of instructions are patched at the same time, to see the impact of these accesses.

Optimization Since the major bottleneck for this routine is data access from RAM combined with a low spatial locality (stride 2 access), various optimizing transformations were performed, but only the following has a significant impact on performance: reshaping the array AM by getting rid of the stride 2 access. More precisely, the array `AM` is split into two distinct arrays still with indirect access but stride 1. This is equivalent to reshaping an array of complex numbers by splitting it into arrays, one containing the real part, the other one containing the imaginary part. Figure 5.4(b) shows the the red loop after the array reshaping optimization.

As expected, the indirect access still prevents the compiler from generating vector instructions. As such, MAQAO static analysis is still "blind" to our code transformation. However, dynamic analysis, and more specifically hardware counter measurements do show that cache misses are almost half what they used to be. Figure 5.5(b) shows the cache misses measurement. Figure 5.5(a) shows that the single core performance has been improved by speedups between 1.2 and 1.3 thanks to this code transformation.

When performing a new dynamic analysis with multiple threads, memory saturation is still the main problem, but saturation of the memory bus happens much later. Figure 5.6 shows that the performance is improved by speedups between 1.3 and 1.4.

5.3.3 Iterative solver for the Navier-Stokes equation

Brief description The most time-consuming subroutine in `ITRLSOL` [7, 40] is `EUFLUXm`, which implements a sparse matrix-vector product. The `EUFLUXm` subroutine contains two groups of quadruply nested loops (2 identical quadruply nested loops in each group) shown in Figure 5.7.

Static analysis The MAQAO vectorization report indicates that no vectorization is performed, no use of SSE instructions. The loads cannot be vectorized due to the non unit stride on the two vectors but the multiplications and the additions could have been vectorized by the compiler. However, the execution port usage report clearly indicates that

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

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

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

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

(a) The most time-consuming loop in `RBgauss` before optimization

```
DO IDO=1,NREDD
      INC  =  INDINR(IDO)
      INC_AMR = INDAMR[IDO]

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

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

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

(b) The most time-consuming loop in`RBgauss` after optimization

Figure 5.4: The most time-consuming loop in `RBgauss` before and after the array reshaping optimization.

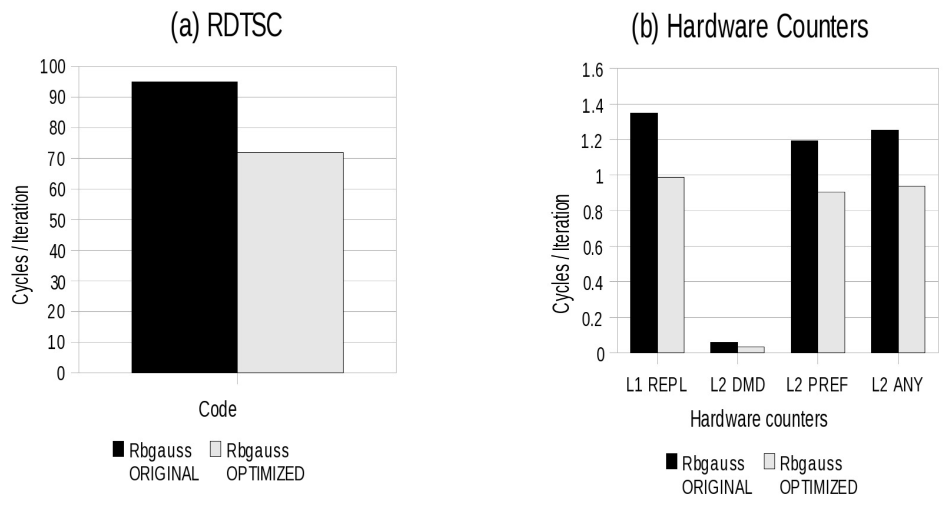


Figure 5.5: Speedup achieved on `RBgauss` with the AM reshaping optimization on unicore.

the vectorization of additions and multiplications will improve P0 and P1 ports, which are the computation units ports, but not the P2 loads port which is the bottleneck.

Dynamic analysis Loop attribute profiling indicates that the main specific feature in the 4-level-nested-loops is that the two innermost loop bounds ($ndof$) are quite small ($4 \leq ndof \leq 10$). The two outermost trip counts are larger and vary throughout execution.

Memory tracing shows that the two innermost loops are accessing all of the arrays along the wrong dimension (row-wise, which in Fortran is the wrong dimension for a traversal) leading to poor spatial locality. Moreover, the values of indexes used for accessing the first dimension of all arrays are not regular and lead to indirect addressing.

Decremental analysis Removing the memory accesses that access the arrays `ompl` and `ompu` shows that when modifying this group of loads the performance are increased, thus indicating that the access to these two arrays is the bottleneck.

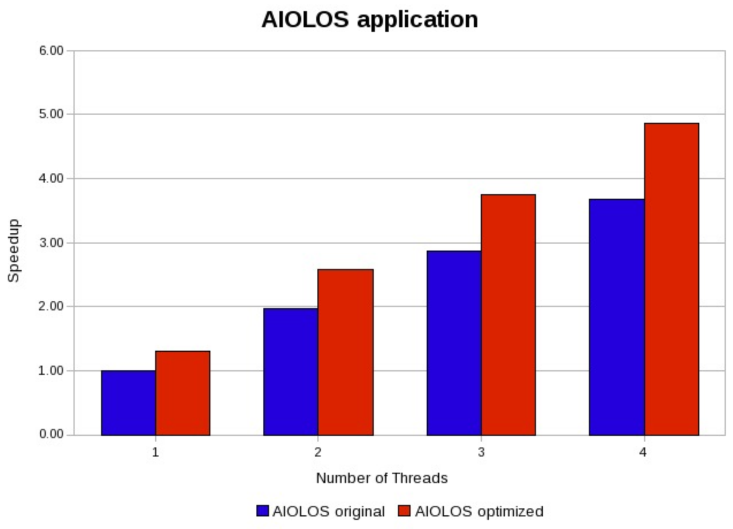


Figure 5.6: Speedup achieved on `RBgauss` with the AM reshaping optimizations for 1..4 threads. The gains obtained on unicore scale linearly with the number of cores in the system.

```
do cb=1,ncbt
   igp = isg
   isg = icolb(icb+1)
   igt = isg + igp
   do ig=1,igt
      e = ig + igp
      i = nnbar(e,1)
      j = nnbar(e,2)
      do k=1,ndof
         do l=1,ndof
            vecy(i,k) = vecy(i,k) + ompu(e,k,l)*vecx(j,l)
            vecy(j,k) = vecy(j,k) + ompl(e,k,l)*vecx(i,l)
         enddo
      enddo
   enddo
enddo
```

Figure 5.7: The most time-consuming quadruply nested loop in `EUFLUXm`. There are 4 different arrays: vecx(2D), ompu(3D), and ompl(3D) are read-only, vecy (2D) is read and written.

Optimization Since the key performance bottleneck for this routine is poor spatial locality because of the accesses on the wrong array dimension, various transformations are performed. Over the various code transformations that are performed on the code, two have a significant impact on performance: hardwiring `ndof` and loop interchange.

Value specialization involves replacing a variable whose value is unknown by the compiler *ndof* by its proper value equal to 4, to help the compiler apply optimizations, in particular for unrolling. The compiler fully unrolls the two innermost loops inside the loop nest. As no SIMD instructions were generated, MAQAO static analysis is fooled by the fact that the innermost loop is not the one it used to be. Hence no direct comparison with the previous reports can be made, except that, according to MAQAO static analysis, no loop vectorization occurred. A speedup of 1.5 is observed for sequential executions.

The second transformation is done by interchanging the second loop on *ig* and the two innermost loops. That means that the *ig* loop becomes the innermost loop. All of the arrays are now accessed column-wise (Figure 5.8(b)). The static analysis of this transformation with MAQAO shows that indirect accesses still prevent the compiler from vectorizing the loop. Statically, there is no information about the transformation. However, dynamic analysis shows that interchanging loops substantially increased the data traffic into L1 but drastically improved performance. Because of the data set size, the L2 traffic remains the same, but the hardware prefetch behavior is vastly improved Figure 5.9(b). Figure 5.9(a) shows this optimization improves sequential performance by a speedup of 2.5.

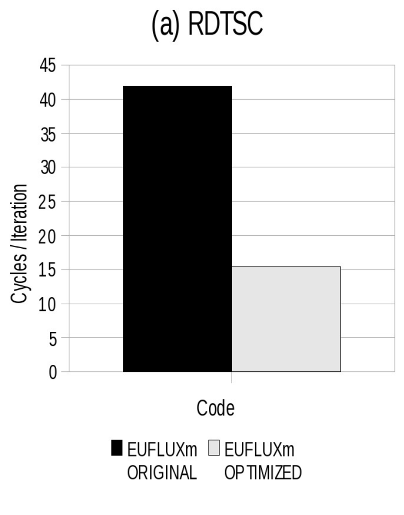
```
do cb=1,ncbt
        igp = isg
        isg = icolb(icb+1)
        igt = isg + igp
        do k=1,ndof
            do l=1,ndof
              do ig=1,igt
                 e = ig + igp
                  i = nnbar(e,1)
                  j = nnbar(e,2
                 vecy(i,k) = vecy(i,k) + ompu(e,k,l)*vecx(j,l)
                vecy(j,k) = vecy(j,k) + ompl(e,k,l)*vecx(i,l)
              enddo
            enddo
        enddo
enddo
```

(a) The most time-consuming loop in `EUFLUXm` before optimization

