Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study

Abdelhafid Mazouz, Sid Touati, Denis Barthou

UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN YVELINES

Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study

Abdelhafid MAZOUZ — Sid-Ahmed-Ali TOUATI — Denis BARTHOU

N° HAL-inria-00514548

July 2010

Domaine 1

RR--HAL-inria-00514548--FR+ENG

Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study

Abdelhafid Mazouz* , Sid-Ahmed-Ali Touati† , Denis Barthou ‡

Domaine : Sciences et techniques
Équipe-Projet ARPA

Rapport de recherche n° HAL-inria-00514548 — July 2010 — 34 pages

Abstract: The recent growth in the number of precessing units in today's multicore processor architectures enables multiple threads to execute simultanesiouly achieving better performances by exploiting thread level parallelism. With the architectural complexity of these new state of the art designs, comes a need to better understand the interactions between the operating system layers, the applications and the underlying hardware platforms. The ability to characterise and to quantify those interactions can be useful in the processes of performance evaluation and analysis, compiler optimisations and operating system job scheduling allowing to achieve better performance stability, reproducibility and predictability.

We consider in our study performances instability as variations in program execution times. While these variations are statistically insignificant for large sequential applications, we observe that parallel native OpenMP programs have less performance stability. Understanding the performance instability in current multicore architectures is even more complicated by the variety of factors and sources influencing the applications performances.

Key-words: OpenMP, Multicore, Parallelism, Performance evaluation

* PRiSM, UVSQ. Abdelhafid.Mazouz@prism.uvsq.fr
† PRiSM, UVSQ. Sid.Touati@uvsq.fr
‡ University of Bordeaux. Denis.Barthou@inria.fr

UFR des sciences
45 avenue des Etats Unis,
78035 Versailles cedex
Téléphone : +33 1 39 25 41 12 — Télécopie : +33 1 39 25 46 04

Mesures et analyse des variations des temps d'exćutions des programmes sur un cas d'architecture multi-cœurs

Résumé : L'accroissement des unités de calculs dans les nouvelles architectures des processeurs multicœurs permet à plusieurs processus de s'exécuter simultanément afin d'obtenir des meilleures performances en exploitant un parallélisme de tâches. Avec la croissante complexité de ce nouveau type d'architectures, il est primordial de bien comprendre les interactions qui existent entre les couches du système d'exploitation, les applications et l'architecture matérielle. L'habilité de bien caractériser et de quantifier ces interactions peut être utile dans les processus d'évaluation et d'analyse des performances, des optimisations de code appliquées par le compilateur et pour l'ordonnanceur de tâches du système d'exploitation. Une bonne compréhension de ces intéractions peut conduire à une meilleure stabilité, reproductibilité et prédictibilité des performances.

Nous considérons dans notre étude que l'instabilité des performances est la variabilité dans les temps d'exécution des programmes. Bien que ces variations sont insignifiantes pour les applications séquentielles, nous avons observé que les programmes parallèles écrits avec le standard OpenMP ont moins de stabilité dans les performances. Comprendre cette instabilité dans le cadre des architectures multicœurs est rendu encore plus compliqué par la variété des facteurs et des sources influençant les performances des applications.

Mots-clés : OpenMP, multicœurs, Parallélisme, Évaluation des performances

Contents

1 Introduction **4**

2 Experimentation **5**
2.1 Experimental setup 5
2.2 Experimental methodology 5

3 Variability of SPEC CPU2006 execution times **6**

4 Variability of SPEC OMP2001 execution times **8**

5 Study of co-running processes impact on programs performance variability **9**
5.1 Experimental setup 9
5.2 Study of system load impact on micro-benchmarks execution times 11
5.2.1 Micro-benchmarks code 11
5.2.2 Micro-benchmarks execution times under co-running tasks 15
5.2.3 Affinity impact on micro-benchmarks execution times 17
5.3 Memory page size and hardware prefetcher impact on performance variability 18
5.4 CPU-bound micro-benchmarks 22
5.5 Variability of execution times under the influence of other co-running processes 24
5.5.1 Controlling the co-running processes time spent in the sleeping state 24
5.5.2 The impact of accessing the L2 cache by the co-running processes on the micro-benchmarks performance 26

6 Study of the affinity effect on SPEC OMP2001 execution times **27**

7 Related Research Activity **32**
7.1 Variability of program execution times 32
7.2 Tools for performance measurement 32

8 Conclusion **33**

1 Introduction

Multi-core architectures are nowadays the state of the art in the industry of processor design for desktop and high performance computing. With this design, multiple threads can run simultaneously exploiting a thread level parallelism. Unforunately, achieving better performances is a little bit hard work. Indeed, programmers have to deal with some issues in both software and hardware levels (task scheduling, memory management, shared resources managements, energy consumption and heat dissipation of cores, etc.). Furthermore, the lack in understanding the interactions between the operating system layers, applications and the underlying hardware makes this task even more difficult. A good understanding of these interactions may be exploited in performance evaluation, compiler optimisations and in task scheduling to achieve a better performance stability, reproducibility and predictability.

In this context, applications designers and performance analysts have to iteratively investigate how to achieve the best performances and checking the behaviour of their applications on that architectures. Most often, they consider the execution time as the first metric to investigate in the process of performance evaluation. The execution time is usually observed by measurements, or can be simulated or predicted with a performance model. In our study we consider direct measurements (either by hardware performance counters, or by OS timing functions calls). Contrary to emulated or virtualised programs (such as Javabyte-codes), native program binaries are executed directly on the hardware with possibly some basic OS requests (OS function calls). Our current study focuses on this family of programs: we consider the sample of SPEC 2006 and SPEC OMP2001 [1] benchmark applications. We do not consider binary virtualisation or byte-code emulation because they add software layers influencing the program performance in a more complex way: garbage collector strategies, threads organisation, caching and dynamic compilation techniques all may dramatically influence the measurements of program execution times. Direct measurements of native applications have one software layer (namely the OS) between the user code and the hardware. Unfortunately, the measurement process may also introduce errors or noise (the act of measuring perturbs the program being measured) that can affect our experimental results. For example, there is a time required to read a timer before the code to measure and store the timer after this code. Experimental setups may also introduce other factors that can lead to variation in program execution times. Some of these factors are: Interrupts, starting heap address, starting execution stack address, thread affinity, OS process scheduling policy, background tasks, sharing of the last level of cache and environment size (that have been investigated in [2]). Thus, if we execute a program N times, we may obtain N distinct execution times.

In this document, we introduce some experiments aimed to measure, quantify and analyse the variations of program execution times on an Intel multicore machine. We report measurement results for single-threaded applications (SPEC CPU2006), as well for parallel multi-threaded applications (SPEC OMP2001). The parallel applications use the OpenMP paradigm, one of the most used in parallel programming model on shared memory computers. We show in this study that large SPEC CPU2006 applications have minor variations with the train data input. This of course does not guarantee that the variations of sequential applications would always be negligible especially for small codes (kernels). Unlike single-threaded applications, we show that the variations of execution times of OpenMP applications are really sensitive from a human user point of view.

The report is organised as follows. Section 2 introduces the experimental setup and the methodology that we follow. Sections 3 and 4 present experiments dealing with the variability in program execution times of SPEC CPU2006 and OMP2001 applications. Section 5 studies the impact of co-running applications on the performances variability. Finally, Section 6 studies the effect of binding threads to cores (affinity).

Parameter	Value
Linux Kernel	X86_64 2.6.26
Linux patch	perfmon kernel 2.81
Compiler	gcc 4.1.3, gcc 4-3.2, icc 11.0
Benchmarks	SPEC CPU2006, SPEC OMP2001
Data input	train
Micro-architecture	Intel Core2
CPU Frequency	2.33 GHz
CPU number	2
Core number	8 (2 x 4)
System Memory	4G
Cache L1	32KB Ins, 32KB Data per 1 core
Cache L2	4MB per 2 cores

Table 1: Experimental Setup

2 Experimentation

2.1 Experimental setup

We use an Intel (Dell) server with two `Clovertown` processors. Each processor has 4 cores, while each couple of cores have a shared level 2 cache. Our system has two L2 caches on each chip with 4 MB, for both instructions and data. The core frequency is 2.33 GHz. The maim memory size is 4 GB RAM. The frontside bus has a clock rate of 1.33 GHz. The main features of the test machine are summarised in Table 1 and Figure 1.

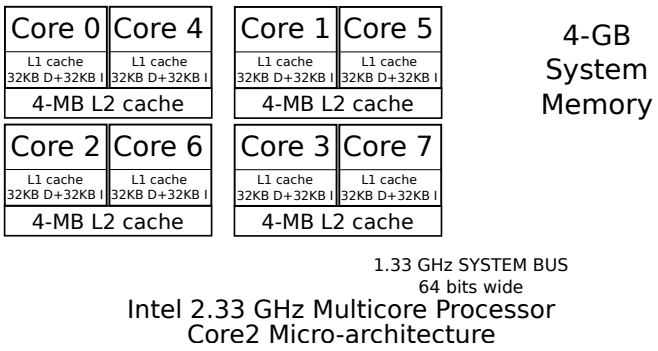


Figure 1: Dual processor architecture

2.2 Experimental methodology

In order to improve the reproducibility of the results, the experiments were done following some practices:

- The test machine was entirely dedicated during the experiments to a single user.
- Running each benchmarks 31 times [3,4] for each software configuration. This high number of runs allows us to report statistics with a high confidence level;
- Unset all the shell environment variables that were inessential;
- The experiments were done on a minimally-loaded machine (disable all inessential OS services except `sshd`);
- Deactivation of the randomisation of the starting address of the stack (this is an option in the Linux kernel versions since 2.6.12);

- Dynamic voltage scaling disabled;
- Using the build system and scripts of SPEC CPU2006 and OMP2001 to compile and optimise the applications, launch them, measure execution times, check validity of the results and report the performance numbers;
- The SPEC system measurement of execution times relies on the `gettimeofday` function;
- The successive executions are performed sequentially in back-to-back way;
- No more than one application was executed at a time, except when we study co-running effects.
- We use violin plot to report the program execution times of the 31 execution of each software configuration. The Violin plot is similar to box plots, except that they also show the probability density of the data at different values. The white dot in each violin gives the `median` and the thick line through the white dot gives the inter-quartile range.

The next section shows that the execution times of long running sequential applications have marginal variability.

3 Variability of SPEC CPU2006 execution times

This section presents the experiments related to the variability of SPEC CPU2006 program execution times. The considered source of variability is the UNIX shell environment size, as studied in [2].

The compiler used is `gcc 4.1.3` with the flags `-O2` and `-O3`. The option `--fno-strict-aliasing` was used for `perlbench` benchmark because of a technical error in that code [1].

The experimentations done on SPEC CPU2006 benchmarks test the relation between the UNIX shell environment size and the variation of programs execution times. Following the methodology explained in [2], we conducted our measurements by varying the size of the Unix shell environment (from 0 to 4095 bytes) and running each benchmark of SPEC CPU2006 31 times with the `-O3` flag optimisation enabled of the `gcc` compiler and 31 times with the `-O2` compiler flag.

Figure 2 shows the execution times of four applications under each Unix shell environment size using violin plot. The leftmost point of the X-axis is for a Unix shell environment size of 0 bytes (the null environment); we generated the data using the `bash` shell and for each point we added 63 bytes to the environment. The width of a violin plot at y-value y is proportional to the number of times we observed y. Figure 2 says that for each UNIX shell environment size in the X-axis, the Y-axis report the 31 execution times.

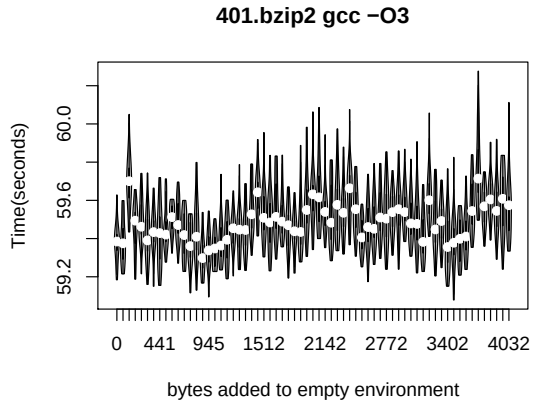
From all these figures we can deduce that: 1) the size of the Unix shell environment may influence the execution times and 2) the variations of execution times are minor and negligible (less than 1%). These observations are valid for all the SPEC CPU2006 benchmarks that we experimented.

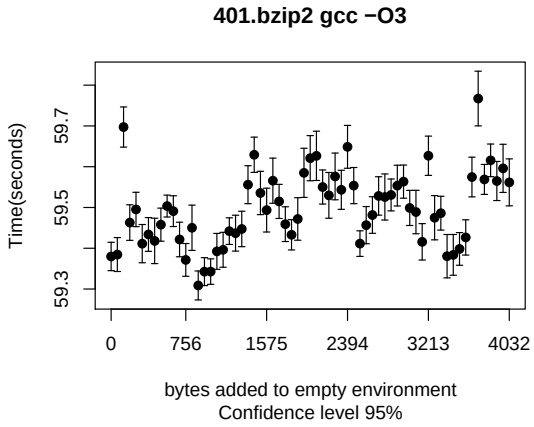
Figure 3 reports the confidence interval of the mean of these benchmarks. We can see that these intervals are sufficiently tights. These figures shows that the sample mean at each Unix shell environment size does not vary in a significant way.

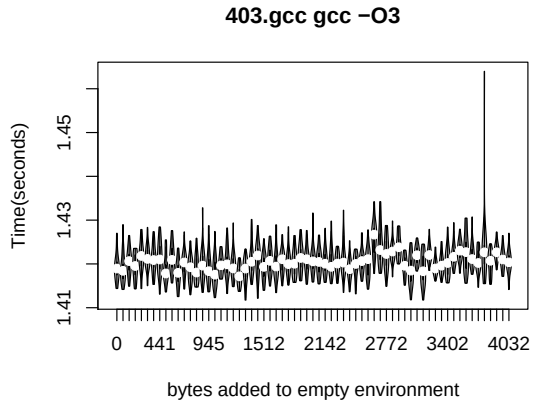
From the experiments presented in this section we deduce that varying the Unix shell environment size has a less impact (less than one second) on the variability of the execution times of SPEC CPU2006 benchmarks whatever the optimisation flag used (`-O2` or `-O3`).

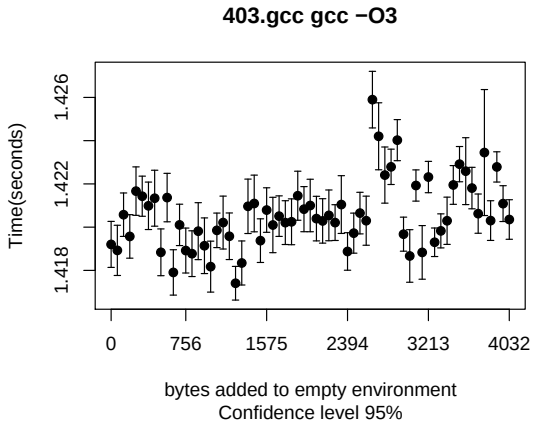
The next section shows the performance variability of the multi-threaded SPEC OMP2001 benchmarks given a fixed experimental setup.

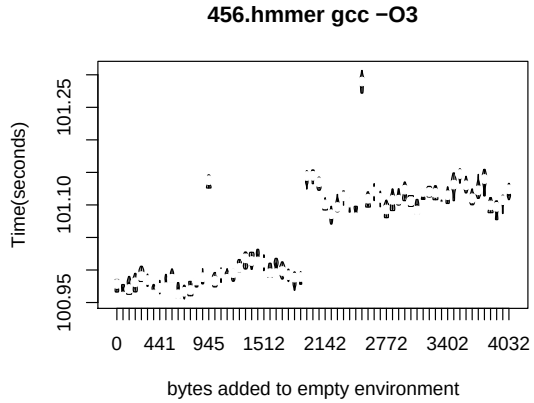
[1] The benchmark has some known aliasing issues. Hence the compilation with high optimisation level will most likely produce binaries assuming strict aliasing. The problem was reported in the SPEC CPU2006 documentation.

