Analysing the Variability of OpenMP Programs Performances on Multicore Architectures

Abdelhafid Mazouz[1], Sid-Ahmed-Ali Touati[1,3], and Denis Barhou[2]

[1] University of Versailles Saint-Quentin-en-Yvelines, France
[2] University of Bordeaux, LaBRI / INRIA, France
[3] INRIA / Saclay, France

Abstract. In [8], we demonstrated that contrary to sequential applications, parallel OpenMP applications suffer from a severe instability in performances. That is, running the same parallel OpenMP application with the same data input multiple times may exhibit a high variability of execution times. In this article, we continue our research effort to analyse the reason of such performance variability. With the architectural complexity of the new state of the art hardware 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. Understanding the performance instability in current multicore architectures is even more complicated by the variety of factors and sources influencing the applications performances. This article focus on the effects of thread binding, co-running processes, L2 cache sharing, automatic hardware prefetcher and memory page sizes.

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 (thread and process scheduling, memory management, shared resources managements, energy consumption and heat dissipation of cores, etc.). Furthermore, a 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 process/thread scheduling to achieve a better performance stability, reproducibility and predictability.

In this context, application designers and performance analysts have to iteratively investigate how to achieve the best performances; they also have to check the behaviour of their applications on the executing machines. 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 SPEC OMP2001 [11] benchmark applications in addition to our own micro-benchmarks. 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 process/thread, sharing of the last level of cache and environment size (that have been investigated in [9]). Thus, if we execute a program (with the same data input) n times, we may obtain n distinct execution times.

In this article, we use extensive experiments aimed to measure, quantify and analyse the variations of program execution times on an Intel multicore machine. We report measurement results for parallel multi-threaded applications (SPEC OMP2001 and micro-benchmarks). Our parallel applications use the OpenMP paradigm, one of the most used in parallel programming model on shared memory computers. Unlike single-threaded applications [8], we show that the variations of execution times of OpenMP applications are really sensitive from a human user point of view. This is the case if the parallel application is run alone on the machine or if it is run with other co-running independent processes.

Our article is organised as follows. Sect. 2 details our experimental setup (the executing machine, the experimental environment and methodology). Then, Sect. 3 studies the impact of thread affinity (binding) on the variation of SPEC OMP execution times when executed alone on the machine. Sect. 4 investigates the same study when a SPEC OMP application is run in parallel with other independent co-running processes. In order to analyse and understand the interaction between the software and low level micro-arhitectural behaviour, we designed some micro-benchmarks in Sect. 5. These micro-benchmarks help us to explain the performance variability observed on complex applications such as SPEC OMP. Before concluding, related work is discussed in Sect. 6. This article is a synthesis of the detailed research report published in [7]. If interested, the reader is invited to read the detailed document to get a precise idea on all the experiments conducted for this study.

2 Experimental setup

2.1 Test machine

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 architecture of the test machine is summarised in Fig. 1.

2.2 Software environment

The version of the linux kernel is X86_64 2.6.26, patched with perfmon kernel 2.81. We used multiple compilers: `gcc` 4.1.3, `gcc` 4-3.2, `icc` 11.0 and `ifort` 11.1, all applied with optimisation level `-O3 -fopenmp`. The experimented benchmarks are SPEC OMP 2001 applications run with the train input and various configurations of thread numbers. We also designed our own micro-benchmarks to analyse the interaction between the software, the micro-architecture and the OS layer.