```
do cb=1,ncbt
        igp = isg
        isg = icolb(icb+1)
        igt = isg + igp
        do k=1,ndof
            do l=1,ndof
              do ig=1,igt
                 e = ig + igp
                  i = nnbar(e,1)
                  j = nnbar(e,2
                 vecy(i,k) = vecy(i,k) + ompu(e,k,l)*vecx(j,l)
                vecy(j,k) = vecy(j,k) + ompl(e,k,l)*vecx(i,l)
              enddo
            enddo
        enddo
enddo
```

(b) The most time-consuming loop in `EUFLUXm` after optimization

Figure 5.8: The most time-consuming quadruply nested loops in `EUFLUXm` before and after loop interchange optimization.



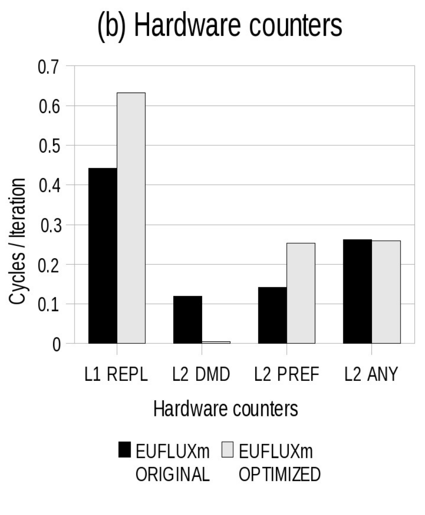


Figure 5.9: Speedup achieved on `EUFLUXm` with the loop interchange optimization on unicore.

In a multicore environment the same optimizations are applied. Variable specialization has an impact on the overall execution of `ITRLSOL`, but interchanging loops gives even better results, with a speedup of up to 2.5 This speedup is shown in Figure 5.10.

There is no doubt that improving data locality in a unicore environment has similar effects on a multicore environment, as the memory bus receives fewer requests. As previously discussed, speedup for memory-bound applications on the experimental platform cannot be greater than 4 due to bus saturation. Since the previous experiment was conducted on a 4-thread execution and gave a speedup around 3.5, it was decided to keep the same amount of threads for `ITRSOL`.

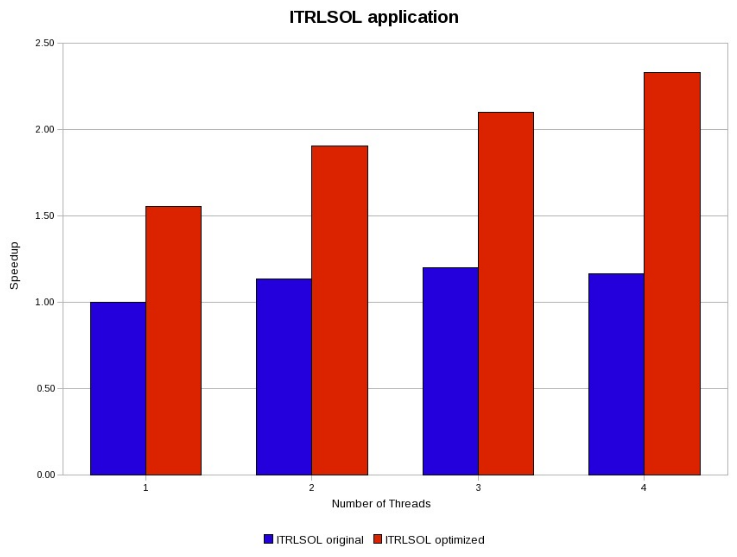


Figure 5.10: Speedup achieved on Dassault applications the loop interchange optimizations for 1..4 threads. The gains obtained on unicore scale linearly with the number of cores in the system.

5.4 Conclusion

In this chapter, we present a methodology to provide a semi-automatic way of analyzing and understanding performance issues for high-performance computing applications. This was done using a combination of different tools: MAQAO's static analysis is used to perform the first step of the process of evaluation. It gives a first view of the quality of the code in term of vectorization, execution ports saturation, etc. Then, a dynamic analysis is done using HPM counters, loop profiling and memory tracing to check cache misses and access pattern. Finally, DECAN and its debug concept is used as a fine-grain tool to pinpoint precisely the root cause of certain bottlenecks.

Better execution times are achieved for kernels used in real-life applications, with a speedup of up to 2.5.

This evaluation process, shows that there is no specific performance analysis tool that solves all the problems. The purpose is to have a good methodology which allows to explore the different aspects of the target program. For that purpose, no single tool is a panacea. However, different tools and techniques are useful to detect precisely the bottleneck and to apply the adequate optimization.

Chapter 6

Conclusion

6.1 Contributions of this thesis

This thesis presents two techniques for performance analysis on binary codes. Several performance analysis tools exist, based on statistical sampling, modelisation, instrumentation, targeting routines and loops but none of the existing tools targets the instructions directly or gives an estimation on the degree of vectorization of a loop.

The first contribution of this thesis, in chapter 3, presents Maqao's static analysis. The static analysis is considered as the first step in the process of performance evaluation. It is a fast process that abstracts the dynamic phenomena. It does not take into account the semantic of the code, since it does not execute the target code and it does not consider the input dataset. Maqao's static analysis implements a Core 2 pipeline model, and gives predictions of how the processor handles instruction stream throughout the pipeline. It computes performance estimates and evaluates the number cycles needed for the execution of the innermost loops. For each loop, several metrics are computed to give a cost of the quality of the code. First, an estimation in cycles of the time spent by the loop in the front-end is given. The front-end corresponds to the first stages of the pipeline: the predecoder stage whose purpose is to detect where each instruction begins. The decoder stage interprets the instructions by decomposing them into micro-operations. Finally, the reorder-buffer stage where the values of the renamed registers are stored. After the front-end, an estimate of the number of cycles spent in the back-end is given. It corresponds to the number of cycles that the loop spent in the execution ports.

After the front-end and the back-end other important estimations about the quality of the code are computed. A vectorization ratio is calculated to estimate the degree of vectorization of the loop. Moreover, an important part of the performance model is the pattern matching done to give an estimation of the performance of the loop in the memory hierarchy.

The drawback of the Maqao's static analysis is the known limitation in all static analysis tool: it does not consider any dynamic phenomena. It computes a lower bound without taking into account any dynamic behavior.