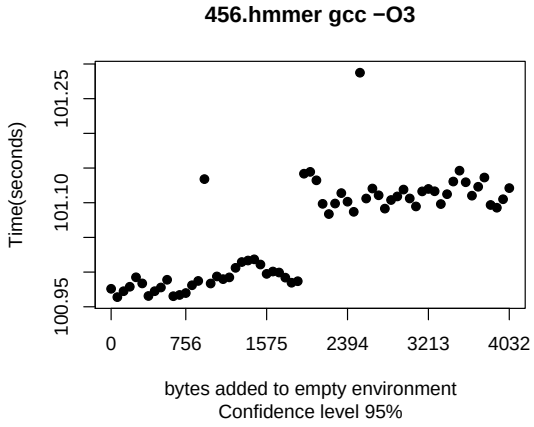


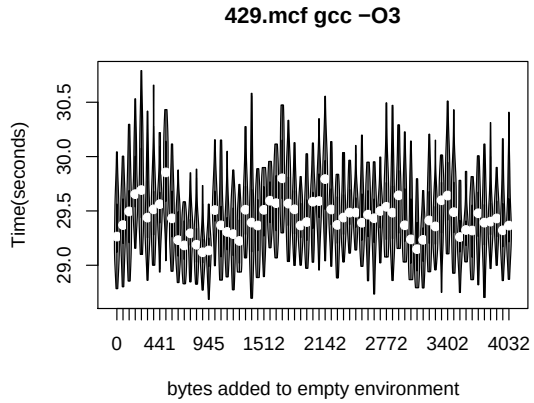












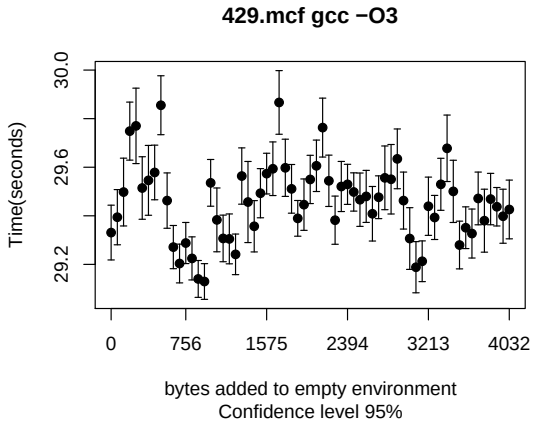


Figure 2: Observed Execution Times of some SPEC CPU 2006 Applications (compiled with `gcc`)

Figure 3: Mean 95% Confidence Interval of some SPEC CPU 2006 Applications (compiled with `gcc`)

4 Variability of SPEC OMP2001 execution times

This section presents experiments related to the variability of program execution times in SPEC OMP2001. The targeted benchmarks are parallel programs written with the OpenMP API. The aim of these experiments is to study two things: 1) Are the parallel execution of the benchmarks with different number of threads lead to variability in program execution times ? 2) Compare the benefit of the parallel execution with different number of threads against the sequential version.

The compilers used are `gcc 4.3.2` and `icc 11.0`. For each application of SPEC OMP2001 benchmarks, we generated two compiled binary codes. The first one is generated by setting only the `-O3` compilation flag (single-threaded version). The second one (multi-threaded version), is generated by setting `-O3 -fopenmp` and `-O3 -openmp` compilation flags respectively for the `gcc` and `icc` compilers. `gcc` was not able to compile the OpenMP version of `mgrid_m` because of a bug (Bugzilla Bug 33904). The parallel execution of `gafort_m` failed because of a segmentation fault (this execution error was also reported if we use the Intel `icc` compiler). The Unix shell environment size was fixed for each software configuration. In the case of the multi-threaded version, we consider 5 configurations. Each configuration sets up the number of thread to be generated at runtime. Typically, this is achieved by setting the environment variable OMP_NUM_THREADS respectively to the values 1, 2, 4, 6 and 8 threads. We limited OMP_NUM_THREADS to 8, because our experimental machine have a maximum number of cores equal to 8. Running a parallel version of each benchmark with only one thread may be surprising, but the goal of such approach is to compare the performances of the two configurations; (sequential code with `-O3` and parallel code running with a single thread).

Figure 4 shows the violin plots of program execution times for four applications from SPEC OMP2001 benchmarks compiled using the `gcc` compiler. These applications are selected because they highlight significant performance variability. The X-axis represents the different software configurations for the application: sequential version (no threads), OMP version with 1 thread, 2 threads, 4 and 8 threads. The Y-axis represents the 31 observed execution times for each software configuration. We conclude the following observations:

1. The sequential and the single threaded versions do not exhibit significant variability.

2. When we use thread level parallelism (2 or more threads), the execution times decreases in overall but with a significant disparity. Consider for instance the case of `swim` in Figure 4. The version with 2 threads runs between 76 and 109 s, the version with 4 threads runs between 71 and 90 s. This variability is also present when `swim` is compiled with `icc`, see Figure 5. The example of `wupwise` in Fig. 4 is also interesting. The version with 2 threads runs between 376 and 408 s, the version with 6 threads runs between 187 and 204 s. This disparity between the distinct execution times of the same program with the same data input cannot be justified by *accidents* or experimental hazards. Applying the Shapiro-Wilk normality check on performance data we concluded that the execution times are not normally distributed, and frequently have a bias.

3. The case of the application `galgel` is also interesting. In addition to the variability of the execution times for each software configuration, we observe that the performance of the program substantially decreases when increasing the number of threads! This examples illustrates that, on a multicore architecture, increasing thread parallelism may bring severe performance loss. We checked the situation of `galgel` when we use the Intel `icc 11.0` compiler instead of `gcc`, and the situation was radically different, see Figure 5: increasing the number of threads decreases the execution times. We can observe a huge difference between the performance of the program compiled with gcc vs. the icc, either in terms of execution times and in terms of variability. We have to notice that using the `gcc-4.4.3` version of the `GNU` compiler has effectively reduced the execution times when we increase the number of threads (see Figure 6). This situation illustrate that the quality of the code generated by a compiler has a significant impact on performance stability.

4. The `galgel` application compiled with the `gcc` compiler, shows that speedup computation is not fair if we consider the minor execution time. We can see from the Figure 4 that the violin plots of the second (PAR (1TH)) and third (PAR(2TH)) configurations gives an interesting result on

how we have to summarise the performance data of one configuration to single number. If we use the *min* function to summarise data of the two configurations, then, we can say that the third configuration is better than the second one. But if we take the *median* function to summarise these data we may conclude that the two configurations are similar. We note that the choice of which function to use to define the execution time is crucial and may lead to misleading conclusions about the real behaviour of the system.

5. Figure 7 shows that the sequential version of `ammp` is better than his parallel version when parallelisation is achieved with: 1 thread by about 25%, and 2 threads by about 15%. The case of `ammp` shows that the OpenMP API does not necessarily produce faster codes against the sequential version. The reason is that the compiler makes better optimisations when OpenMP is not enabled.

5 Study of co-running processes impact on programs performance variability

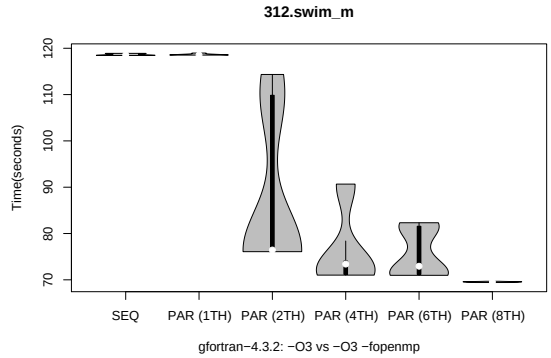
This section describes experiments trying to quantify and qualify the factors influencing the variations of program execution times. One of the factors which can influence the variability of program execution times is the core resource sharing. In our study, we focus on the sharing between the OpenMP parallel programs and some artificial applications. For each OpenMP benchmark we retrieve its execution times in user mode (run level 3 : least privileged mode), system mode (run level 0 : most privileged mode) and real execution time (total elapsed execution time).

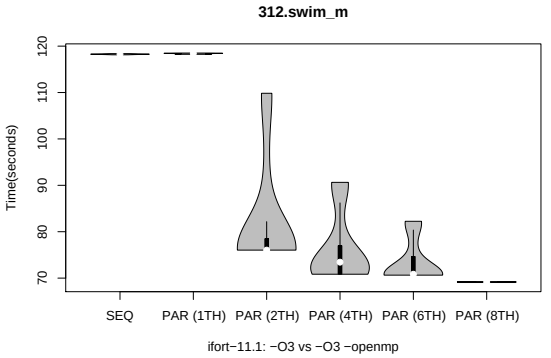
In this experiments we generated a system load by running some artificial co-running processes in background. These processes are launched by one process executing the `fork` system call a number of times equal to the number of processes that we need to generate at runtime. This number is supplied as an argument to the command line. The code executed by these co-running processes is a dummy loop with a reduced memory cache access. The code of the co-running processes is given by Listing 3 in page 15.

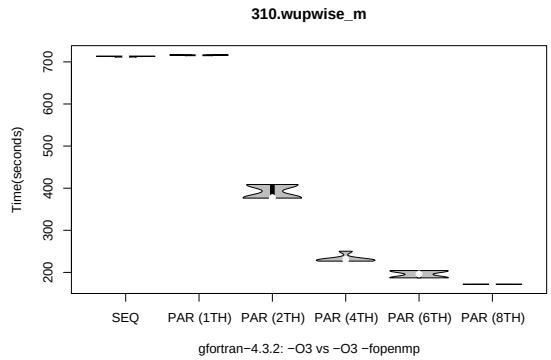
5.1 Experimental setup

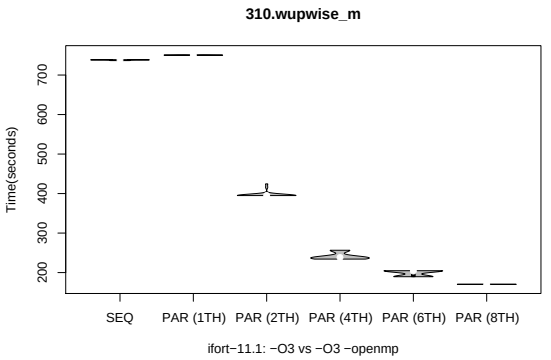
- Each benchmark was compiled with the `-O3 -fopenmp` flags.
- The program execution time measurement are only done on process (thread) 0 of each application (we retrieve the execution time of the master thread because it is the one that defines the whole application execution time).
- The SPEC OMP2001 benchmarks are launched with 8 threads at runtime. Thus, we assign each thread to an available core.
- We have five runtime configurations:
  1. Running each SPEC OMP2001 benchmark (8 threads) as a single application on the machine (minimal system load).
  2. Running each SPEC OMP2001 benchmark (8 threads) with 8, 16, 24 and 32 co-running processes respectively (performance perturbation created in background).
- The SPEC OMP2001 and the co-running processes are launched without scheduling affinity to the system cores (no explicit binding of threads on cores).
- The number of the OpenMP threads (from applications under study) and co-running processes running on each core are respectively : 1, 2, 3, 4 and 5. For example a configuration with 5 threads or co-running process per core, consist of 1 OpenMP thread plus 4 co-running processes.

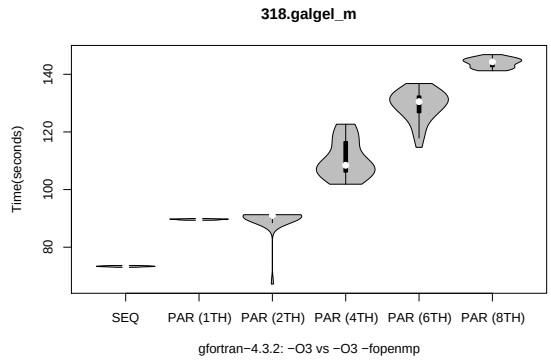
Figure 11 and Figure 12 show the violin plots of the user and real program execution times for four applications from SPEC OMP2001 benchmarks compiled using the `gcc-4.3.2` compiler. The X-axis represents the violin plots of program execution times when the SPEC OMP2001 benchmarks run

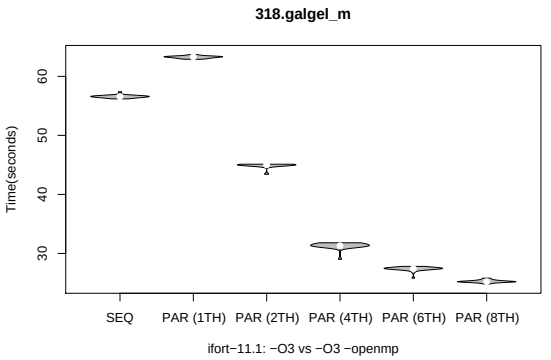


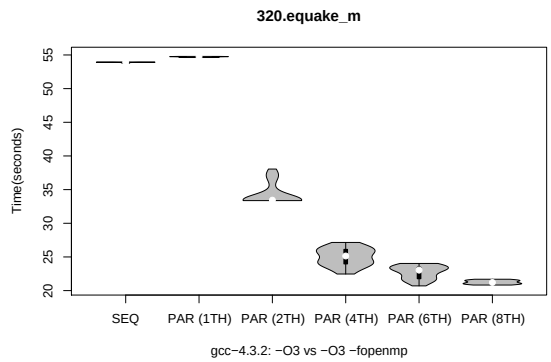












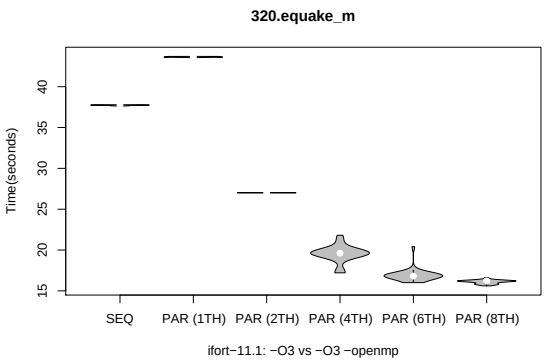


Figure 4: Observed Execution Times of some SPEC OMP 2001 Applications (compiled with `gcc`)

Figure 5: Observed Execution Times of some SPEC OMP 2001 Applications (compiled with `icc`)

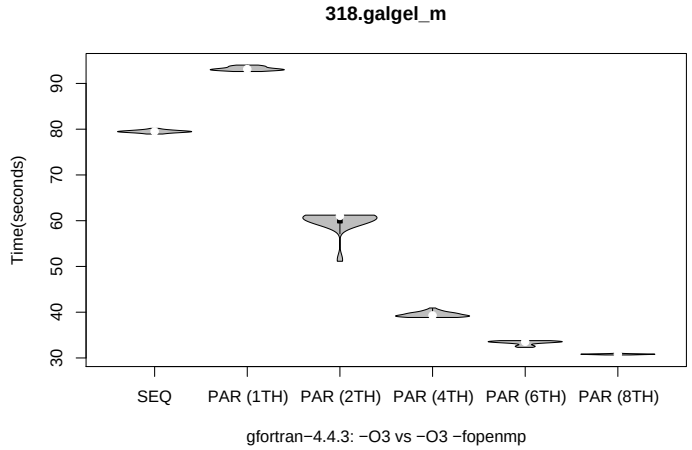


Figure 6: Observed Execution Times of the `galgel` benchmark compiled with `gcc-4.4.3`

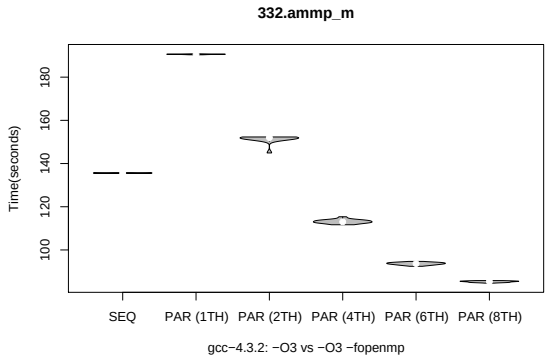


Figure 7: Observed Execution Times of `ammp_m` benchmark (compiled with `gcc`)

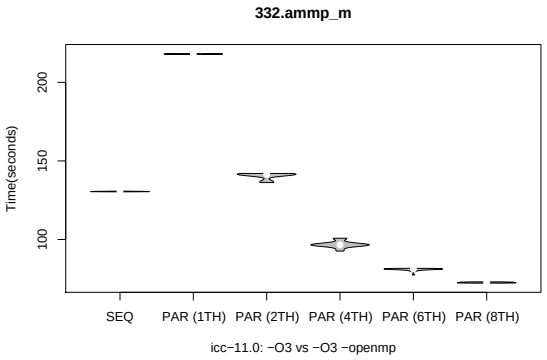


Figure 8: Observed Execution Times of `ammp_m` benchmark (compiled with `icc`)

together with the co-running processes (as explained above). The Y-axis represents the 31 observed execution times for each software configuration.