The second contribution of this thesis is the Decan tool, in chapter 4. Decan is a new approach for decremental performance analysis. It is a fine-grained performance analysis tool that target memory access instructions in regular loops. The concept of Decan is similar to the debug process that every programmer follows. When a bug occurs in a program, the developer generally modifies the code by removing and/or transforming some source code lines in the program and runs it to check if the bug still occurs or not. Decan follows a similar process. Bad performance is considered `a bug`. Decan perform on SSE memory instructions. It replaces SSE memory instructions with `nop` instructions, at the binary level. This replacement generates a set of modified binaries that are associated to performance. The replacement is performed in different ways: replace all loads at the same time, replace all stores at the same time, replace all loads and stores at the same time, replace just one memory access, or replace a group of dependent instructions.

For the evaluation time of all of these versions of binaries, Decan performs a kernel execution that allows to run the hot code region within its original execution environment without running the whole application. Moreover, any new binary version leading to a crash is considered by DECAN as an error, and the version is removed from the set of programs to analyze. Any version with an execution time larger than the original code is considered as an error and the version is also removed from the analysis. Also, another drawback that can occur is the generation of *Floating Point Exceptions(FPE)* that are not present in the original binary. Hence, the appearance of new FPE can affect the execution time. In order to tackle this problem, a small tool to detect and count the FPEs is implemented and used.

The last part of this dissertation shows a performance evaluation process based on the tools developed in this thesis. It describes a semi-automated methodology to analyze performance and guide the optimization process. Both static analysis MAQAO and DECAN analysis, of memory access, of the code are performed. This process has been performed on two real-life HPC applications: `RBgauss` from RECOM Services and `EUFLUXm` from Dassault-Aviation. Our modifications achieved a speedup of 1.4 for the `RBgauss` application, and a speedup of 2.5 for the `EUFLUXm` application, both in multicore.

6.2 Future Work

MAQAO's static analysis is based on an implementation of a performance model for Core 2 architecture. The MAQAO framework can be extended to target different other architectures. The actual framework targets the Core 2 architecture but the next step will be to implement other x86 architectures such as the Nehalem and the Sandy Bridge. As the performance model for the Core 2 is considered as a plugin in MAQAO, it is easy to add new plugins for new architectures. The main difficulty is in the understanding of the behavior of these new architectures to add. The better is the understanding, the most accurate the performance model will be.

DECAN as a new concept in performance analysis, opens different research directions.

DECAN is actually targeting SSE memory access instructions in regular loop. To broaden the scope of DECAN, loops with control flow can be addressed: in a non-linear control flow, identifying the particular execution path which triggers the performance problem is not a trivial task. For instance, such cases can confuse most of the approaches based on hardware counters. The model proposed by DECAN can be extended beyond memory instructions and address branch instruction as shown in Figure 6.1.

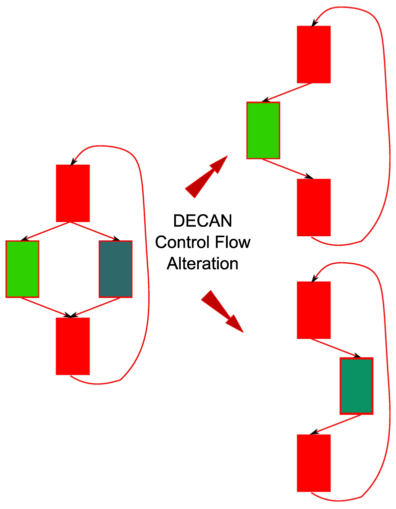


Figure 6.1: Addressing the control flow with DECAN.

The concept of Decremental Analysis can be extended to go from instructions to threads' tasks. This parallel decremental analysis can be applied to delete the work of one or several threads. The purpose is to detect scheduling problem, load balancing, or conflicting data location, and to know in which thread or group of threads do we have to look. Figure 6.2 illustrates how one, or more, thread's work can be *decanned*.

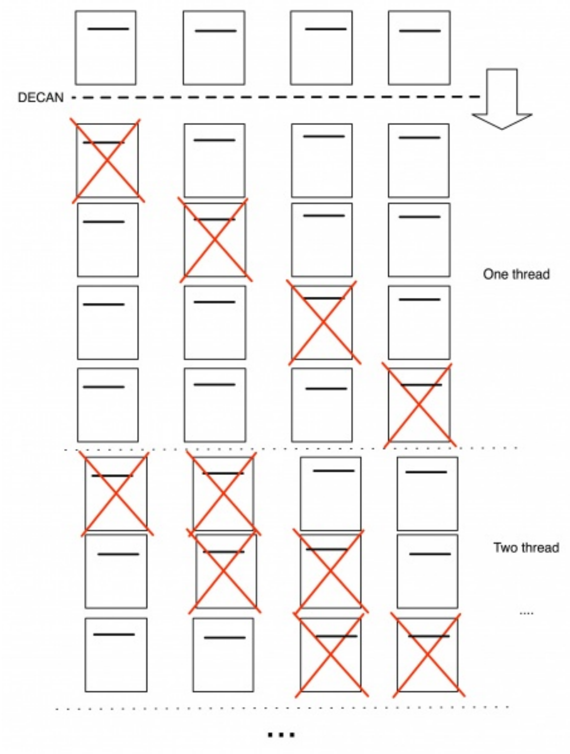


Figure 6.2: DECAN on one thread job.

Appendix A

Microbenchmarking on Core 2 and Nehalem

A.1 Microbenchmarks Description

In this section we describe the different kernels (ie. microbenchmarks) we have built. Some of them correspond to artificial kernels (load/store kernels written on assembly) and others represent real kernels (DAXPY/Copy kernels).

There are 9 groups of artificial kernels and 12 groups of real kernels. The artificial kernels have been built with two kinds of assembly instructions: aligned & unaligned instructions:

- Aligned instruction : data must be aligned on a 16-byte boundary.
- Unaligned instruction : data don't have to be aligned on a 16-byte boundary.

Each group contains 3 levels of unrolling of the kernel : unroll2, unroll4 and unroll8. Two kinds of unrolling are done (in case of more than 1 vector): vectorized & interleaved.

If the no-unrolled code is **X[i]Y[j]** then:

- The unrolled code (interleaved) is: **X[i]Y[j]..X[i+1]Y[j+1]..X[i+2]Y[j+2]**
- The unrolled code (vectorized) is: **X[i]X[i+1]X[i+2]..Y[j]Y[j+1]Y[j+2]..**

The assembly kernels are:

- **artificial kernels**
  - Lx : one load on vector X
  - Sx : one store on vector X
  - LxLy : two loads on 2 different vectors (X/Y)
  - LxSy : one load and one store on 2 different vectors (X/Y)
  - LxLySz : two loads and one store on 3 different vectors (X/Y/Z)
  - LyLxSy : one load on vector X and one load/store on vector Y
  - LxLyLzSt : three loads and one store on 4 different vectors (X/Y/Z/T)
  - LxLySzSt : two loads and two stores on 4 different vectors (X/Y/Z/T)
  - LxLyLzStSu : three loads and two stores on 5 different vectors (X/Y/Z/T/U)
- **real kernels**
  - Copy : one load & one store on the same vector.
  - DAXPYasm , DAXPY2asm , DAXPY3asm , DAXPY4asm , DAXPY5asm , DAXPY6asm

An example of LxSy kernel:

```
.L4:
    movaps  0(%rsi),%xmm2       #load [rsi] (X[i]) in xmm2
    movaps  %xmm1, 0(%rcx)      #store xmm1 in [rcx] (Y[i])

    addq    $16,%rsi            #address increment
    addq    $16,%rcx            #address increment
    sub     $2, %rdi            #counter
    jg      .L4
```

An example of LxSy kernel unroll 2:

```
        INTERLEAVED                             VECTORIZED
        -----------                             ----------
.L4:                                    .L4:
        movaps  0(%rsi),%xmm2                   movaps  0(%rsi),%xmm2
        movaps  %xmm1, 0(%rcx)                  movaps  16(%rsi),%xmm3

        movaps  16(%rsi),%xmm3                  movaps  %xmm1, 0(%rcx)
        movaps  %xmm4, 16(%rcx)                 movaps  %xmm4, 16(%rcx)

        addq    $32,%rsi                        addq    $32,%rsi
        addq    $32,%rcx                        addq    $32,%rcx
        sub     $4, %rdi                        sub     $4, %rdi
        jg      .L4                             jg      .L4
```

An example of a DAXPY2asm kernel:

```
        Y[i] = Y[i] + ALPHA1*X[i] + ALPHA2*Z[i]
        ---------------------------------------
.L4:
     movaps  0(%rsi),%xmm2          #load [rsi] (X[i]) in xmm2
     mulpd   %xmm0, %xmm2           #xmm2*=xmm0 (alpha1*X[i] in xmm2)
     addpd   0(%rcx),%xmm2          #xmm2+=[rcx] (xmm2+=alpha1*X[i])
     movaps  0(%rdx),%xmm3          #load [rdx] (Z[i]) in xmm3
     mulpd   %xmm1,%xmm3            #xmm3*=xmm1 (alpha2*Z[i] in xmm3)
     addpd   %xmm3,%xmm2            #xmm2+=xmm3 (xmm2+=alpha2*Z[i])
     movaps  %xmm2,0(%rcx)          #store xmm2 in Y[i]

     addq    $16,%rsi               #address increment
     addq    $16,%rdx               #address increment
     addq    $16,%rcx               #address increment
     sub     $2, %rdi               #counter
     jg      .L4
```

The C kernels are:

- DAXPY2c : a[i] += x1*b[i] + x2*c[i]
- DAXPY3c : a[i] += x1*b[i] + x2*c[i] + x3*d[i]
- DAXPY4c : a[i] += x1*b[i] + x2*c[i] + x3*d[i] + x4*e[i]
- DAXPY5c : a[i] += x1*b[i] + x2*c[i] + x3*d[i] + x4*e[i] + x5*f[i]
- DAXPY6c : a[i] += x1*b[i] + x2*c[i] + x3*d[i] + x4*e[i] + x5*f[i] + x6*g[i]

An example of DAXPY2c kernel:

```
for (i=0 ; i<N ; i++){
   A[i] = A[i] + alpha1*B[i] + alpha2*C[i];
}
```

An example of DAXPY4c kernel:

```
for (i=0 ; i<N ; i++){
  A[i] = A[i] + alpha1*B[i] + alpha2*C[i] + alpha3*D[i] + alpha4*E[i];
}
```

A.2 Methodology Description

In this section, we describe our methodology of the experimentation.

A.2.1 Assembly Kernels

We measured our assembly kernels in two INTEL architectures : Core 2 & Nehalem.

- Our measures are focused on the different levels of cache.
- The different codes have been measured thanks to a microbenchmarking framework.
- Different alignments have been measured.

The purpose is to analyze the behavior (unrolling impact) of ARTIFICIAL kernels on the target architecture on the different levels of cache (L1/L2 for CORE 2 - L1/L2/L3 for NEHALEM).

In our assembly kernels, we are using SSE instructions (2*64 bits = 128 bits). No unroll = 128 bits Unroll2 = 2 * 128 bits Unroll4 = 4 * 128 bits Unroll8 = 8 * 128 bits

A.2.2 C Kernels

The purpose with these different C versions of DAXPY is to force the compiler to ALIGN data so we can have the best performance for these codes on the focused architecture.

The command used is (Intel C Compiler 11.0):