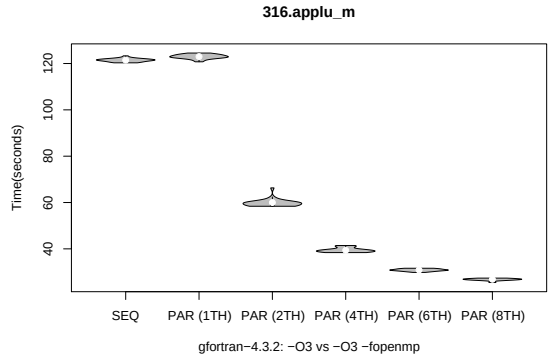
From the Figure 11, we can see that when we increase the number of processes running in background, the program execution times at user level of the OpenMP applications decreases while we observe in Figure 12 the increases of real execution time. Running the threads of the SPEC OMP2001 benchmarks and the co-running processes with a scheduling affinity leads to the same conclusion: when we increase the number of co-running processes, we observe a decrease in program execution times at user level of the OpenMP applications. The following section presents measurements to check if the phenomenon presented above appears when running some basic micro-benchmarks instead of SPEC OMP2001.

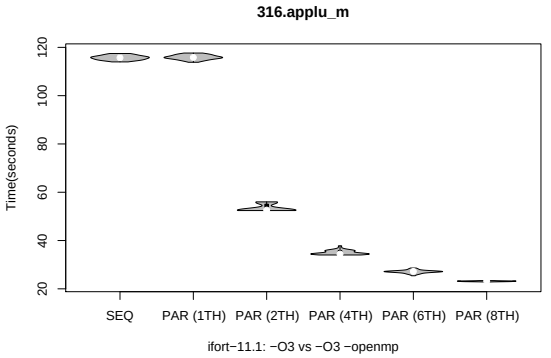
5.2 Study of system load impact on micro-benchmarks execution times

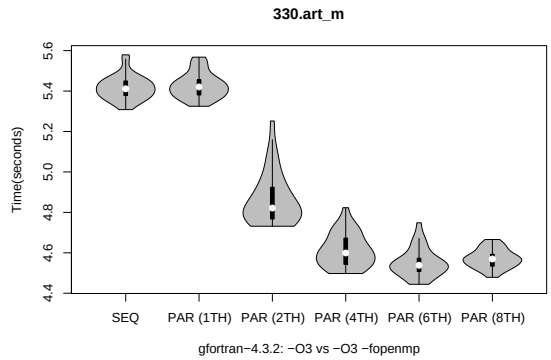
We follow the same steps presented in the previous section, except this time we use some synthetic (simple algorithms) benchmarks which we call micro-benchmarks. These micro-benchmarks are used to isolate some micro-architectural events. We have to notice that all of the micro-benchmarks and the co-running processes were compiled with the compiler optimisation flag `-O3`.

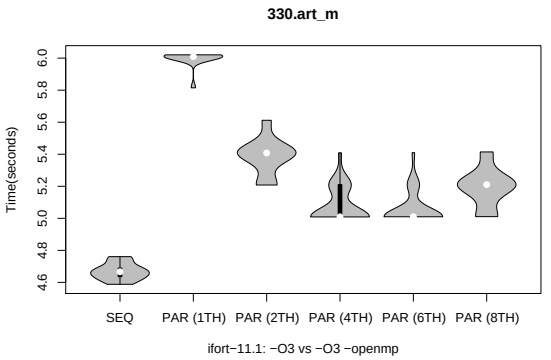
5.2.1 Micro-benchmarks code

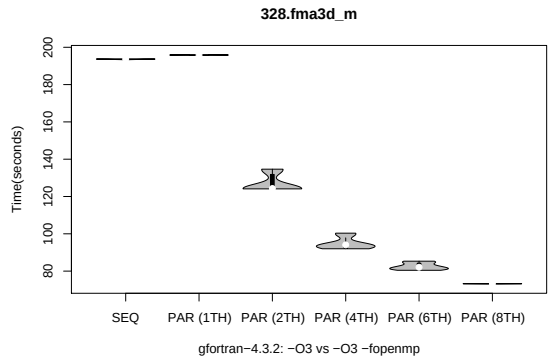
The code of the micro-benchmarks in Listing 1 (page 14) is composed of three loops `Loop1` (parallel loop), `Loop2`, `Loop3` and one statement `S1`. We have to notice that the `L2` loop was added for repetition

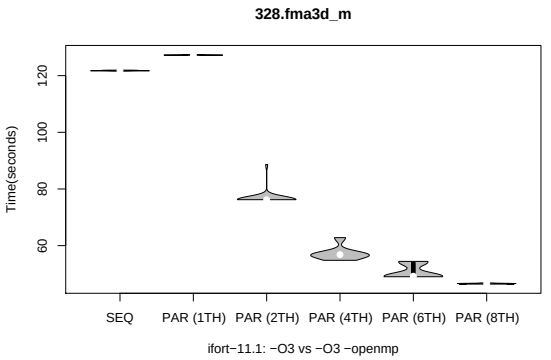


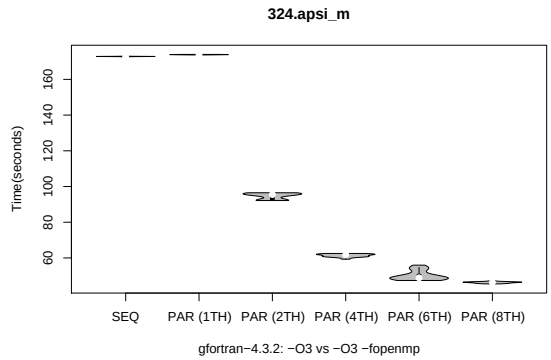












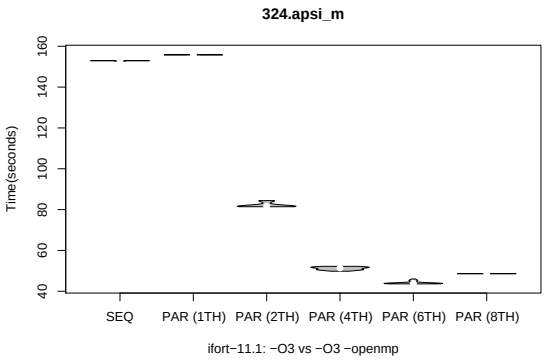
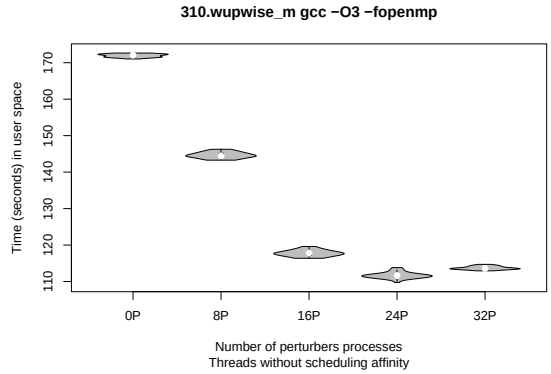
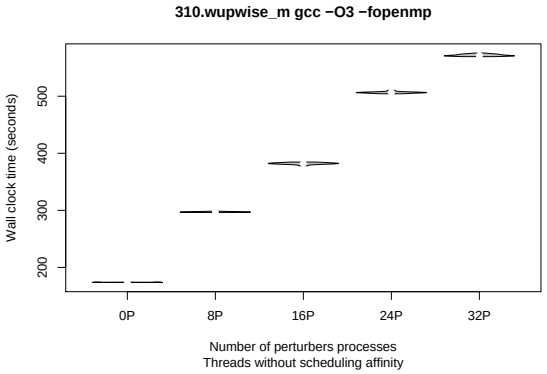
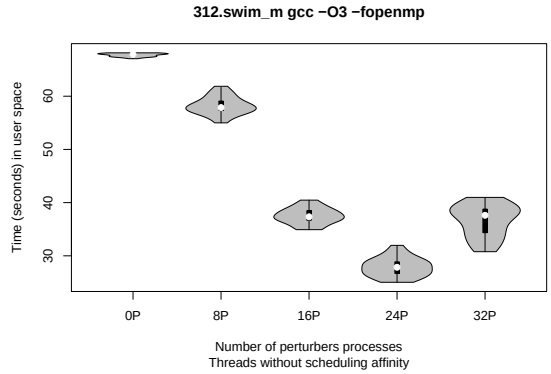


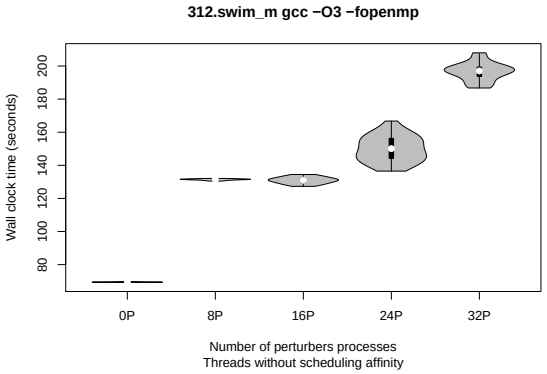
Figure 9: Observed Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

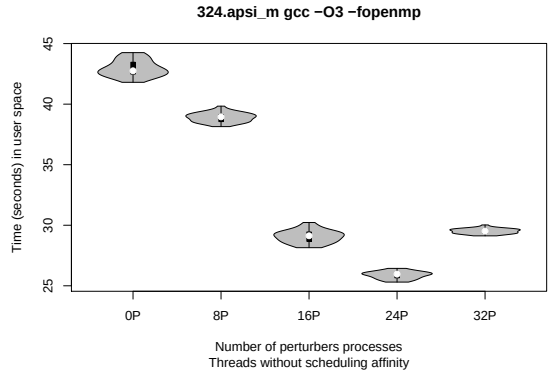
Figure 10: Observed Execution Times of some SPEC OMP2001 Applications (compiled with `icc`)

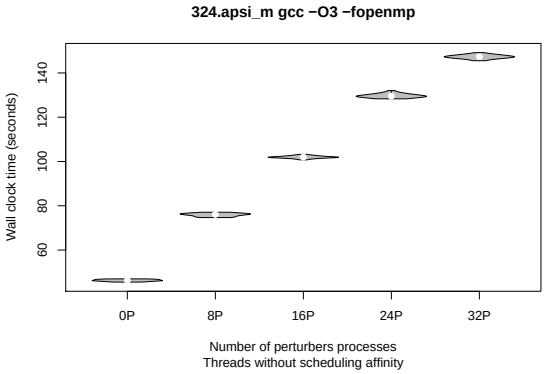


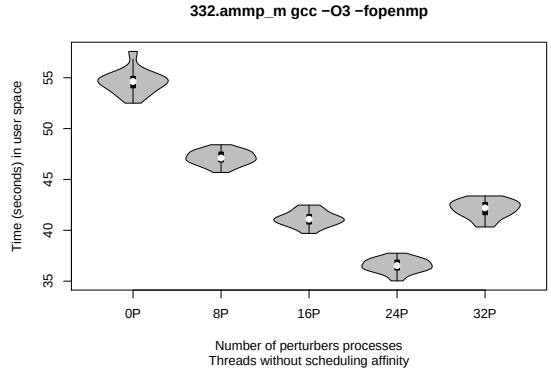












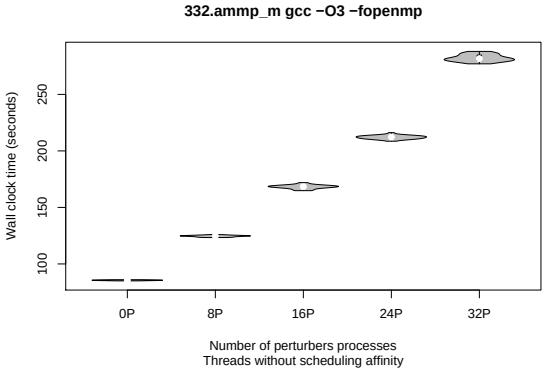


Figure 11: Observed User Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

Figure 12: Observed Real Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

N	M	Benchmark	Chunk size = M*sizeof(int64)	N	M	Benchmark	Chunk size = M*sizeof(int64)
8	196608	mb_8_196608	1536 KB	1536	1024	mb_1536_1024	8 KB
16	98304	mb_16_98304	768 KB	3072	512	mb_3072_512	4 KB
32	49152	mb_32_49152	384 KB	6144	256	mb_6144_256	2 KB
64	24576	mb_64_24576	192 KB	12288	128	mb_12288_128	1 KB
128	12288	mb_128_12288	96 KB	24576	64	mb_24576_64	0.5 KB
256	6144	mb_256_6144	48 KB	49152	32	mb_49152_32	0.25 KB
512	3072	mb_512_3072	24 KB	98304	16	mb_98304_16	0.125 KB
1024	1536	mb_1024_1536	12 KB	196608	8	mb_196608_8	0.0625 KB

Table 2: Values taken by N and M in the outermost and the innermost loops of th micro-benchmarks code

purpose to increase the measurement accuracy. The data set accessed by all the micro-benchmarks is equal to $N * M * sizeof(int64) = D$ *bytes* where N is the number of iterations of the outermost loop (`Loop1` loop) and M is the number of iterations of the inner most loop (`Loop3` loop). In addition, since we want to give the same workload to every thread, this leads to consider values for N which are multiple of the number of threads. Having all this constraints, the values taken by N are from 8 to 196608 and the values taken by M are from 196608 to 8. For instance, when we have 8 threads, the value of N starts at 8. Furthermore, whatever the values of N and M are, the workload assigned to each thread has a working set of size $1.5MB$. This size is chosen to be less than the half of the size of the cache L2 preventing from frequently accessing the DRAM in case of L2 cache misses.

Now we define the notion of memory *chunk*. In the context of our study the *chunk* represents the size of the vector fraction from `tab` accessed by the innermost `M`-loop: in the peticular code of Listing 1, we clearly see that each iteration of the `M`-loop accesses to a single element from `tab`, consequently *chunk size* = $M \times sizeof(int64)$. Table 2 gives the couples of values of N and M that we have experimented. To every couple of values (N,M) we associate the micro-benchmark mb_N-value_M-value. Following this denomination, the first micro-benchmark is mb_8_196608, the second mb_16_98304 and so on. With this micro-benchmark structure, we access the array `tab` in an indexed way represented by the `S1` statement. Thus, the size of the chunk accessed in the `M-loop` depends on the number of iterations of the `Loop1`. So, Larger values of N lead to smaller sizes of chunks. In contrary, smaller values of N lead to larger sizes of chunks.

Listing 1: OpenMP micro-benchmarks code

```
void wastetime() {
  #pragma omp parallel for default(none) private(i,j,k) shared(tab)
Loop1:   for(i = 0 ; i < N ; i++)
Loop2:         for(j = 0 ; j < 10000; j++)
Loop3:               for(k = 0 ;  k < M; k++)
S1:                    tab[i*M+k]++;
}
```

The first micro-benchmark mb_8_196608 corresponds to the case where every thread executes a single outer loop iteration (`Loop1` or the `N-loop`) and the innermost loop `M-loop` accesses to a chunk of size $M * sizeof(int64) = 1.5\ MB$. Thus, chuck size is sufficient to keep data inside the L2 cache but not inside the L1 data cache. In other words, the first micro-benchmark guarantees that every thread has its data in L2 but not in L1.

The last micro-benchmark mb_196608_8 corresponds to the case where every thread executes $N/8 = 24576$ `N` iterations, the innermost `M`-loops access to a chunk of size $M * sizeof(long) = 64\ B$ (a single cache line size). In other words, this last micro-benchmark guarantees that every thread has all its data

in L1.

The other micro-benchmarks between mb_8_196608 and mb_196608_8 cover the range for other values of (N,M). They give us the performance of the intermediate situations when data are fully or partly in L1. We should have (in theory) all data fully inside L2 because the chunk sizes are all less than half of L2 size, but we see later that threads sharing common L2 may create cache conflicts, thus data are ejected from L2.

Listing 2 shows the code of the co-running processes executing in parallel with Listing 1. We can see that it is a simple code consisting of a dummy `while` loop. We have to notice that the code showed below represents only the function that takes the most of the CPU cycles in the co-running processes. The `stress` function is called by the `main` function and it never returns unless there is an explicit user intervention by sending the `Linux SIGINT` signal to stop it.

Listing 2: co-running processes code

```
int stress () {
    while (1) {
        ;
    }
    return 0;
}
```

The generated assembly code after compiling the C code of the co-running processes with the `-O3` compilation flag is given in listing 3.

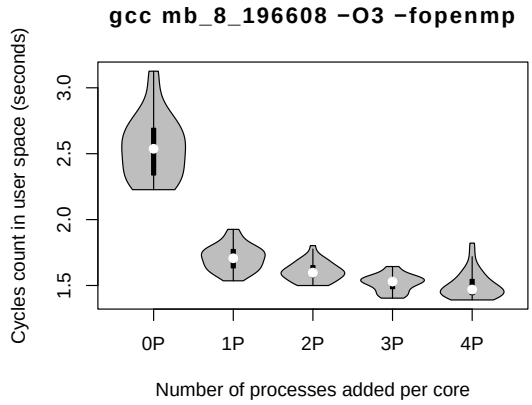
Listing 3: The generated assembly code of the function *stress* in the co-running process code

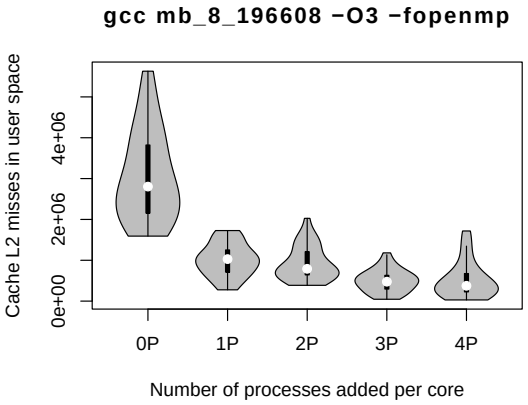
```
.label:
    jmp .label
```

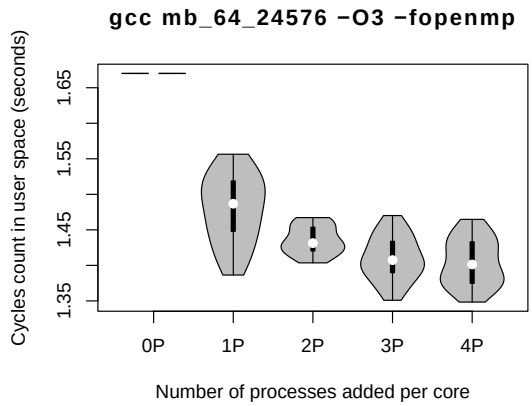
5.2.2 Micro-benchmarks execution times under co-running tasks

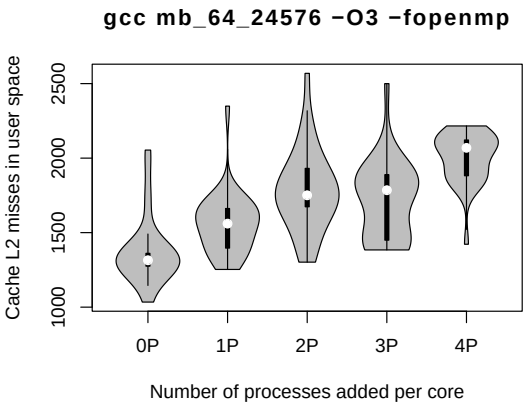
From Figure 13, we observe that when we increase the number of co-running processes (X-axis), we a observe a decrease of program execution times at user level. This observation is done with the mb_8_196608, mb_64_4576 and mb_256_6144 micro-benchmarks, but a slight increase of user level execution time is observed in the mb_512_3072 micro-benchmark. Furthermore, we observed that all the micro-benchmarks which have a N value in the outermost loop (the i loop) greater or equal than 512 (chunk sizes less than 32KB), have a slight increase (can be neglected) in program execution times at the user level. At the other side, all the micro-benchmarks which have a N value less than 512 (chunk sizes greater than 32 KB), have a noticeable decrease of program execution time at the user level.

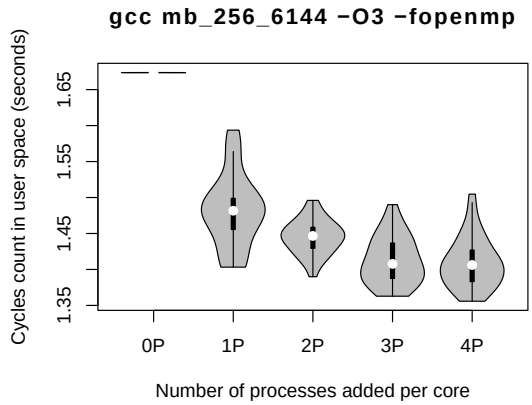
The common point between the micro-benchmarks where the program execution times decrease is the size of the chunk that each thread treats. Indeed, each thread from those benchmarks treats a chuck which have a size greater than 32 KB. The size of those chunks fits in the L2 cache but it does not fit in L1 data cache leading to frequently access the last level cache. In addition, the benchmarks presenting a decrease in program execution times exhibits a high number of L2 cache misses. What we have observed is this phenomenon does not appear in the mb_512_3072 micro-benchmark. For each iteration of the i loop in this micro-benchmark, a 24KB data size are provided to the L1 data cache which have a 32KB size.

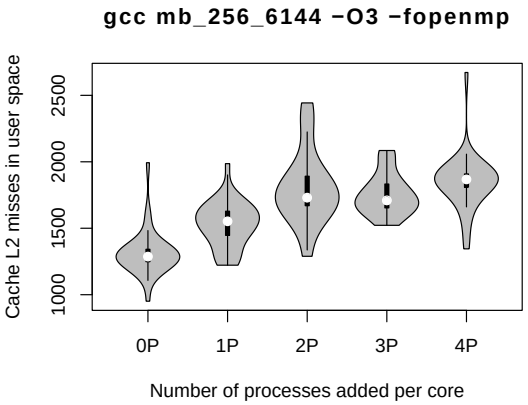


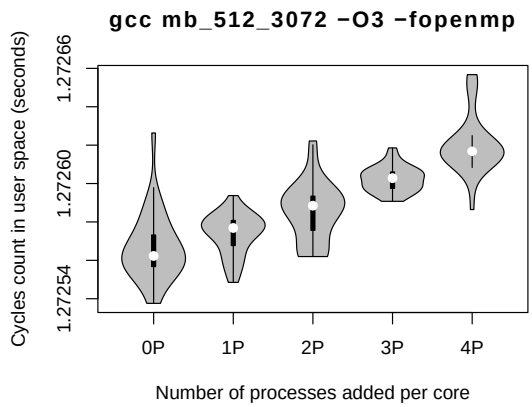












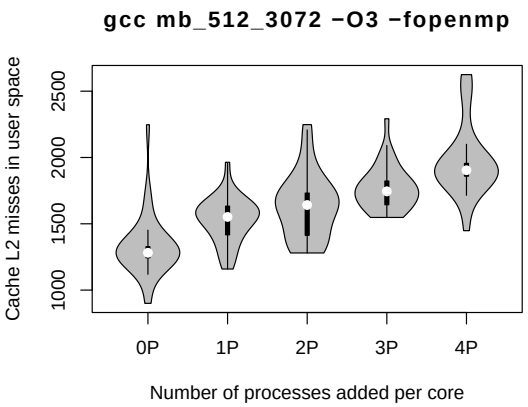


Figure 13: Observed User Execution Times of some micro-benchmarks (compiled with `gcc`)

Figure 14: Observed cache L2 misses of some micro-benchmarks (compiled with `gcc`)

Another remark is that the mb_512_3072 micro-benchmark has the least program execution time compared to the other micro-benchmarks. Even if there are some micro-benchmarks which have program execution times decrease, mb_512_3072 still have the best observed execution time.

Now let try to understand the performances of Figure 13 by plotting the L2 cache miss rate in Figure 14:

1. We have eight threads running on eight cores and four L2 caches, each L2 is shared between two cores (see the architecture pf the machine in Figure 1 in page 5). Then we have two threads running on two adjacent cores meaning that they are competing to access a shared L2 cache. The decrease of execution times was observed on micro-benchmarks which access chunks having a size greater than the size of the L1 data cache. Thus, this subset of micro-benchmarks needs a high frequency access of the L2 cache to provide the needed data by the L1 data cache leading to a high number of L1 data cache misses.

2. The high number of L2 cache misses observed can be due to the hardware prefetcher of the L2 cache. We think that the automatic hardware prefetcher evict some useful data from L2. The main effect of prior behaviour, is for each iteration of `Loop2` (the j loop), data must be reloaded to the L2 cache leading to the high number of L2 cache misses. Another explanation for the high number of L2 cache misses is due to using small virtual memory pages. In this case several 4 KB pages are mapped to the block of L2 cache lines, resulting in conflict misses.

From the foregoing facts, we emit another hypothesis trying to explain why user-level program execution times decrease when we increase the number some co-running processes. The hypothesis is based on the influence of the thread scheduling policy (Round-Robin time sharing) on the execution times. We think that the threads running on cores sharing data in L2 will not access them simultaneously: for each couple of cores sharing an L2 cache, the scheduling of the thread may lead to situations where one core will run an OpenMP thread and the adjacent core will run the co-running process resulting in none simultaneously access to the shared L2.

Therefore, the number of L2 cache misses decreases, reducing the pressure on the shared L2 cache. Finally, this situation leads to decrease the program execution times at the user level. The following subsection presents experiments to check the accuracy of our hypothesis.

5.2.3 Affinity impact on micro-benchmarks execution times

The aim of these experiments is to study the role played by the co-running processes on the decrease of program execution times at user level. To achieve this goal, we conducted our experiments following the experimental setup presented in Section 5.1 with some modifications at the runtime level.

- The micro-benchmarks are launched with 4 threads at runtime.
- We have five runtime configurations:
  1. Running each micro-benchmark (4 threads) as a single application on the machine (minimal system load).
  2. Running each micro-benchmark (4 threads) with 4, 8, 12 and 16 co-running processes respectively.
- We have also two scheduling affinity configurations:
  1. The micro-benchmarks and the co-running processes are launched on the system cores 0, 1, 4 and 5, where cores 0 and 4 share one L2 cache, similarly cores 1 and 5 share another L2 cache (see Figure. 1 in page 5).
  2. The micro-benchmarks and the co-running processes are launched on the system cores 0, 1, 2 and 3, where there is no sharing of L2 cache between these cores.

- The number of co-running processes and OpenMP threads (micro-benchmarks threads) running on each core are respectively : 1, 2, 3, 4 and 5.

The main difference of this experiment against the last one presented in Section 5.2.2, appears in the way the threads of the benchmarks run. You should notice that in this experiment, the number of threads of the micro-benchmarks used at runtime is four. These threads are launched with a scheduling affinity to cores in two manners: 1) configuration where there are share of the L2 cache between the threads, and 2) configuration where there are no L2 cache sharing. To allow a comparison between the actual experiment and the last one (Section 5.2.2), threads must treat the same data size. To achieve this goal, we changed the N and M values of Listing 1 to run the micro-benchmarks with four threads. Thus, we keep the same data size accessed by each thread as in the case where micro-benchmarks were launched with eight threads. The new values taken by N are from 4 to 196608 and from 196608 to 4 for the M parameter in the micro-benchmarks code.

Figure 15 and Figure 17 give us the results when the micro-benchmarks threads and the co-running processes are binded to system cores 0, 1, 4 and 5 and to system cores 0, 1, 2 and 3 respectively. From Figure 15 we can see that program execution times at user level still decrease in the mb_4_196608, mb_16_24576 and mb_128_x6144 micro-benchmarks. We can also observe in Figure 15 that the execution times of mb_256_3072 increase. We clearly observed the phenomena of program execution times decreasing when 1) each couple of the micro-benchmarks threads share an L2 cache and 2) the sum of the array element sizes accessed by all the iterations of the innermost loop (the k loop) is greater than the size of the L1 which is not the case for the mb_256_3072 micro-benchmarks (all the iterations of innermost loop access a contiguous memory with a size less than the L1 cache line).

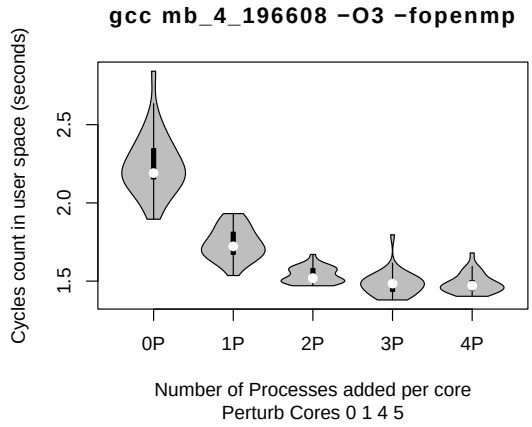
Figure 17 shows results where there are not sharing of cache L2 between threads. The results are surprising, because except for mb_4_196608 that has a negligible decrease in execution time, all other micro-benchmarks show a slight increase. We must remember that mb_256_3072 and mb_512_3072 micro-benchmarks have the same workload per thread, the difference between them is that the former does the computation with four threads, where the latter does it with eight threads. The results presented by Figure 15 and Figure 17 show that the scheduling affinity has an impact on how the parallel programs (OpenMP threads) interacts with other tasks (the co-running processes). It confirms our intuition that when application thread is executed in parallel with a co-running process, then it is beneficial for user level execution time.