```
icc  -S   -O3   -align   tested-kernel
```

A.3 Results Description

In the different tables we present, the columns represent:

- UNR: UNRoll
- ’2/4/8’ + ’I/V’: Unroll_Factor + Interleaved/Vectorized
- best : the best performance for the current kernel –> MIN(UNR 2I,UNR 2V,UNR 4I,UNR 4V,UNR 8I,UNR 8V)
- NO_UNRO / best: the profit obtained with the unrolling
- 2I / 8I: the profit obtained when we unroll 8 times (Interleaved form)

The results represent number of cycles per iteration in case of -level-cache- data (data fit in L1, L2 & RAM).

A.3.1 The performance with aligned instructions in assembly kernels

Each measurement given corresponds to the best measurement we obtained for the aligned code.

A.3.1.1 Cache level 1

Core 2

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	0.55	0.55	0.55	0.55				1.01	1.01
Sx	0.57	0.56	0.56	0.57				1.02	0.98
LxLy	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.00	1.00
LxSy	1.05	0.56	0.56	0.56	0.59	0.63	0.68	1.88	0.88
LxLySz	1.11	1.11	1.11	1.10	1.10	1.11	1.11	1.02	0.99
LxLySx	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.00	1.00
LxLyLzSt	1.62	1.61	1.62	1.61	1.61	1.64	1.64	1.00	0.99
LxLySzSt	1.62	1.21	1.39	1.25	1.38	1.42	1.54	1.35	0.85
LxLyLzStSu	1.64	1.63	1.63	1.62	1.62	1.96	1.84	1.01	0.83
COPY	1.06	0.57	0.57	0.57	0.58	0.56	0.64	1.89	1.02
DAXPY	1.45	1.06	1.05	1.06	1.05	1.05	1.05	1.38	1.01
DAXPY2asm	2.13	1.75	1.69	1.70	1.67	1.77	1.87	1.27	0.99
DAXPY3asm	2.88	2.39	2.32	2.34	2.25	2.34	2.48	1.28	1.02
DAXPY4asm	3.73	3.30	3.16	3.13	3.14			1.19	
DAXPY5asm	4.31	3.99	3.96	3.74	3.95			1.15	
DAXPY6asm	5.66	4.91	4.72					1.20	
DAXPY2c	2.85								
DAXPY3c	4.09								
DAXPY4c	6.33								
DAXPY5c	7.93								
DAXPY6c	9.71								

We can notice that when the unrolling level is increased, it is profitable with less than 5%. In most of cases, unroll2 is enough.

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.04	0.54	0.53	0.54				1.94	1.01
Sx	1.04	0.54	0.54	0.53				1.97	1.03
LxLy	1.51	1.03	1.03	1.03	1.03	1.04	1.04	1.46	0.99
LxSy	1.53	0.79	0.79	0.55	0.66	0.54	0.54	2.84	1.46
LxLySz	1.56	1.08	1.08	1.08	1.08	1.09	1.09	1.45	0.99
LxLySx	1.53	1.04	1.04	1.03	1.03	1.04	1.04	1.48	1.00
LxLyLzSt	2.06	1.80	1.58	1.59	1.59	1.62	1.62	1.30	1.11
LxLySzSt	2.08	2.02	1.80	1.09	1.09	1.22	1.19	1.91	1.65
LxLyLzStSu	2.12	2.07	1.85	1.61	1.61	1.64	1.64	1.32	1.27
COPY	1.49	0.78	0.79	0.55	0.55	0.54	0.54	2.77	1.45
DAXPY	1.55	1.05	1.05	1.05	1.05	1.04	1.04	1.49	1.01
DAXPY2asm	2.11	1.61	1.61	1.58	1.63	1.61	1.80	1.33	1.00
DAXPY3asm	2.87	2.45	2.37	2.15	2.17	2.17	2.46	1.33	1.12
DAXPY4asm	3.62	2.97	2.96	2.86	2.88			1.27	
DAXPY5asm	4.15	3.66	3.66	3.52	3.66			1.18	
DAXPY6asm	4.67	4.43	4.19					1.12	
DAXPY2c	1.83								
DAXPY3c	2.35								
DAXPY4c	3.24								
DAXPY5c	3.67								
DAXPY6c	4.17								

For all codes, the unrolling is profitable with more than 10%. The unroll2 and unroll4 are enough to reach the best performance. The unroll8 gives the best performance in case of codes with many Loads than Stores.

A.3.1.2 Cache level 2

Core 2

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.18	1.16	1.18	1.10				1.07	1.05
Sx	1.22	1.22	1.22	1.22				1.00	1.00
LxLy	2.09	2.11	2.10	2.11	2.00	2.05	1.94	1.08	1.03
LxSy	1.82	1.82	1.82	1.81	1.81	1.82	1.80	1.01	1.00
LxLySz	2.72	2.72	2.74	2.71	2.73	2.60	2.54	1.07	1.05
LxLySx	2.24	2.26	2.21	2.25	2.16	2.19	2.08	1.08	1.03
LxLyLzSt	3.60	3.59	3.62	3.62	3.69	3.52	3.57	1.02	1.02
LxLySzSt	4.20	4.15	3.88	4.19	3.60	4.21	3.66	1.17	0.99
LxLyLzStSu	4.81	4.79	4.66	4.79	4.59	4.82	4.46	1.08	0.99
COPY	1.82	1.81	1.81	1.81	1.80	1.81	1.80	1.01	1.00
DAXPY	2.26	2.25	2.22	2.24	2.17	2.19	2.10	1.07	1.03
DAXPY2asm	3.18	3.17	3.14	3.16	3.03	3.05	2.94	1.08	1.04
DAXPY3asm	3.90	3.98	3.96	3.97	3.95	3.89	3.71	1.05	1.02
DAXPY4asm	4.77	4.87	4.87	4.87	5.01			1.00	
DAXPY5asm	5.73	5.76	5.71	5.80	5.70			1.00	
DAXPY6asm	6.79	6.60	6.51					1.04	
DAXPY2c	3.67								
DAXPY3c	4.97								
DAXPY4c	6.89								
DAXPY5c	8.77								
DAXPY6c	10.90								

In most of cases, unrolling is not profitable and the unroll8 provokes a loss of performance.

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.05	1.04	1.03	0.85				1.23	1.22
Sx	1.01	0.83	0.83	0.82				1.22	1.00
LxLy	1.69	1.55	1.53	1.69	1.51	1.50	1.50	1.12	1.03
LxSy	1.49	1.28	1.28	1.24	1.26	1.29	1.31	1.20	0.99
LxLySz	2.04	1.99	2.00	2.05	2.02	2.04	2.08	1.03	0.97
LxLySx	1.85	1.86	1.78	1.87	1.67	1.62	1.71	1.14	1.15
LxLyLzSt	3.09	2.76	2.79	2.74	2.92	2.75	2.89	1.13	1.00
LxLySzSt	2.82	2.94	2.76	3.07	2.57	2.80	2.63	1.10	1.05
LxLyLzStSu	3.38	3.56	3.20	3.59	3.27	3.34	3.35	1.06	1.06
COPY	1.50	1.30	1.23	1.27	1.27	1.28	1.29	1.21	1.01
DAXPY	1.79	1.75	1.74	1.81	1.67	1.56	1.56	1.15	1.12
DAXPY2asm	2.35	2.28	2.27	2.33	2.31	2.29	2.27	1.03	1.00
DAXPY3asm	3.02	3.01	2.96	3.03	3.03	3.03	2.89	1.04	0.99
DAXPY4asm	4.13	3.78	3.76	3.84	3.78			1.10	
DAXPY5asm	4.74	4.64	4.57	4.71	4.61			1.04	
DAXPY6asm	5.80	5.43	5.51					1.07	
DAXPY2c	2.23								
DAXPY3c	3.11								
DAXPY4c	3.83								
DAXPY5c	4.79								
DAXPY6c	5.77								

The unrolling is more profitable than in the Core 2 architecture. The unroll8 reaches the best performance for the Load_kernels.

A.3.1.3 Cache level 3

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.12	1.12	1.12	1.05				1.07	1.07
Sx	1.21	1.21	1.21	1.21				1.00	1.00
LxLy	1.98	1.97	1.96	1.96	1.93	1.88	1.88	1.06	1.04
LxSy	2.04	1.95	1.95	1.94	1.94	1.94	1.95	1.05	1.00
LxLySz	2.96	2.94	2.94	2.94	2.93	2.85	2.83	1.05	1.03
LxLySx	2.49	2.47	2.47	2.48	2.44	2.33	2.29	1.09	1.06
LxLyLzSt	3.87	3.82	3.82	3.82	3.83	3.67	3.66	1.06	1.04
LxLySzSt	5.37	5.38	5.28	5.37	3.90	5.40	3.90	1.38	1.00
LxLyLzStSu	5.87	5.86	5.69	5.85	4.76	5.85	4.71	1.25	1.00
COPY	2.03	1.95	1.95	1.94	1.94	1.94	1.95	1.05	1.00
DAXPY	2.53	2.50	2.49	2.50	2.45	2.32	2.29	1.11	1.07
DAXPY2asm	3.38	3.31	3.31	3.34	3.30	3.10	3.10	1.09	1.07
DAXPY3asm	4.20	4.14	4.12	4.16	4.13	3.90	3.89	1.08	1.06
DAXPY4asm	5.09	4.95	4.90	5.06	4.93			1.04	
DAXPY5asm	5.94	6.53	5.75	5.83	5.70			1.04	
DAXPY6asm	6.88	6.75	6.62					1.04	
DAXPY2c	3.36								
DAXPY3c	4.19								
DAXPY4c	5.02								
DAXPY5c	5.99								
DAXPY6c	6.83								

A.3.1.4 RAM

Core 2

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	3.92	3.97	3.97	3.96				1.00	1.00
Sx	8.72	8.71	8.71	8.71				1.00	1.00
LxLy	7.25	7.25	7.26	7.27	7.25	7.26	7.38	1.00	1.00
LxSy	10.87	10.85	10.81	10.81	10.82	10.85	10.82	1.01	1.00
LxLySz	13.28	13.17	13.10	13.18	13.21	13.20	13.24	1.01	1.00
LxLySx	11.03	10.97	10.98	10.99	10.97	11.00	11.01	1.01	1.00
LxLyLzSt	16.76	16.74	16.67	16.73	16.59	16.68	16.57	1.01	1.00
LxLySzSt	20.75	20.77	20.78	20.80	20.71	20.81	20.84	1.00	1.00
LxLyLzStSu	23.54	23.43	23.48	23.51	23.24	23.52	23.39	1.01	1.00
COPY	10.85	10.84	10.82	10.84	10.82	10.81	10.82	1.00	1.00
DAXPY	11.13	10.94	10.97	10.94	10.99	10.97	11.04	1.02	1.00
DAXPY2asm	13.81	13.58	13.58	13.55	13.60	13.68	13.77	1.02	0.99
DAXPY3asm	17.32	17.20	17.09	17.05	16.76	16.98	16.87	1.03	1.01
DAXPY4asm	21.73	21.70	21.63	21.79	21.25			1.02	
DAXPY5asm	26.64	27.09	26.72	27.18	26.73			1.00	
DAXPY6asm	34.49	32.97	32.70					1.05	
DAXPY2c	13.80								
DAXPY3c	17.41								
DAXPY4c	23.13								
DAXPY5c	29.02								
DAXPY6c	35.38								

The unrolling has no effect. It is covered by the memory access.

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	2.06	1.71	1.71	1.66				1.25	1.03