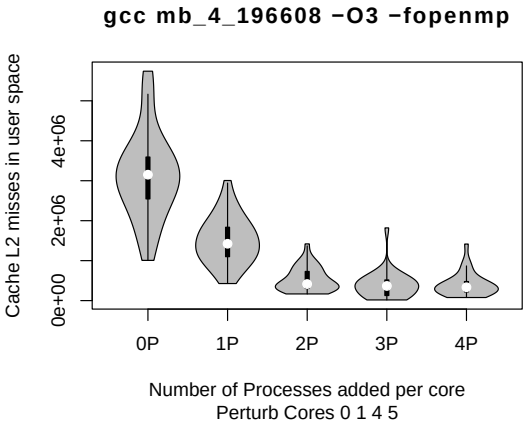
The next section explores two other factors that may influence the decreases of user level program execution times. We focus on the impact of using huge pages and the impact of disabling the hardware prefetcher on the performance stability.

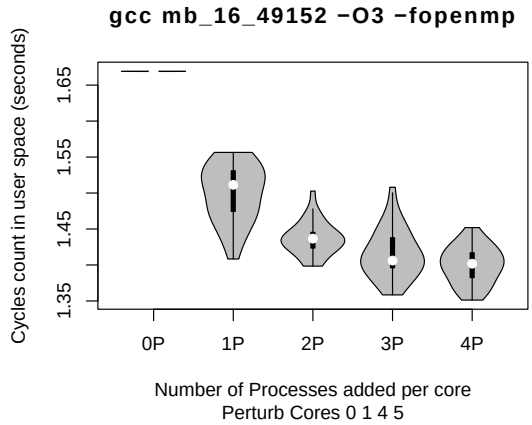
5.3 Memory page size and hardware prefetcher impact on performance variability

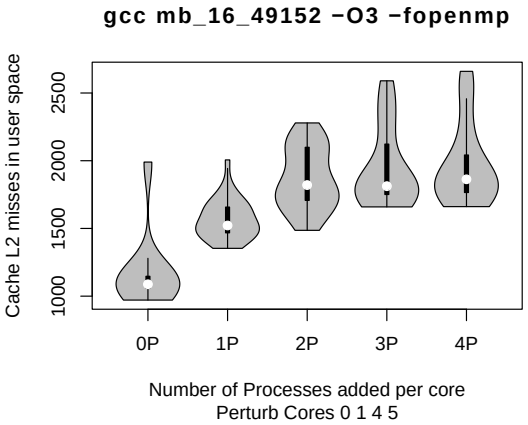
We have shown that the co-running processes plays a role in decreasing the execution times at user level for some benchmarks by smoothing the access to L2. A question arises: what is the origin of the high number of cache misses, while the working set of these micro-benchmarks fit in the L2. For this reason we perform some experiments to check the accuracy of our intuition (the high number of cache L2 misses is due either to the hardware prefetcher or to using small pages or to the combined effect of the two). The results of the experiment when the hardware prefetcher is disabled (see Figure 19) have shown that running the micro-benchmark mb_4_196608 on the cores 0, 1, 4 and 5 has not affected the decreasing of program execution times in user level when we add co-running processes. This because the micro-benchmark still exhibits a high number of cache L2 misses (see Figure 20). The decrease in (`median`) user execution time is about 31% which is equivalent for the configuration where the micro-benchmark runs with hardware prefetcher enabled and using small pages (see Figure 15).

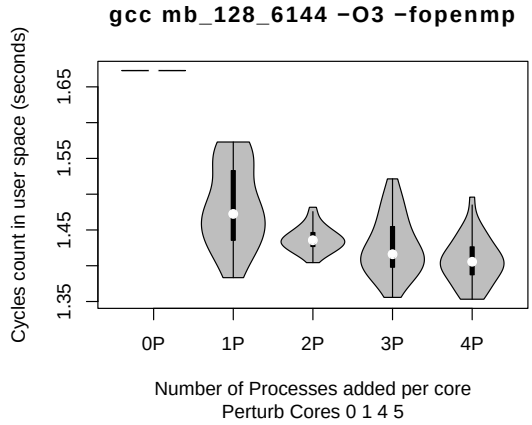
Figure 21 shows program execution times at user level for the mb_4_196608 micro-benchmark running on the 0, 1, 4 and 5 cores with large pages. We can see that using large pages has a significant impact on the performances of mb_4_196608. Indeed, we observe a 25% performance improvement compared to the

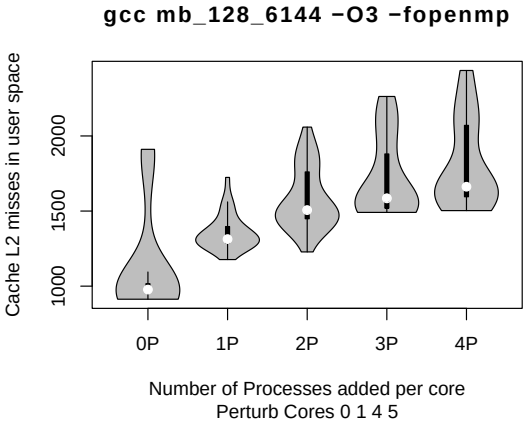


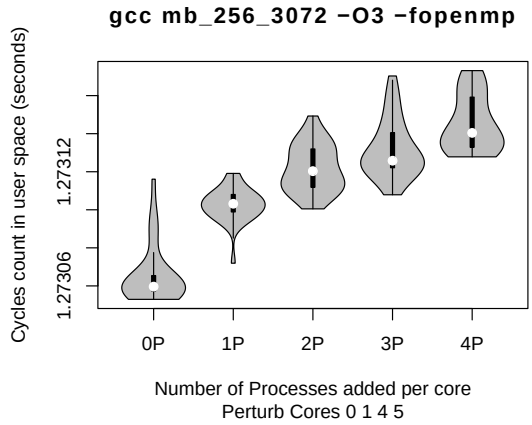












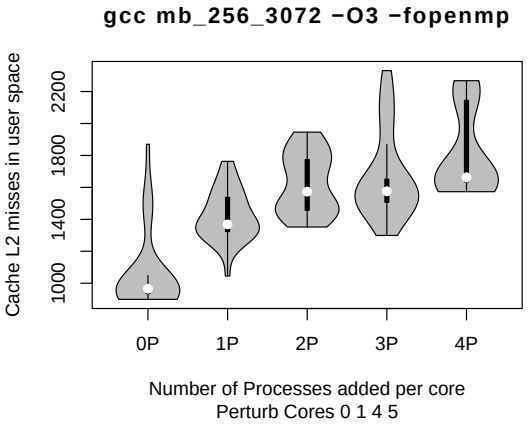
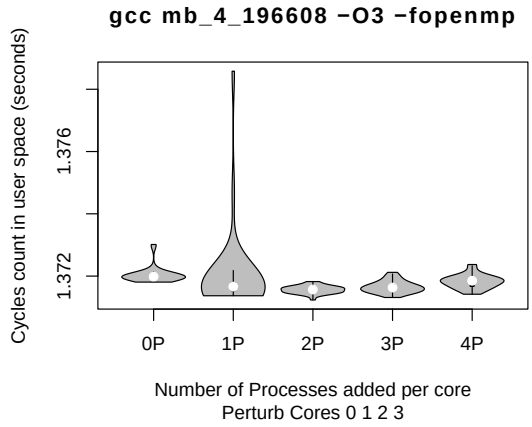
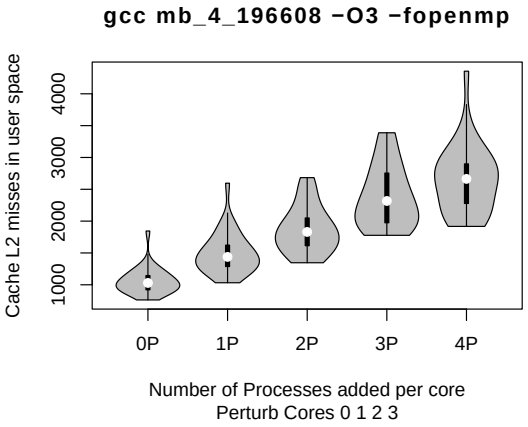
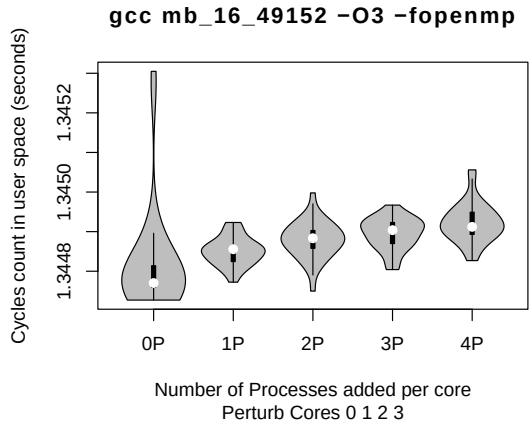


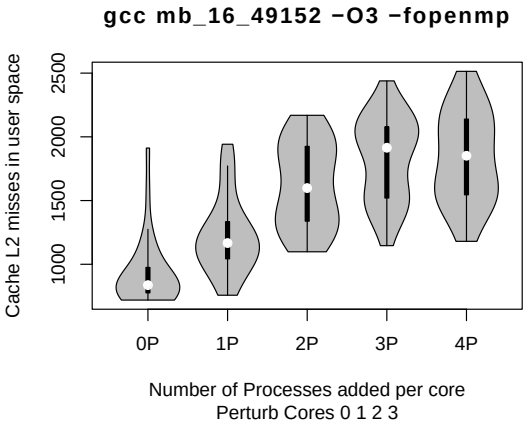
Figure 15: Observed User Execution Times of some micro-benchmarks binded to the 0, 1, 4 and 5 cores

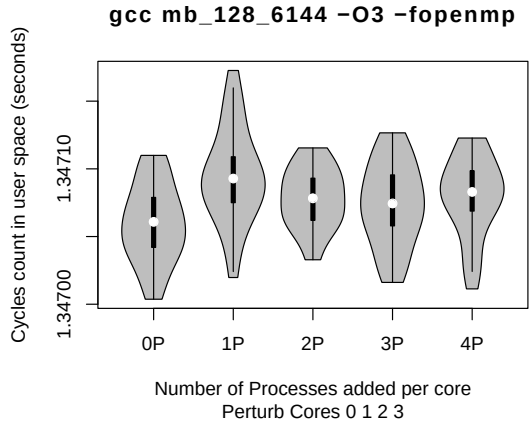
Figure 16: Observed cache L2 misses of some micro-benchmarks binded to the 0, 1, 4 and 5 cores

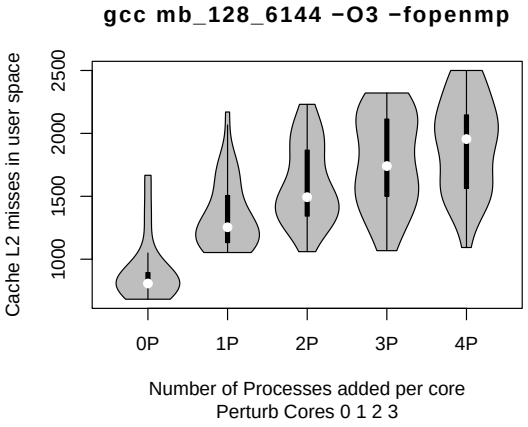


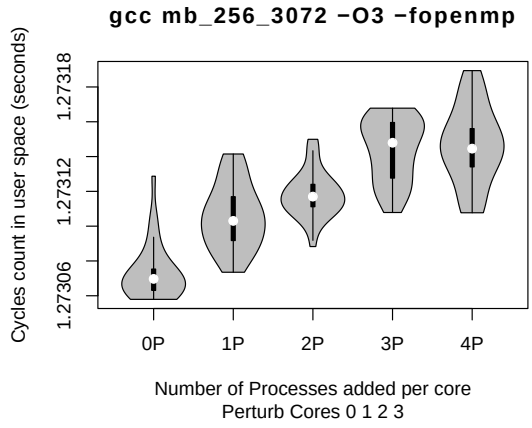












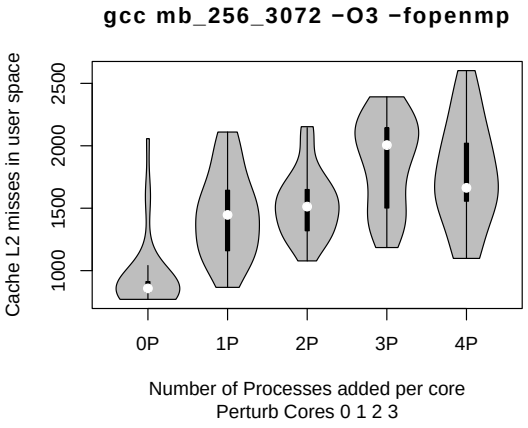


Figure 17: Observed User Execution Times of some micro-benchmarks binded to the 0, 1, 2 and 3 cores

Figure 18: Observed cache L2 misses of some micro-benchmarks binded to the 0, 1, 2 and 3 cores

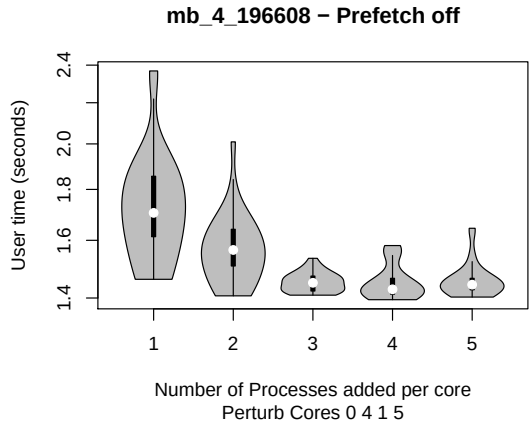


Figure 19: Observed user execution times for mb_4_196608 running on the 0, 1, 4 and 5 cores with hardware preftcher disabled

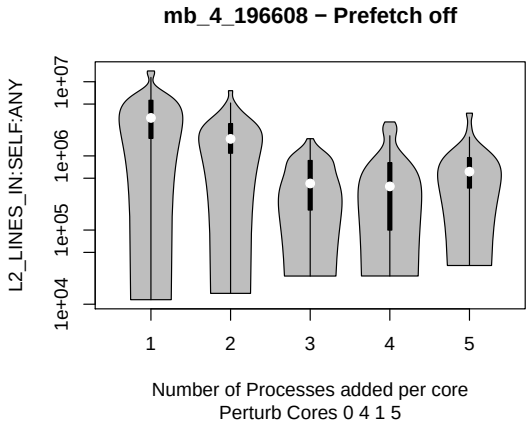


Figure 20: Observed L2 cache misses for mb_4_196608 running on the 0, 1, 4 and 5 cores with hardware preftcher disabled

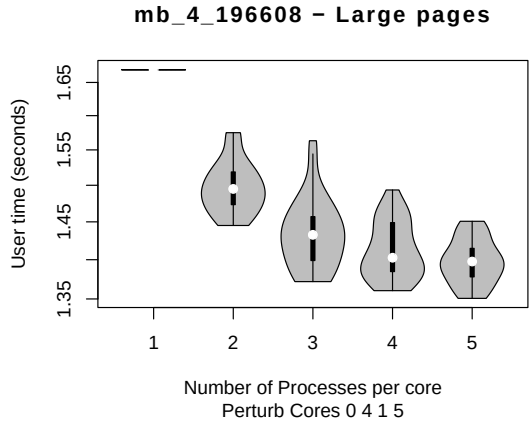


Figure 21: Observed user execution times for mb_4_196608 running on the 0, 1, 4 and 5 cores with large pages