Sx	2.58	2.54	2.57	2.60				1.02	0.98
LxLy	3.54	3.52	3.50	3.44	3.45	3.49	3.50	1.03	1.01
LxSy	5.20	4.84	4.73	4.70	4.69	4.70	4.68	1.11	1.03
LxLySz	6.86	6.80	6.78	6.63	6.59	6.63	6.78	1.04	1.02
LxLySx	5.00	5.25	4.76	4.76	4.77	4.90	4.75	1.05	1.07
LxLyLzSt	8.91	8.88	8.67	8.64	8.66	8.55	8.58	1.04	1.04
LxLySzSt	9.96	9.96	10.26	11.54	9.85	10.32	9.79	1.02	0.96
LxLyLzStSu	12.08	12.38	11.98	13.98	12.36	12.04	13.00	1.01	1.03
COPY	4.74	4.70	4.70	4.71	4.71	4.68	4.83	1.01	1.00
DAXPY	5.05	4.86	5.01	5.01	4.98	4.95	4.92	1.04	0.98
DAXPY2asm	6.49	6.24	6.29	6.17	6.18	6.18	6.23	1.05	1.01
DAXPY3asm	8.57	8.62	8.60	8.86	8.61	8.59	8.51	1.01	1.00
DAXPY4asm	10.84	14.79	11.16	11.13	10.74			1.01	
DAXPY5asm	13.01	12.85	12.89	12.91	12.84			1.01	
DAXPY6asm	15.05	14.93	14.95					1.01	
DAXPY2c	7.02								
DAXPY3c	9.08								
DAXPY4c	10.82								
DAXPY5c	12.86								
DAXPY6c	15.50								

The unrolling has no effect. It is covered by the memory access.

Synthesis: we can notice that the impact of the unrolling decreases with the cache levels. It is overlapped by the memory(cache) access. CORE 2 vs NEHALEM: the unrolling has more effect on NEHALEM than on CORE 2.

A.3.2 The performance with unaligned instructions in assembly kernels

Each measurement given corresponds to the best measurement we obtained for the unaligned code.

A.3.2.1 Cache level 1

Core 2

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.05	1.05	1.05	1.05				1.00	1.00
Sx	2.54	2.30	2.17	2.11				1.21	1.09
LxLy	2.05	2.05	2.05	2.05	2.05	2.06	2.06	1.00	1.00
LxSy	3.04	2.80	2.80	2.67	2.79	2.61	2.68	1.17	1.07
LxLySz	4.10	3.60	3.85	3.35	3.47	3.27	3.35	1.26	1.10
LxLySx	3.55	3.30	3.55	3.17	3.30	3.12	3.19	1.14	1.06
LxLyLzSt	5.11	4.35	4.36	3.98	3.99	3.86	4.26	1.32	1.13
LxLySzSt	6.10	5.61	5.85	5.37	5.50	5.33	5.40	1.15	1.05
LxLyLzStSu	7.12	6.37	6.37	6.01	6.02	5.91	6.32	1.20	1.08
COPY	3.05	2.80	2.80	2.67	2.80	2.61	2.68	1.17	1.07
DAXPY	1.86	1.56	1.56	1.56	1.56	1.55	1.55	1.20	1.00

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.04	0.69	0.69	0.69				1.51	0.99
Sx	1.04	0.78	0.78	0.77				1.34	1.01
LxLy	1.52	1.29	1.29	1.29	1.29	1.30	1.30	1.18	0.99
LxSy	1.49	0.82	0.82	0.80	0.80	0.80	0.79	1.88	1.04
LxLySz	1.57	1.32	1.32	1.32	1.32	1.36	1.36	1.19	0.97
LxLySx	1.53	1.30	1.30	1.30	1.30	1.31	1.31	1.18	0.99
LxLyLzSt	2.09	2.08	2.08	2.10	1.97	2.13	2.04	1.06	0.98
LxLySzSt	2.08	2.03	1.81	1.56	1.57	1.70	1.63	1.33	1.20
LxLyLzStSu	2.12	2.09	2.09	2.11	1.98	2.20	2.13	1.07	0.95
COPY	1.50	0.83	0.82	0.80	0.81	0.79	0.80	1.90	1.04
DAXPY	1.60	1.19	1.18	1.18	1.17	1.17	1.17	1.37	1.01

A.3.2.2 Cache level 2

Core 2

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.91	1.89	1.89	1.81				1.05	1.04
Sx	2.51	2.26	2.13	2.07				1.21	1.09
LxLy	3.37	3.34	3.32	3.36	2.67	3.24	3.16	1.27	1.03
LxSy	3.01	2.76	2.76	2.65	2.76	2.60	2.64	1.16	1.06
LxLySz	4.02	3.94	3.71	3.98	3.67	3.93	4.07	1.10	1.00
LxLySx	3.71	3.77	3.77	3.78	3.38	3.76	3.89	1.10	1.00
LxLyLzSt	5.06	5.14	4.88	5.14	4.70	5.12	5.66	1.08	1.00
LxLySzSt	6.01	5.52	5.76	5.62	5.40	5.82	5.59	1.11	0.95
LxLyLzStSu	7.07	6.61	6.46	6.60	5.96	6.58	8.09	1.19	1.00
COPY	3.01	2.79	2.76	2.65	2.76	2.62	2.64	1.15	1.06
DAXPY	2.82	2.80	2.79	2.79	2.73	2.62	2.54	1.11	1.07

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.17	1.17	1.17	0.94				1.24	1.24
Sx	1.01	0.84	0.84	0.84				1.20	1.00
LxLy	1.91	1.86	1.83	1.89	1.89	1.73	1.56	1.23	1.08
LxSy	1.52	1.39	1.42	1.39	1.40	1.39	1.39	1.09	1.00
LxLySz	2.28	2.28	2.28	2.28	2.28	2.26	2.26	1.01	1.01
LxLySx	2.17	2.04	2.02	2.10	2.01	1.83	1.81	1.20	1.12
LxLyLzSt	3.04	3.03	3.03	3.03	3.08	3.02	3.04	1.01	1.00
LxLySzSt	5.17	5.13	4.83	5.07	4.50	5.01	1.97	2.62	1.02
LxLyLzStSu	5.40	5.19	4.99	5.17	4.92	5.30	3.92	1.38	0.98
COPY	1.52	1.44	1.39	1.38	1.39	1.38	1.39	1.10	1.04
DAXPY	1.94	1.84	1.83	1.94	1.81	1.60	1.58	1.22	1.15

A.3.2.3 Cache level 3

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.28	1.27	1.27	1.11				1.15	1.15
Sx	1.22	1.21	1.21	1.22				1.00	1.00
LxLy	2.19	2.15	2.14	2.15	2.16	1.94	1.94	1.13	1.10
LxSy	2.18	2.11	2.11	2.11	2.11	2.08	2.08	1.05	1.02
LxLySz	3.24	3.20	3.20	3.20	3.22	3.02	3.04	1.07	1.06
LxLySx	2.72	2.69	2.67	2.70	2.67	2.53	2.52	1.08	1.06
LxLyLzSt	4.21	4.14	4.17	4.16	4.26	3.90	3.97	1.08	1.06
LxLySzSt	6.67	6.67	6.51	6.66	6.28	6.75	2.44	2.73	0.99
LxLyLzStSu	7.21	7.19	7.06	7.21	6.87	7.29	5.94	1.21	0.99
COPY	2.18	2.11	2.11	2.11	2.11	2.08	2.08	1.05	1.01
DAXPY	2.59	2.57	2.56	2.57	2.53	2.39	2.37	1.09	1.07

A.3.2.4 RAM

Core 2

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	4.22	4.22	4.21	4.24				1.00	0.99
Sx	8.79	8.79	8.79	8.79				1.00	1.00
LxLy	7.80	7.84	7.84	7.85	7.82	7.99	8.04	1.00	0.98
LxSy	11.35	11.30	11.32	11.32	11.32	11.37	11.32	1.01	0.99
LxLySz	14.27	14.32	14.23	14.29	14.33	14.29	14.76	1.00	1.00
LxLySx	11.61	11.51	11.53	11.53	11.54	11.80	12.02	1.01	0.98
LxLyLzSt	18.46	18.91	18.92	18.32	18.29	18.13	18.92	1.02	1.04
LxLySzSt	21.72	21.95	22.27	21.88	22.32	21.95	16.90	1.28	1.00
LxLyLzStSu	25.43	26.38	26.32	26.11	26.54	25.93	26.34	1.00	1.02
COPY	11.34	11.31	11.36	11.31	11.26	11.39	11.34	1.01	0.99
DAXPY	11.17	11.12	11.08	11.13	11.17	11.15	11.26	1.01	1.00

Nehalem

	NO UNR	UNR 2I	UNR 2V	UNR 4I	UNR 4V	UNR 8I	UNR 8V	NO UNR/ best	2I/ 8I
Lx	1.61	1.80	1.58	1.45				1.11	1.24
Sx	1.81	1.84	1.73	1.78				1.05	1.03
LxLy	3.58	3.52	3.63	3.56	3.60	3.54	3.47	1.03	0.99
LxSy	5.97	4.65	4.64	4.63	5.76	4.62	4.73	1.29	1.00
LxLySz	6.36	6.16	6.18	6.20	6.19	6.19	6.09	1.04	0.99
LxLySx	4.78	4.71	4.72	4.70	4.86	4.69	4.85	1.02	1.00
LxLyLzSt	8.62	8.67	8.49	8.74	8.75	8.48	8.73	1.02	1.02
LxLySzSt	10.15	10.41	10.37	10.49	10.34	10.09	5.02	2.02	1.03
LxLyLzStSu	12.54	12.55	12.55	12.11	12.47	12.53	12.46	1.04	1.00
COPY	4.76	4.64	4.59	4.62	4.75	4.75	4.75	1.04	0.98
DAXPY	5.00	4.96	4.80	4.78	4.78	4.75	4.85	1.05	1.04

A.3.3 The performance measurements in C kernels

In this section we will give some remarks about our compilation of the DAXPY kernels with **-align** option and **ivdep/vector aligned** pragmas.

Compilation with ICC 11.0 and -align option:

The **-align** option has no effect on the kernels. The assembly code generated still contains unaligned instructions.

DAXPY2 and DAXPY3 : packed instructions are used.

DAXPY4, DAXPY5 and DAXPY 6 : no packed instructions used.

Compilation with ivdep/vector aligned pragmas:

The **ivdep** pragma instructs the compiler the ignore vector dependencies so the code is vectorized : movsd/movhpd for load & movaps for stores.

The **vector aligned** pragma instructs the compiler to use aligned data so the movsd/-movhpd instructions used for loads are changed to movaps.

A.3.3.1 The performance measurements without pragma unroll

In these tests, no pragma UNROLL and no pragma NOUNROLL are used.

Core 2

	L1	L2	RAM
DAXPY2c	2.21	3.08	13.50
DAXPY3c	3.10	3.93	17.00
DAXPY4c	4.25	4.74	21.57
DAXPY5c	4.17	5.69	26.96
DAXPY6c	5.81	6.54	33.07

Nehalem

	L1	L2	L3	RAM
DAXPY2c	1.83	2.23	3.36	7.02
DAXPY3c	2.35	3.11	4.19	9.08
DAXPY4c	3.24	3.83	5.02	10.82
DAXPY5c	3.67	4.79	5.99	12.86
DAXPY6c	4.17	5.77	6.83	15.50

A.3.3.2 The performance measurements with pragma unroll

The pragma UNROLL is used in the case of unroll 2/4/8 and pragma NOUNROLL is used for the no unroll case (we force the compiler not to unroll).

A/ Core 2
Cache level 1

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	2.24	2.21	2.21	2.15