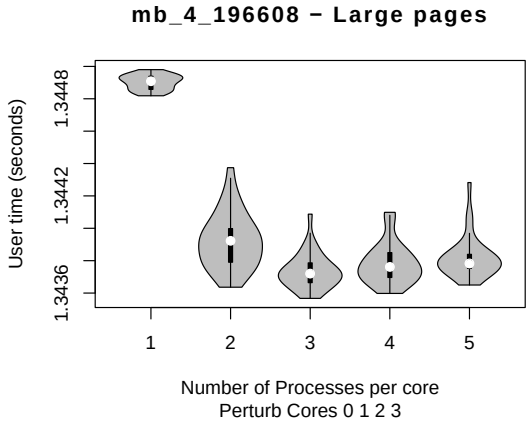


Figure 22: Observed user execution times for mb_4_196608 running on the 0, 1, 2 and 3 cores with large pages

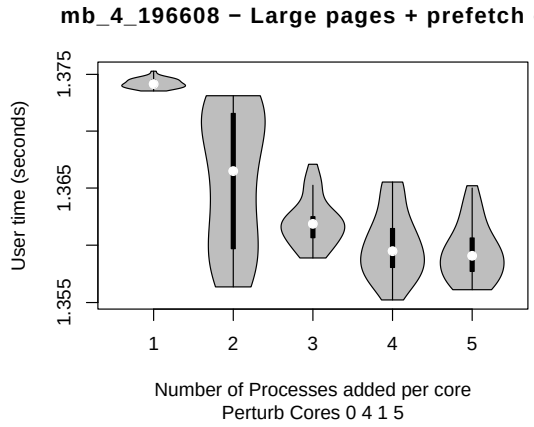


Figure 23: Observed user execution times for mb_4_196608 running on the 0, 1, 4 and 5 cores with large pages and hardware prefetcher disabled

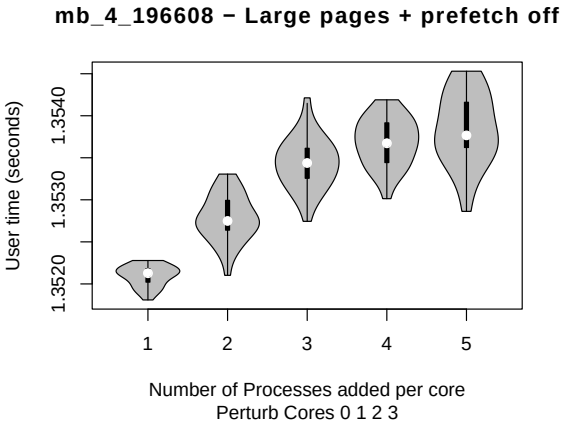


Figure 24: Observed user execution times for mb_4_196608 running on the 0, 1, 2 and 3 cores with large pages and hardware prefetcher disabled

version with small pages. In addition, when we launch co-running processes in background, the program execution times at the user level decreases indicating that something is still going wrong. The last combination that we tried was using large pages and hardware prefetcher disabled together. Figure 23 shows the user execution times for mb_4_196608. The analysis of the figure leads us to the following observations:

1. approximately 37% performance improvement against the basic runtime configuration (small pages and hardware prefetcher enabled);

2. adding more co-running process in background lead to a negligible decrease in program execution times at user level $\approx 1\%$ (from $1,375$ to $1,355$ seconds).

The reason of this negligible impact of the co-running process is due to using large pages and disabling the hardware prefetcher. In this runtime configuration mb_4_196608 does not suffer from a significant number of L2 cache misses. Consequently, the influence of the the co-running process to reduce the contention on L2 is no longer as beneficial as observed in the previous section. In the opposite side, the use of small pages and the hardware prefetcher has a negative impact on the performances of mb_4_196608. Indeed, using both of them lead to a significant number of cache L2 misses (conflict misses where useful cache lines are evicted) making this micro-benchmark behaving poorly. In that situation the co-running processes contribute to significantly reduce contention on L2, decreasing the program execution time at the user level.

5.4 CPU-bound micro-benchmarks

In the previous sections, we presented performance data related to applications intensively accessing the memory hierarchy (particularly the L2 cache). But, what is the expected behaviour when the application has a low access rate to memory? Answering the latter question leads us to expet running applications having low memory access frequency. Listing 4 shows the code of `primenumber` application. As we can see from the listing, the code is CPU-bound (the memory access is limited to L1 data and instruction caches).

Listing 4: OpenMP primenumber code

```
int prime_number ( int n ) {
  int i, j, prime, total = 0;

  # pragma omp parallel shared (n) private (i,j,prime) reduction (+:total)
        # pragma omp for
        for ( i = 2; i <= n; i++ )
        {
        prime = 1;
        for ( j = 2; j < i; j++ )
        {
                if ( i % j == 0 )
                {
                        prime = 0;
                        break;
                }
        }
        total = total + prime;
        }
        return total;
}
```

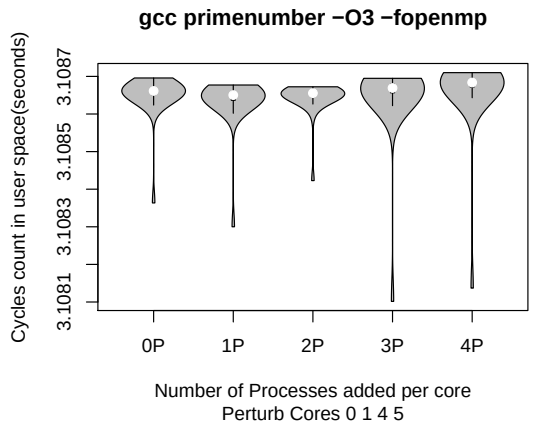


Figure 25: Observed User Execution Times for primenumber application running on the 0, 1, 4 and 5 cores

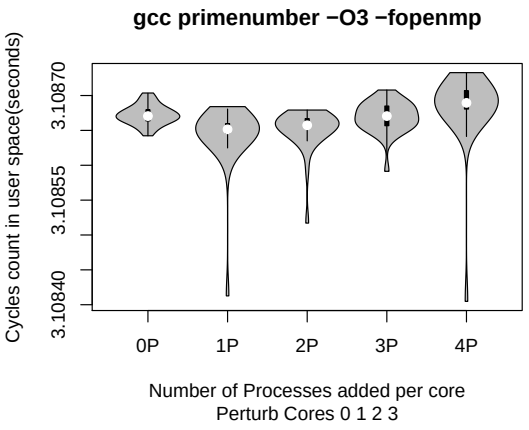


Figure 26: Observed User Execution Times for primenumber application running on the 0, 1, 2 and 3 cores

Figures 25 and 26 show program execution times for `primenumber` application running on shared L2 caches (0,1,4,5 cores) and non shared ones (0,1,2,3 cores) respectively. The interesting observations from these experiments are:

1. Running the application on the two runtime affinities does not change user execution times.

2. The variability of user execution times for each violin plot on the figures is negligible (looking very carefully to the Y-axis).

3. Running the co-running processes affect only the whole (real) execution time. Indeed, the whole execution times increases when we add more co-running processes. This situation is expected since there are many tasks in the system and they need some time to complete.

N	M	Benchmark	Chunk size = M*sizeof(int64)
4	196608	mb_4_196608	1.5 MB
4	190000	mb_4_190000	1.44 MB
4	100000	mb_4_100000	0.76 MB

Table 3: Values taken by N and M in the outermost and the innermost loops in the micro-benchmarks code

Till now, co-running tasks where empty non terminating loops. The next section studies the behaviour of the micro-benchmarks when running with some modified versions of co-running processes. The co-running processes are of two kind : 1) processes having a varying amount of time spent in the `sleeping` state, and 2) processes intensively accessing cache L2 caches

5.5 Variability of execution times under the influence of other co-running processes

The code of co-running processes used in previous sections was a non terminating loop with no data access. To be mre confident in our analysis, we generated two other versions of the co-running processes. The aim of the following experiments is to analyse the access behaviour of the micro-benchmarks to shared resources (cores, L2 caches). Following the experimental setup introduced in Section 5.2.3, we tested three micro-benchmarks having the values of N and M presented in Table 3.

We have to notice that we kept the two runtime scenarios which are : 1) running all the tasks in cores that not share any L2 cache and 2) running all the tasks in cores where there are sharing of cache L2. Details and results of the two experiments are presented in the following sections.

5.5.1 Controlling the co-running processes time spent in the sleeping state

In this experiment we have re-written the C code of the co-running process of Listing 2. The difference between the initial version and the modified version (Listing 5) of co-running processes is on how long they runs. We mean by "how long they runs", the time spent by tasks in the running and sleeping states in the scheduling algorithm of the `Linux` kernel. Therefore, the time spent in these states may influence how the threads of the micro-benchmarks access to the shared resources which may disturb their execution times.

In this modified version of the co-running processes, we added a system call to the `usleep` [2] function. We tried four values given as parameters to `usleep`: 10000, 1000, 100 and 10 micro seconds. The last values highlight four different periods where the disturbing processes are in the sleeping state. For larger values given to `usleep`, the disturbing processes spent the most of their slice time in the sleeping state. At the opposite side, smaller values means that the co-running processes are consuming a large part of their slice time. Listing 5 gives the code of the modified version of the co-running processes:

Listing 5: Background concurrent processes code

```
int stress () {
    while (1) {
        for (g = 0 ; g < 65536 ;g++)
            ;
        usleep(S);
    }
    return 0;
}
```

[2] The `usleep` suspend execution of the calling process for microseconds interval

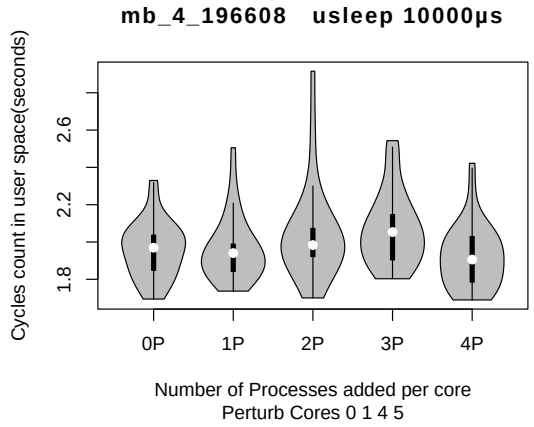


Figure 27: Observed User Execution Times for mb_4_196608 with co-running concurrent processes calling `usleep` for 10ms

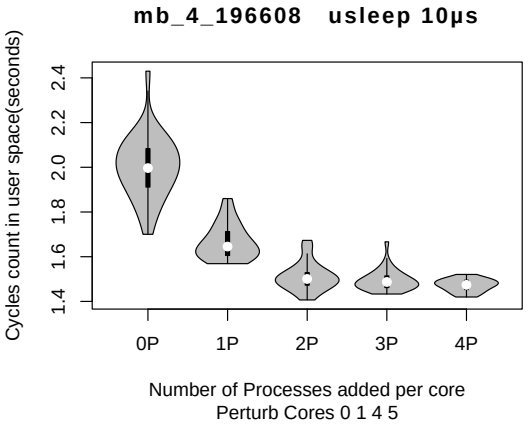


Figure 28: Observed User Execution Times for mb_4_196608 with co-running concurrent processes calling `usleep` for 0.01ms

Figures 27 and 28 show program execution times for the benchmark mb_4_196608 when his threads are launched in a shared L2 configuration. The X-axis represents the number of co-running processes running in each core at the same time with the micro-benchmarks threads. We highlight two cases which represent the two extreme scenarios in the experiment:

1. The co-running processes calls the `usleep` system call for 10000 micro seconds (equivalent to 10 milliseconds);

2. The co-running processes calls the `usleep` system call for 10 micro seconds.

When the co-running processes call `usleep` for 10000 micro seconds, we can see clearly that the execution times does not decrease after adding the co-running processes. The benchmarks run approximately between 1.8 and 2.6 seconds whatever the number of co-running processes is. In addition, from Figure 3, we can see that the number of cache L2 misses are very similar in all the runs. This situation is not the same when the co-running processes call `usleep` for 10 micro seconds. In this scenario, the sleeping period is too short to let the micro-benchmarks running without disturbance. For this reason, we can observe clearly in Figure 3 that the user level program execution times decrease when we add more running co-running processes. The observation of execution times decrease for co-running processes having a short time sleeping period are very similar to those we have seen in the previous sections.

Analysing the figures of the current experiment strengthens our intuitions. We explained that the decrease in program execution times at the user level is due to the influence of the co-running processes in reducing the pressure on the caches. Thus, when they consume a large part (co-running processes calling `usleep` for 10 micro seconds for example) from their slice time, the probability that the threads of the micro-benchmarks try to access at the same time to the shared L2 caches decreases. This situation leads to decrease the cache L2 miss rate, leading in turn, user execution times to decrease. At the other side, when the co-running processes spend most of their time in the sleeping state (co-running processes calling `usleep` for 10 milliseconds for example), they do not interfere enough to play a role of a regulator. Thus, the micro-benchmarks threads run normally as if they were the only ones to run in the cores, leading to very close `median` user level execution times of the different configurations. But we have to notice, that the variability in program execution times of each configuration still significant.

It should be noted that we can not extend the last conclusion to any kind of co-running processes (those we have tested does not access to the L2 cache for example). The next section illustrate cases where the co-running processes access to the L2 cache. Thus, the behaviour of the micro-benchmarks is different.

5.5.2 The impact of accessing the L2 cache by the co-running processes on the micro-benchmarks performance

For this experiment we wrote a new version of the co-running processes. Listing 6 shows the code of the new version.

Listing 6: Co-running processes code

```
int stress () {
    long i, M = 196608;
    long tab[M];

    for(i = 0 ;  i < M; i++)
        tab[i] = 0;
    while (1) {
        for(i = 0 ;  i < M; i++)
            tab[i]++;
    }
    return 0;
}
```

From listing 6, we see that the co-running processes now access to the L2 cache. The access to the L2 cache is highlighted by the declaration of an array of 196608 elements which fills up 1.5 MB approximately in the L2 cache for each co-running process added to the system. Listing 6 shows also that all the elements of the array `tab` are accessed (read and write access).

Figure 29 reports program execution times at user level for the mb_4_196608 micro-benchmark using four threads sharing two L2 caches. The figure does not shows any decrease in program execution times at the user level. In contrary, we observe that the new version of the co-running processes influence differently the micro-benchmarks behaviour. In this configuration, we can conclude two things:

1. The user level execution times tend to increase (looking to the `median` execution times of the runs).
2. A significant variability is observed on the 31 execution times of the micro-benchmarks. This is due to the perturbation effect induced by the co-running processes which pollute the L2 cache. We have to notice that this variability does not appears for real execution times.

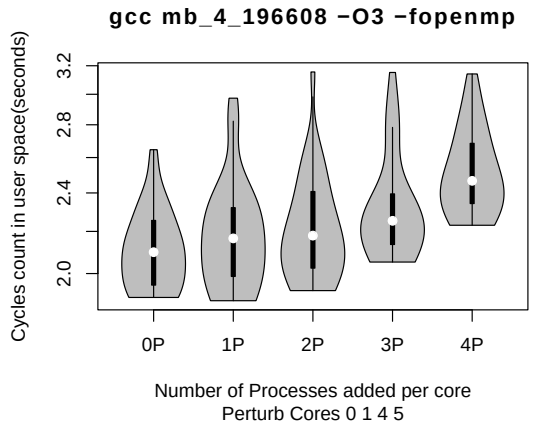


Figure 29: Observed User Execution Times for mb_4_196608 binded to the 0, 1, 4 and 5 cores

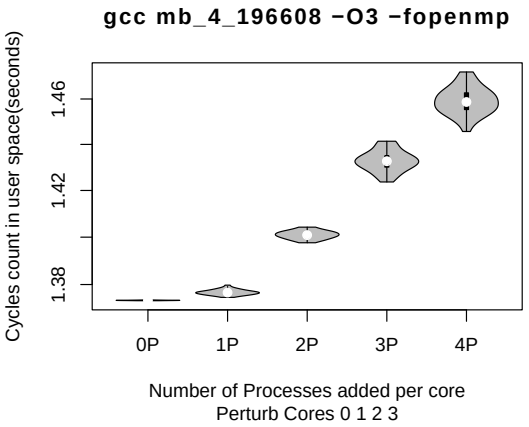


Figure 30: Observed User Execution Times for mb_4_196608 binded to the 0, 1, 2 and 3 cores

Running the four threads of mb_4_196608 on cores which does not share L2 cache also shows that the `median` user execution times slightly increases. Figure 30 reports the execution times in this software

configuration. We observe too that the 31 execution times at user level of each software configuration do not exhibit a significant variability.

The next section presents a study of the effect of running real applications with an affinity to the system cores taking into account the impact of sharing the last level cache (L2).

6 Study of the affinity effect on SPEC OMP2001 execution times

In the previous section, we have seen the relation between performances variability and the effect of co-running independant background processes. We conduct the following experiments aiming to check the impact of changing the scheduling affinity of SPEC OMP2001 threads on performances variability (studied in Section 4). When affinity is enabled, we mean that we fix the placement of the threads on the cores of the processor.

We used the `gcc 4.4.3` and `icc 11.0` compilers. For each SPEC OMP2001 application, we generated a multi-threaded version of each benchmark by setting `-O3 -fopenmp` and `-O3 -openmp` compilation flags respectively for the `gcc` and `icc` compilers. We run each application with respectively 2, 4 and 6 threads under three runtime configurations :

1. Running the benchmarks without scheduling affinity (affinity disabled, threads placement let to the OS).

2. Running the benchmarks under the `icc` compiler *compact* [5] affinity strategy. Specifying *compact* as affinity assigns the OpenMP thread <n>+1 to a free core as close as possible to the core where the <n> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads. We selected this configuration because it leads to increase the cache L2 sharing between threads. Thus, we can see the impact on performances. Indeed, not all the applications can take advantage from it.

3. Running the benchmarks under the `icc` compiler *scatter* [5] strategy. Specifying *scatter* affinity strategy distributes the threads as evenly as possible across the entire system. scatter is the opposite of compact. Running applications under this strategy may be benifical to alleviate the problem of system bus contention of neighbours cors.

Figure 33 and Figure. 34 show violin plots of program execution times (CPU time) for the `wupwise` and `swim` applications (from SPEC OMP2001 benchmarks) compiled with the `gcc` and `icc` compilers. In each figure, three violin plots report the execution times when the benchmarks are launched with 2, 4 and 6 threads. The X-axis represents the three runtime configurations for the applications explained above. The Y-axis represents the 31 observed execution times for each software configuration. We can make the following observations:

1. When the scheduling affinity is disabled, we observe a significant variability in execution times for SPEC OMP2001 benchmarks. If we consider the case of `swim` in Figure 34 compiled with `gcc`, the version with 2 threads runs between 79 and 110 s, the version with 4 threads runs between 73 and 90 s and the version with 6 threads runs between 71 and 82 s. Figure 34 shows that when the benchmark is compiled with the `icc` compiler, it exhibits the same variability.

2. The variability was insignificant in almost all the benchmarks when the schedlung affinity was enabled. The variability disappears either when the threads shares L2 (*compact*) cache or not (*scatter*). Figure 33 shows for the `wupwise` application compiled with `gcc` that the version with 2 threads runs $\approx$ 454 s when they shares the cache L2 (2 threads runs on 2 cores sharing single L2 cache *compact*) and runs between 419 and 421 s when they do not share it *scatter*.

3. The `art` benchmark used with both compilers and the `apsi` with `gcc` exhibits a less sensitivity to changing scheduling affinity. We observed that even when we set up the binding feature, variability in execution times still appear (see Figures 35 and 36).

4. We observed in 7 from the 9 tested benchmarks that they run faster when they are launched with a *scatter* strategy). The benchmarks which take benefits from L2 cache sharing *compact* are `ammp` and `galgel` with both compilers (see Figures 37 and 38).

5. Fixing the affinity between the threads does not remove the performance variability of all the benchmarks. For instance, the applications `art` and `apsi` compiled with `gcc`, and `art` compiled with `icc`, still have some sensitive variability. It can be explained by cache effects, as explained below. Furthermore, we suspect that OpenMP and synchronisation overhead may play a role in performance variability.

To check the origin of the performance variability observed when we disable the affinity, we study the impact of thread placements (fixed by the OS) on the cache effects. For instance, we run `swim` and we report his number of last level cache misses (L2 cache misses). Figure 32 shows violin plots summarising the number of cache L2 misses when `swim` runs with 2 threads. We observe clearly that the variability of the execution times observed in Figure 31 is closely related to the number of cache L2 misses. Indeed, when we binded the threads of `swim` explicitly to the system cores, we observed insignificant variability in the execution times. But when we let the operating system to handle the threads placement on the cores the situation was completely different and we observed an important variability. The interesting thing is that higher execution time in the configuration without affinity was accompanied with a higher cache L2 misses number. This situation shows that `swim` is sensitive to cache affinity.

The increase in the number of L2 cache misses does not explain the cause of the variability observed but just an effect. We observed that the main factor contributing to the important performance variability was the thread migration by the operating system kernel. Indeed, we reported for the `swim` application the mapping of threads to cores each time a new parallel region is entered. The analysis of the tracing of the mapping event allow us to see that the runs with high execution times, the application threads have suffered from a thread migration. Thus, the migration impact negatively the cache utilisation leading to a significant performance variability.

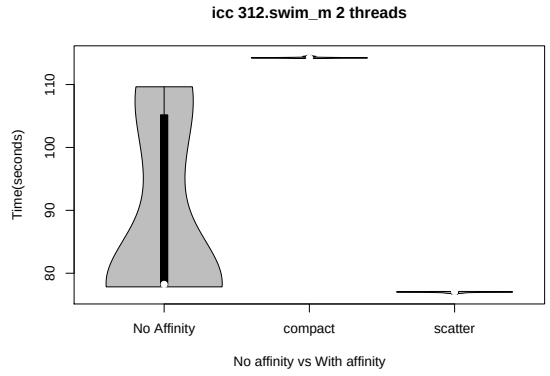


Figure 31: Observed cycles count in `swim` running with 2 threads

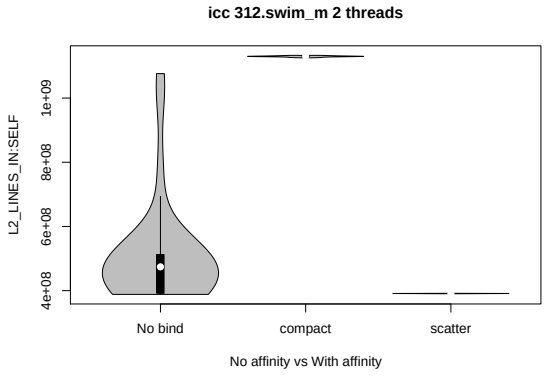


Figure 32: Observed cache L2 lines misses in `swim` running with 2 threads

In addition to `swim`, we observed also that the performances of `wupwise, applu, equake, apsi, fma3d, ammp` applications are sensitive to cache affinity too.

In this section, we clearly observe that fixing affinity between threads removes performances variability in many applications, but not all: there are still other influencing factors that make executions time to vary. By now, it is not clear if fixing affinity would improve the average or the median execution time. Sometimes, it is better to let some hazard (OS) to decide about thread binding, since it is not clear if simple strategies such as *scatter* and *compact* are efficient.

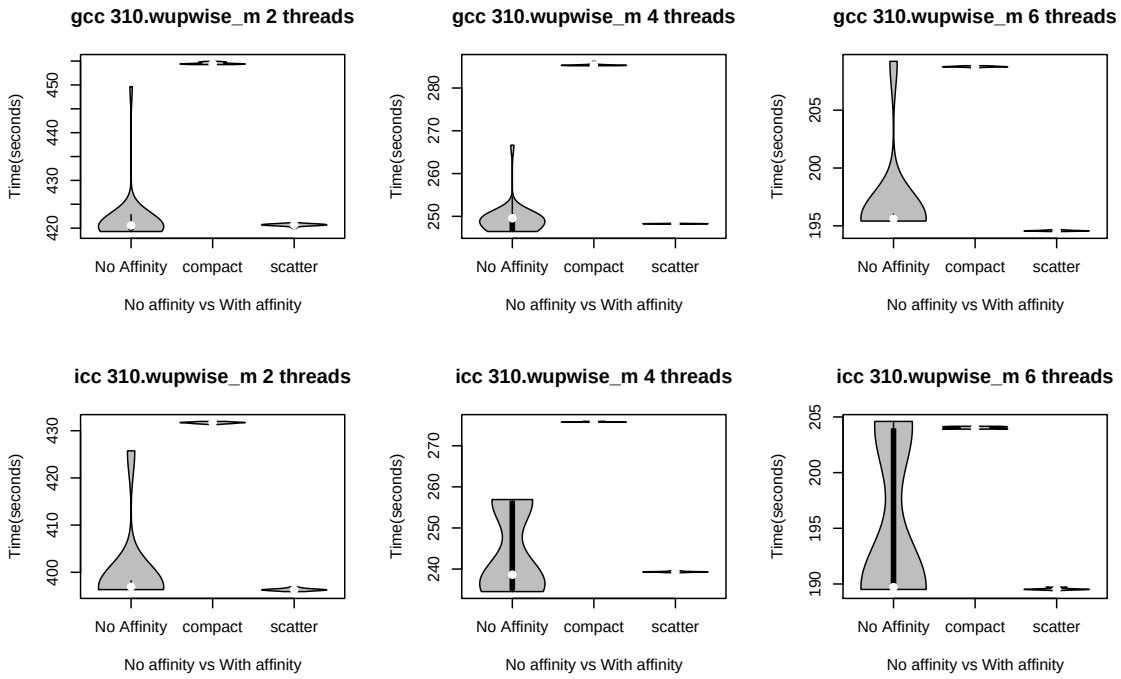


Figure 33: Observed Execution Times of the `wupwise` Application (compiled with `gcc` and `icc`)

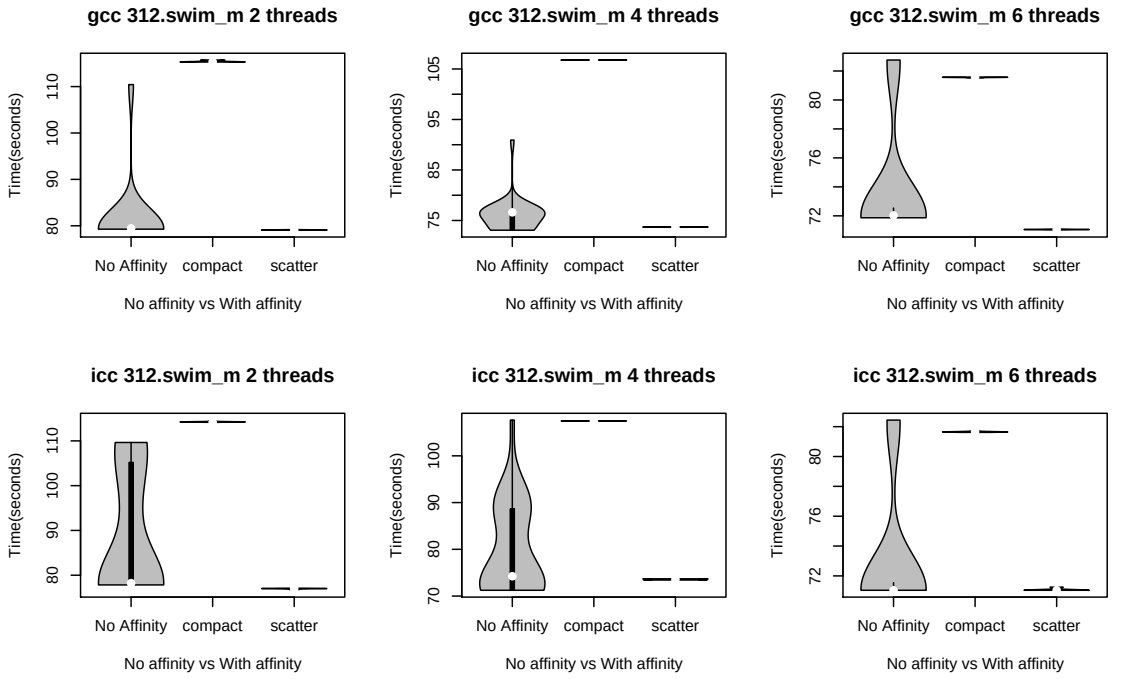


Figure 34: Observed Execution Times of the `swim` Application (compiled with `gcc` and `icc`)

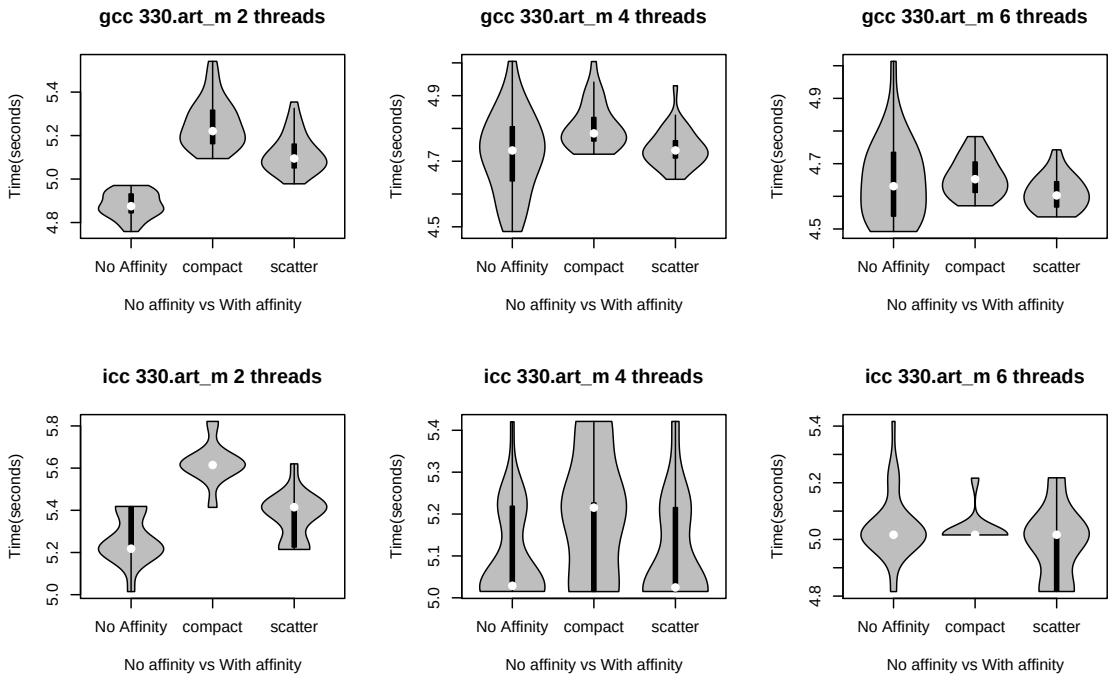


Figure 35: Observed Execution Times of the `art` Application (compiled with `gcc` and `icc`)