DAXPY3c	3.08	3.10	3.10	3.15
DAXPY4c	3.65	4.25	4.25	4.19
DAXPY5c	4.17	4.90	5.23	5.09
DAXPY6c	5.81	6.05	6.17	6.17

Cache level 2

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	3.09	3.08	3.08	2.93
DAXPY3c	3.94	3.93	3.93	3.73
DAXPY4c	4.74	4.74	4.75	4.65
DAXPY5c	5.70	5.73	5.78	5.60
DAXPY6c	6.54	6.62	6.70	6.46

RAM

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	13.95	13.61	13.51	13.59
DAXPY3c	17.32	16.96	16.99	16.98
DAXPY4c	21.74	21.56	21.57	21.44
DAXPY5c	27.09	26.97	27.00	26.91
DAXPY6c	32.98	33.18	33.50	33.28

B/ Nehalem
Cache level 1

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	2.10	1.83	1.83	1.87
DAXPY3c	2.62	2.36	2.35	2.53
DAXPY4c	3.15	3.25	3.25	3.24
DAXPY5c	3.67	3.65	3.83	4.01
DAXPY6c	4.17	4.18	4.42	4.44

Cache level 2

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	2.31	2.33	2.33	2.29
DAXPY3c	3.03	3.03	3.04	3.04
DAXPY4c	3.91	3.91	4.00	3.77
DAXPY5c	4.71	4.76	4.65	4.63
DAXPY6c	5.90	5.58	5.80	5.33

Cache level 3

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	3.41	3.36	3.36	3.11
DAXPY3c	4.24	4.18	4.20	3.95
DAXPY4c	5.05	5.01	5.03	4.79
DAXPY5c	6.62	5.90	6.74	5.73
DAXPY6c	7.58	6.80	6.81	6.54

RAM

	NO UNR	UNR 2	UNR 4	UNR 8
DAXPY2c	6.83	6.79	6.76	6.67
DAXPY3c	8.98	8.66	8.71	8.95
DAXPY4c	11.24	10.65	10.75	10.65
DAXPY5c	13.58	12.89	13.42	15.19
DAXPY6c	14.98	14.95	15.42	14.92

C/ Conclusion The pragma NOUNROLL is respected by the compiler. This pragma forces the compiler not to unroll and the compiler does. The pragma UNROLL has no effect on the code.

Bibliography

[1] Acumem. http://www.roguewave.com/. 29

[2] Core microarchitecture. http://en.wikipedia.org/wiki/Core_%28microarchitecture%29. xii, 40

[3] Dyninst: An application program interface (api) for runtime code generation. http://www.dyninst.org/. 28, 29, 35

[4] Intel VTune: software performance analyzer for x86-based machines. http://www.intel.com/software/products/vtune. 31, 35, 57, 59

[5] Intel(R) 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference. http://www.intel.com/Assets/PDF/manual/253666.pdf, http://www.intel.com/Assets/PDF/manual/253667.pdf. 64

[6] Lua. http://www.lua.org. 36

[7] ParMA: Parallel programming for multi-core architectures - ITEA2 project (06015). http://www.parma-itea2.org. xii, 51, 53, 54, 70, 71, 81, 83

[8] PGAS: Partitioned global address space. http://pgas.org. 29

[9] PIN: a dynamic binary instrumentation tool. http://www.pintool.org. 35

[10] Valgrind: instrumentation framework for building dynamic analysis tools. http://www.valgrind.org. 31, 57

[11] Vampir. http://www.vampir.eu. 28

[12] Likwid: Like i know what i am doing, 2011. http://code.google.com/p/likwid/. 27

[13] M. Abrash. *Graphics Programming Black. Chapter 21.* Morgan Kaufmann, 2007. 14

[14] A. Aggarwal. Energy efficient asymmetrically ported register files. pages 2–7, 2003. 17

[15] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization made easier with intel performance-tuning utility, 2007. 31, 35, 78, 79, 81

[16] S. Almeida. Neural branch prediction. 2006. 20

[17] AMD. Amd64 architecture programmers manual volume 4 : 128-bit media instructions. http://support.amd.com/us/Processor_TechDocs/26568.pdf, 2007. 3

[18] AMD. Amd64 architecture programmers manual volume 1 : Application programming. http://support.amd.com/us/Processor_TechDocs/24592.pdf, 2009. 3

[19] AMD. Amd64 architecture programmers manual volume 3 : General purpose and system instructions. http://support.amd.com/us/Processor_TechDocs/24594.pdf, 2009. 3

[20] AMD. Amd64 architecture programmers manual volume 5 : 64-bit media and x87 floatingpoint instructions. http://support.amd.com/us/Processor_TechDocs/26569.pdf, 2009. 3

[21] AMD. Amd64 architecture programmers manual volume 6 : 128-bit and 256-bit xop and fma4 instructions. http://support.amd.com/us/Embedded_TechDocs/43479.pdf, 2009. 3

[22] AMD. Amd64 architecture programmers manual volume 2 : System programming. http://support.amd.com/us/Processor_TechDocs/24593.pdf, 2010. 3

[23] J.-L. Baer. *Microprocessor Architecture : From Simple Pipelines to Chip Multiprocessors*. Cambridge, 2010. 12

[24] D. Barthou, A. C. Rubial, W. Jalby, S. Koliai, and C. Valensi. Performance tuning of x86 openmp codes with maqao. In *Parallel Tools Workshop*, pages 95—113, Dresden, Germany, Sept. 2009. Springer-Verlag. vii, 40, 41, 43, 45, 47, 49, 56, 81

[25] K. Beyls. The processor-memory gap: Cache remapping and related techniques, 2000. 4

[26] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998. 57

[27] U. Bondhugula and J. Ramanujam. P.: Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, 2008. 6

[28] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000. 28

[29] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 62:1–62:15, Piscataway, NJ, USA, 2008. IEEE Press. 25

[30] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. xi, 23, 24, 25, 26

[31] J. F. Cantin and M. D. Hill. Cache performance for spec cpu2000 benchmarks, 2003. xi, 7

[32] C. Carvalho. The gap btween processor and memory speeds. In *Proceedings of IEEE International Conference on Control and Automation*, 2002. 4

[33] I.-H. Chung and J. K. Hollingsworth. Automated cluster-based web service performance tuning. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 36–44, Washington, DC, USA, 2004. IEEE Computer Society. 29

[34] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999. 57

[35] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm, 2001. 39

[36] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 29

[37] D.C.Burger and T.M.Austin. The simplescalar tool set, version 2.0. 25(3), 1997. 57

[38] M. R. de Alba and D. R. Kaeli. Path-based hardware loop prediction, 2002. 20

[39] D. J. Department. Fast path-based neural branch prediction, 2003. 20

[40] Q. V. Dinh, A. Naïm, and G. Petit. Projet fame2: rapport final de synthèse sur l'optimisation des logiciels de simulation numérique de l'aéronautique. Technical report, Dassault Aviation, 2007. xii, 51, 53, 54, 70, 71, 83

[41] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. Exploring Application Performance: a New Tool For a Static/Dynamic Approach. Santa Fe, NM, Oct. 2005. 35, 63, 71, 79

[42] U. Drepper. What every programmer should know about memory, 1997. xi, 4, 5, 6, 9, 10

[43] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 167–178, Washington, DC, USA, 1998. IEEE Computer Society. 20

[44] S. Eranian. Perfmon2: a flexible performance monitoring for linux, 2006. 79

[45] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *In Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, pages 40–51, 1995. 17

[46] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988. 6

[47] P. Feautrier. Automatic parallelization in the polytope model. In *Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex*, pages 79–103. Springer-Verlag, 1996. 6

[48] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Interaction cost and shotgun profiling. *ACM Trans. Archit. Code Optim.*, 1(3):272–304, 2004. 57

[49] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. an optimization guide for assembly programmers and compiler makers, 2011. http://www.agner.org/optimize/. 15, 16, 17, 19, 40, 43, 44, 47

[50] GCC: the gnu compiler collection, 2011. http://gcc.gnu.org/. 36