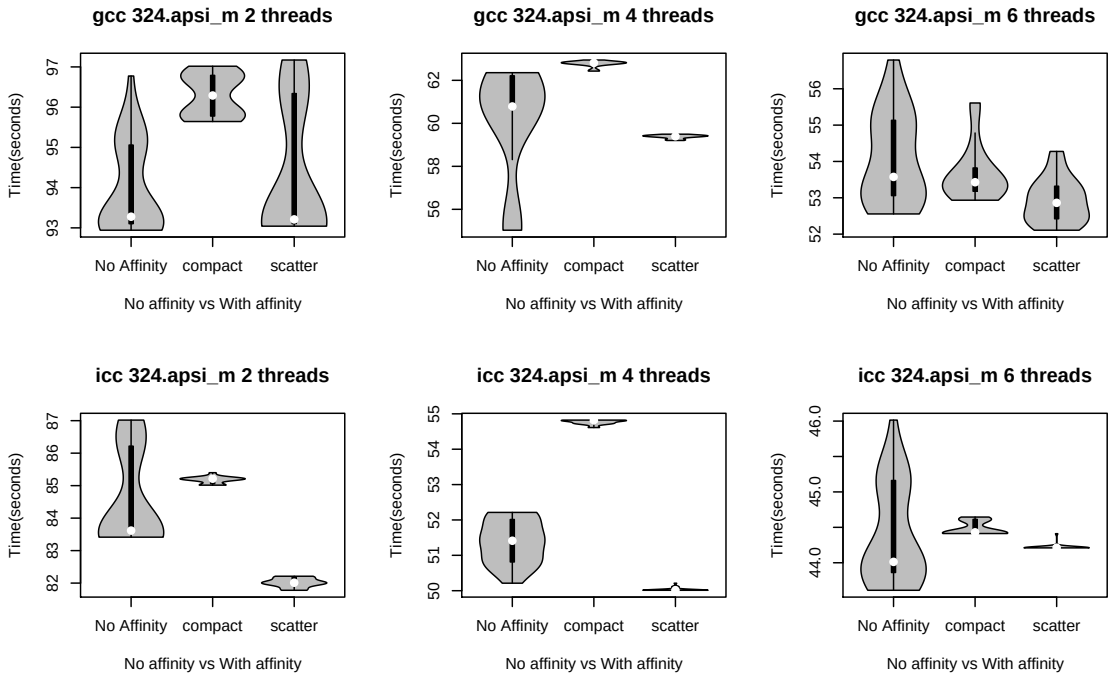


Figure 36: Observed Execution Times of the `apsi` Application (compiled with `gcc` and `icc`)

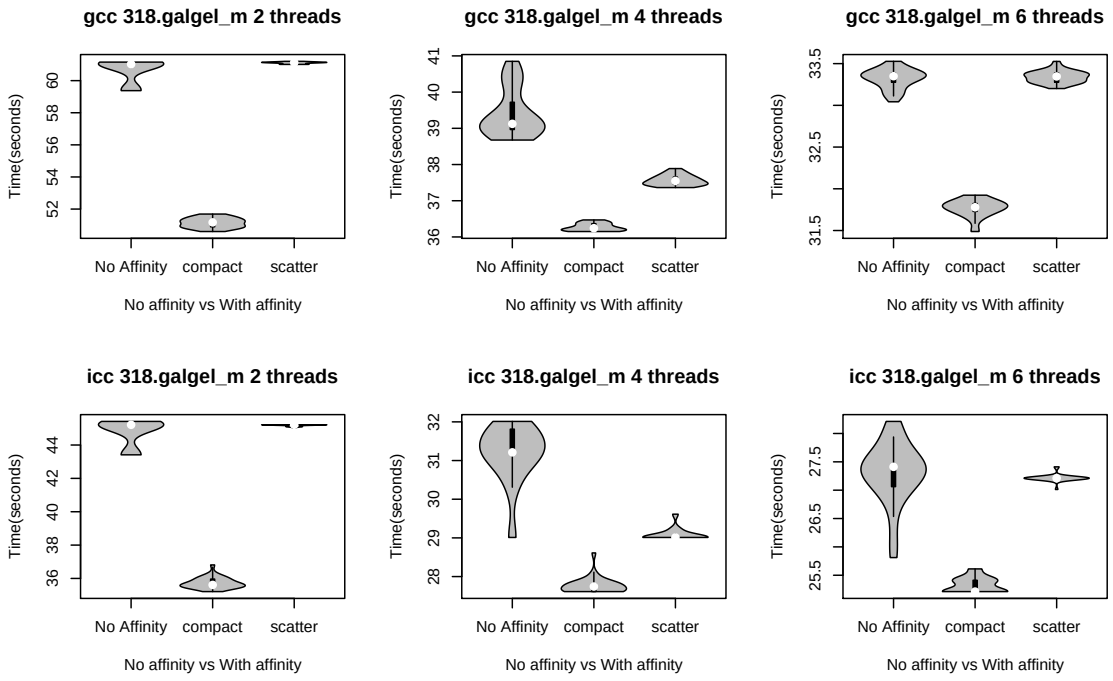


Figure 37: Observed Execution Times of the `galgel` Application (compiled with `gcc` and `icc`)

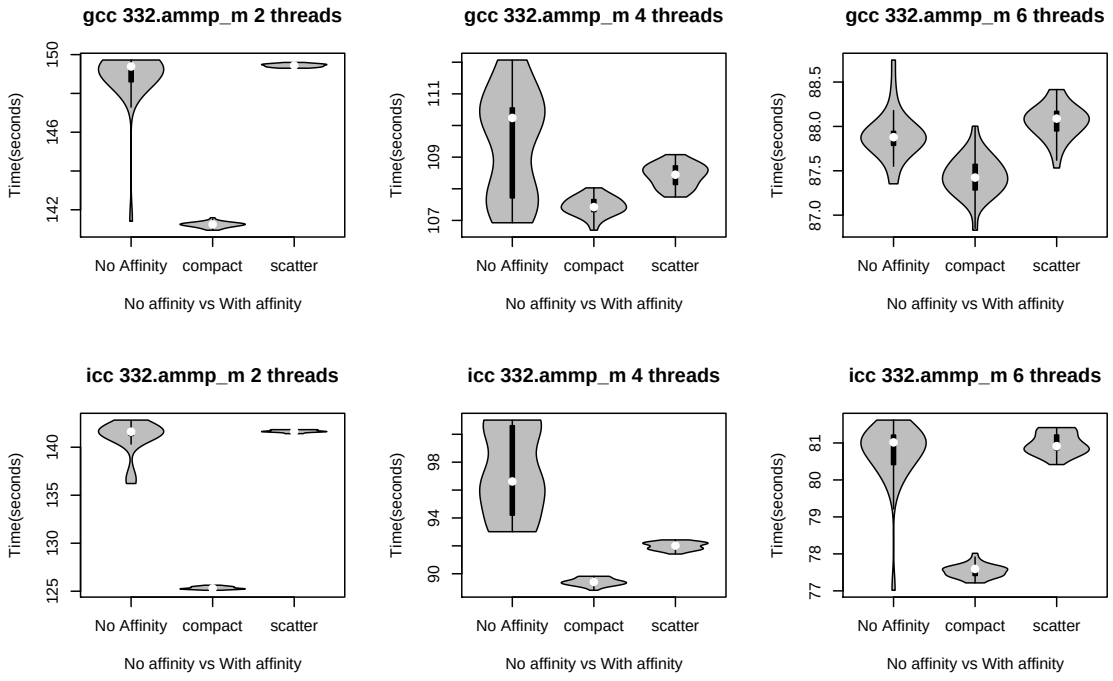


Figure 38: Observed Execution Times of the `ammp` Application (compiled with `gcc` and `icc`)

7 Related Research Activity

7.1 Variability of program execution times

Collective optimisation [6] is a valuable effort in the community of program optimisation aiming to log performance numbers in a central database. One of the main motivations behind this effort is the disparity of performance scores reported in the literature, and the difficulty in comparing, checking and reproducing them. A fraction of the non reproducibility of experimental code optimisation results comes from the variability of program execution times; if not correctly reported or evaluated, the overall reported speedups would have a low chance of being reproduced.

Another effort dealing with variability is the article of *raced profiles* [7]. That performance optimisation system is based on observing the execution times of code fractions (functions, and so on). The mean execution time of such code fraction is analysed thanks to the test of student, aiming to compute a confidence interval for the mean. We have two main differences with the previous work. First, we execute multiple times whole programs, not fractions of them. Consequently, our successive executions are independent (this is not the case when we execute the same function multiple times inside the same program). Second, this previous article does not fix the data input of each code fraction: the variability of execution times when the data input varies cannot be analysed with the Student t-test. Simply because when data input varies, the execution time varies inherently based on the algorithmic complexity, and not on the structural hazard. In other words, observing distinct execution times when varying data input cannot be considered as hazard, but as an inherent reaction of the program under analysis.

Eeckhout et al. [8] study the impact of input data sets on program behaviour. They use statistical data analysis techniques cluster analysis to explore the workload space in microprocessors design. The final goal is to select a limited set of representative benchmark-input pairs that span the complete workload space. Alameldeen et al. [9] study time and space variability in architectural simulation studies of multi-threaded workloads. Time variability occurs when a workload exhibits different characteristics during different phases of a single run. Space variability occurs when two runs exhibit different performance characteristic. For instance, in our work we focus on the later definition of performance variability. Last, program execution times variability has been shown to lead to wrong conclusions if some execution environment parameters are not kept under control [2]. For instance, the experiments on sequential applications reported in [2] show that the size of Unix shell variables and the linking order of object codes both may influence the execution times.

7.2 Tools for performance measurement

A standard approach to achieve or study the performance of applications codes is to use performance measurement tools. `GNU Prof` [10] is a well known tool for timing measurement. It relies on time based sampling, it is not suitable for multi-threaded programs. `VTune` [11] is a commercial performance evaluation tool from Intel. Contrary to `GNU Prof`, it is based on event based sampling using hardware performance counters available on modern microprocessors. `OProfile` [12] is a free software relying on statistical sampling of hardware performance counters. `OProfile` and `VTune` suffer from the lack of counting measurement functionality and in some cases may need some privileged credentials to use them.

All the process of configuring and accessing the hardware performance counters have to be done in kernel mode (kernel mode privileges). Furthermore, the need to keep per-thread counting have lead to the development of kernel extensions extending the operating system's context switch code to save restore the counter registers. Thus, allowing user code to access the counters in a transparent way. For Linux, the two frequently used kernel extensions are `perfctr` [13] and `perfmon` [14]. `perfmon` is aiming to design and implement, on all major architectures, a standard Linux kernel interface (OS-dependent), to access the hardware performance counters of modern processors. The project also developed a user library, `libpfm`, and a tool, `pfmon` which we have used extensively in the context of our study. `PAPI` [15] is a library to access hardware performance counters. The aim of the project is to provide a standard and platform independent (OS and processor) API for accessing hardware performance counters across the variety of hardware architectures and operating systems.

8 Conclusion

This report shows clearly that, even if a machine has low overhead and the dynamic voltage scaling is inactive and the automatic hardware prefetcher is disabled, the execution times of OpenMP applications on multicore platforms may be very instable (variable). This implies to study a new performance criteria for code optimisation that was not important for sequential codes, which is performance stability. We showed that binding threads on cores removes the performance variability in most of the cases, but some applications have still instable performances after fixing the thread affinity. This means that other factors (distinct from thread binding) are still playing important role on performance variation.

Our report also highlights that executing separate co-running processes in parallel with the threads of an OpenMP application may be beneficial for the user level execution time (but not for the whole real time execution). Indeed, co-running processes may reduce the contention or competition between the threads on data that reside on shared cache levels: co-running processes push the threads of the OpenMP application to run with less concurrency, smoothing the conflicts on shared cache levels. While co-running processes are not beneficial for real execution times, they contribute to reduce the user level execution times, which means that the efficiency of the whole system is improved (fraction of time where the CPU really executes applications is improved).

The speedups that are reported in the literature are usually observed in ideal environments, in ideal experimental setups, after retaining *good* execution times. The end-user however may not observe the declared speedups, which may cause frustration. The reason is that end users do not work in ideal environments: they may not know what are the hidden factors that influence the performance stability of their codes, or simply they may not have a root (or enough rights) to the machine to fix it. Consequently, when an end-user executes an application that is declared optimised, he would have a very low chance to observe such performance improvements with the declared speedup.

Performance stability problems have been already noticed in the past for parallel applications on massively parallel supercomputers or on distributed systems. Since both fields are devoted to experts in computer science and engineering, execution times variability was not a high priority problem compared to other ones: indeed, in high performance computing, only maximal performances matter till now, since high scores allow to have highly ranked machines in TOP500.

The multicore era brings high performance computing to the general purpose market. By now, non experts will have to use small supercomputers, which are their desktop workstations. In this case, we think that performance variation becomes an important quality criteria for general purpose end users. We cannot rely on their expertise on performance tuning and analysis to deal with the problem, as what was the case in classical high performance computing. When execution times vary substantially, we have to make correct statistics to evaluate the execution speed improvements. We have already studied and implemented a rigorous statistical protocol for declaring fare speedups for the average and the median execution times. The tool is called *The Speedup-Test*, and is available in [4].

In a future work, we will study ways of reducing performance variations, hopefully without losing too much average or median execution times. We will focus on better strategies for thread binding on cores using affinity between threads.

References

[1] Standard Performance Evaluation Corporation, "SPEC CPU." http://www.spec.org/, 2006.

[2] T. Mytkowicz, A. Diwan, P. F. Sweeney, and M. Hauswirth, "Producing wrong data without doing anything obviously wrong!," in *ASPLOS*, 2009.

[3] Raj Jain, *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling*. New York: John Wiley and Sons, 1991.

[4] S.-A.-A. Touati, J. Worms, and S. Briais, "The Speedup-Test," tech. rep., University of Versailles Saint-Quentin en Yvelines, Jan. 2010. http://hal.archives-ouvertes.fr/inria-00443839.

[5] Intel Corporation, "Intel C++ Compiler 11.1 User and Reference Guides." http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/index.htm.

[6] G. Fursin and O. Temam, "Collective Optimization," in *The 4th International Conference on High Performance and Embedded Architectures and Compilers (HIPEAC)*, 2009.

[7] H. Leather, M. O'Boyle, and B. Worton, "Raced Profiles: Efficient Selection of Competing Compiler Optimizations," in *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09)*, ACM SIGPLAN/SIGBED, June 2009.

[8] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the impact of input data sets on program behavior and its applications," *J. Instruction-Level Parallelism*, vol. 5, 2003.

[9] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), p. 7, IEEE Computer Society, 2003.

[10] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.

[11] Intel Corporation, "Intel: VTune Performance Analyser." http://software.intel.com/en-us/intel-vtune/.

[12] J. Levon, "OProfile manual." http://oprofile.sourceforge.net/doc/.

[13] Pettersson, Mikael, "perfctr." http://user.it.uu.se/ mikpe/linux/perfctr/.

[14] S. Eranian, "The perfmon2 interface specification," tech. rep., Hewlett-Packard Laboratory, Feb. 2004. http://www.hpl.hp.com/techreports/2004/HPL-2004-200R1.html.

[15] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, (Washington, DC, USA), p. 42, IEEE Computer Society, 2000.

UFR des sciences	:	45 avenue des Etats Unis. 78035 Versailles cedex
IUT de Velizy et de Rambouillet	:	10-12 avenue de l'Europe. 78140 Vélizy.
UFR des Sciences Sociales et des Humanité	:	47 boulevard Vauban. 78047 Guyancourt cedex
Faculté de droit et de science politique	:	3, rue de la Division Leclerc. 78280 Guyancourt
IUT de Mantes en Yvelines	:	7 rue Jean Hoët - 78200 Mantes la Jolie
UFR de Médecine Paris-Ile-de-France Ouest	:	9 boulevard d'Alembert Bâtiment François Rabelais. 78280 Guyancourt
Institut des Langues et des Etudes Internationales	:	5-7, boulevard d'Alembert. 78280 Guyancourt
Institut des Sciences et Techniques des Yvelines	:	45 avenue des Etas Unis - 78035 Versailles cedex
Observatoire des Sciences de l'Univers de l'UVSQ	:	11 boulevard d'Alembert. 78280 Guyancourt

Éditeur
UVSQ - Siège : 55 avenue de Paris - 78035 Versailles Cedex (France)
http://www.uvsq.fr