[51] GDB: The gnu project debugger, 1986. http://www.gnu.org/sotware/gdb. 69

[52] S. Graham, P. Kessler, and M. McKusick. GProf: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982. 57

[53] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM. 78

[54] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2007. 12

[55] Intel Itanium Architecture Software Developer's Manual. http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm, 2010. 36

[56] IBM. IBM and motorola. Powerpc microprocessor family : The programmers reference guide. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600741775`, 1995. 3

[57] IBM. IBM powerpc 970fx risc microprocessor, 2005. 3

[58] Intel. Intel64 and IA-32 architectures optimization reference manual. http://www.intel.com/Assets/PDF/manual/248966.pdf, 2010. 3

[59] Intel. Intel64 and IA-32 architectures softwares developper manual, instruction set reference a-m. http://www.intel.com/Assets/PDF/manual/248966.pdf, 2010. 3

[60] Intel. Intel64 and IA-32 architectures softwares developper manual, instruction set reference n-z. http://www.intel.com/Assets/PDF/manual/248966.pdf, 2010. 3, 11

[61] Intel. Intel64 and IA-32 architectures softwares developper's manual, basic architecture. http://www.intel.com/Assets/PDF/manual/248966.pdf, 2010. 3, 36

[62] Intel Compilers and Libraries - Intel Software Network. http://software.intel.com/en-us/articles/intel-compilers/, 2011. 36

[63] M. Ivascot, W. Jalby, S. Koliai, and S. Zuckerman. Deliverable 2.6.1_b: Prototype of optimization tool for multithreaded codes: Identifying key performance counters. Technical report, University of Versailles, France, 2009. `http://www.parma-itea2.org`. 81

[64] W. Jalby, C. Lemuet, and X. L. Pasteur. A New Set of Microbenchmarks to Explore Memory System Performance for Scientific Computing, 2004. International Journal of High Performance Computing Applications. 48

[65] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons, 2001. 20

[66] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby. A balanced approach to application performance tuning. In *LCPC*, pages 111–125, 2009. vii, 40, 41, 43, 45, 47, 49, 56

[67] A. Lakshminarayanan and S. Shriraghavan. Introduction: Neural branch prediction, 2004. 20

[68] D. Libes. Exploring Expect: A Tcl-based toolkit for automating interactive programs, 1994. 70

[69] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. 28

[70] Link-OProfile. OProfile: system-wide profiler for Linux systems, capable of profiling all running code at low overhead. `http://oprofile.sourceforge.net`. 57, 59

[71] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005. 29, 57

[72] J. Magee, Q. Yi, and R. C. Whaley. Automated Timer Generation for Empirical Tuning. In *The 4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion (SMART)*, Pisa, January 2010. 68

[73] J. Mellor-Crummey. Hpctoolkit: performance tools for scientific computing. 24, 57

[74] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995. 29

[75] B. Mohr and F. Wolf. Kojak - a tool set for automatic performance analysis of parallel programs. In *Euro-Par*, pages 1301–1304. Springer-Verlag, 2003. 29

[76] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 2007 International Conference on Computing Frontiers*, May 2007. xi, 29, 30

[77] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA) held in conjunction with ASPLOS-12*, October 2006. 29

[78] T. Moseley, N. Vachharajani, and W. Jalby. Hardware performance monitoring for the rest of us. 2009. 79

[79] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Trans. Comput. Syst.*, 16:55–92, February 1998. 11

[80] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999. 57, 79

[81] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997. 39

[82] J. Noudohouenou and W. Jalby. Prediction of the best unroll factor using static performance characterization. Technical report, Exascale Computing Research Center, Versailles, France, 2011. 51

[83] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14:2003, 2003. 6

[84] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors. Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995. 28

[85] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes.* Cambridge University Press, Cambridge, 1986. xvii, 51, 52

[86] X. Qian, H. Huang, Z. Duan, J. Zhang, N. Yuan, Y. Zhou, H. Zhang, H. Cui, and D. Fan. Optimized register renaming scheme for stack-based x86 operations. In *Proceedings of the 20th international conference on Architecture of computing systems*, ARCS'07, pages 43–56, Berlin, Heidelberg, 2007. Springer-Verlag. 16

[87] B. Risio, N. Passmann, F. Wessel, and E. Reinartz. 3d-flame modelling in power plant applications. In *Proceedings of the High-Performance-Computing on Vector Systems*, 2008. xii, 51, 53, 70, 82

[88] R. Schreiber, J. J. Dongarra, R. Schreiber, J. J. Dongarra, and J. J. D. T. Automatic blocking of nested loops, 1990. 6

[89] S. Shende, A. Malony, S. Moore, P. Mucci, and J. Dongarra. Integrated tool capabilities for performance instrumentation and measurement. 2007. 27, 57

[90] S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006. xi, 27, 28

[91] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002. 57

[92] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 209–218, New York, NY, USA, 2009. ACM. 11

[93] The Standard Performance Evaluation Corporation. `http://www.specbench.org`. 30

[94] B. Strong. A Look Inside Intel: The COre (Nehalem) Microarchitecture, 2008. `www.cs.utexas.edu/users/cart/arch/beeman.ppt`. xii, 46

[95] H.-H. Su, M. Billingsley, and A. D. George. Parallel performance wizard: A performance system for the analysis of partitioned global-address-space applications. *Int. J. High Perform. Comput. Appl.*, 24:485–510, November 2010. 29

[96] S. Thoziyoor and N. Muralimanohar. Cacti 5.0, 2007. 11

[97] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM journal*, 1967. 16

[98] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003. 57

[99] T. Uliel. Intel architecture code analyzer. `http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/`, 2010. xi, 22, 23

[100] C. Valensi and D. Barthou. MADRAS: Multi-Architecture Disassembler and Reassembler. http://maqao.prism.uvsq.fr/wiki/wiki/MadrasDownload, 2009. 36, 37, 63

[101] L. N. Vintan. Towards a high performance neural branch predictor. In *In Proceedings of the International Joint Conference on Neural Networks*, pages 868–873, 1999. 20

[102] Z. Wang and L. R. B. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society. 10

[103] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998. 57

[104] H. A. Wijshoff. Implementing sparse BLAS primitives on concurrent/vector processors: a case study. Technical Report no. 843, Center for Supercomputing Research and Development, University of Illinios, 1989. xii, 46, 63, 64

[105] Wikipedia. `http://support.amd.com/us/Processor_TechDocs/24593.pdf`. xi, 7

[106] F. Wolf, B. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proc. of the 2nd HLRS Parallel Tools Workshop*, pages 157–167, Stuttgart, Germany, July 2008. Springer. ISBN 978-3-540-68561-6. 35

[107] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992. 20