2.3 Experimental methodology

Our experiments were done using the following practices:

– The test machine was entirely dedicated during the experiments to a single user. The experiments were done on a minimally-loaded machine (disable all inessential OS services except `sshd`);

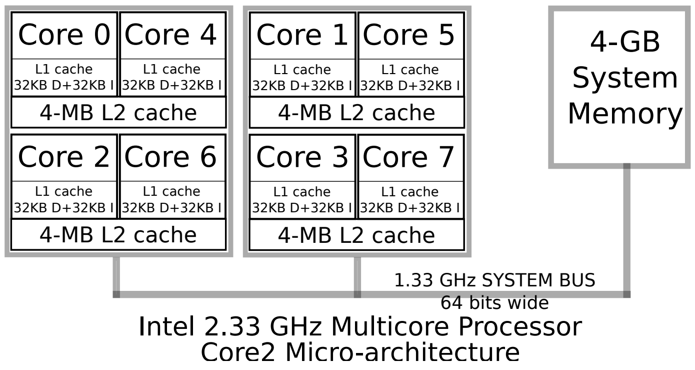


Fig. 1. Dual processor architecture

- The run of each benchmark is repeated 31 times for each software and experimental configuration. This high number of runs allows us to report statistics with a high confidence level;[1]
- Unset all the shell environment variables that were inessential to the execution. We keep constant the total size of the shell variables [9];
- Deactivate 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 (DVS) disabled;
- Use the build system and scripts of SPEC 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 (Runs are repeated one after the other, and not in parallel);
- 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.
- All observed execution times are reported, we do not remove any outlier (except if the run crashes or produces wrong result).

2.4 Definition of program performance variability

When we observe a sample of execution times of program P, say $\{t_1, \cdots, t_n\}$ where t_i is the execution time of the i^{th} run, then we may define the variability according to many metrics. Any used metric must define the *feeling* of the end user about the instability of the execution time of the application. We can use the usual sample variance, or $\frac{|\max_i t_i - \min_i t_i|}{\bar{t}}$ where $\bar{t}$ is the sample mean, or $\frac{|\max_i t_i - \min_i t_i|}{med(t)}$ where $med(t)$ is the sample median. In our study, we use metrics that measure the disparity between extrema observations (outliers):

1. An absolute variability, which is the difference between the maximal and the minimal observed execution times $AV(P) = |\max_i t_i - \min_i t_i|$;
2. A relative variability, which is the absolute variability divided by the maximal observed execution time $RV(P) = \frac{AV(P)}{\max_i t_i} = \frac{|\max_i t_i - \min_i t_i|}{\max_i t_i}$.

[1] We demonstrated in [8] that the observed execution times of most of the applications do not follow a Gaussian distribution (using the Shapiro-Wilk statistical test). This means that the usual mean confidence interval formula and some statistical tests (Student t-test) cannot be used unless the sample size is large enough. While not proved, it is commonly admitted that conducting more than 30 observations constitutes large samples [10, 12].

Now the question is how to decide about a definition of a program with non negligible performance variability? Since any experimental measure brings a sample variation (it is impossible in practice to observe exactly equal execution times), when can we speak about non negligible variability ? In our study, we say that a program P has non negligible performance variability if its absolute variability exceeds one second ($AV(P) > 1s$) or if its relative variability exceeds 1% ($RV(P) > 1\%$). Another definition of variability may exist; In our context we chose the previous definition in order to be close to the feeling of a user executing a program interactively (*i.e.* when he launches the program and he waits for its termination). For instance, we showed in [8] that for SPEC CPU 2006 (executed more than 30 times), $AV(P) \leq 1s$ and $RV(P) \leq 1\%$, which means that their variability is marginal. The next section studies the variability of the SPEC OMP 2001 applications and analyses the impact of thread affinity on it.

3 Analysing the impact of thread affinity on SPEC OMP performance variability

We conduct the following experiments aiming to check the impact of changing the scheduling affinity of SPEC OMP2001 threads on performances variability. When affinity is enabled, we mean that we fix the placement of the threads on the cores of the processor.

We used both `gcc` and `icc` 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 `gcc` and `icc`. 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* [4] 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 OpenMP thread n was placed.
3. Running the benchmarks under the `icc` compiler *scatter* [4] strategy. Specifying *scatter* affinity strategy distributes the threads as evenly as possible across the entire system. Scatter is an opposite affinity strategy compared to compact. Running applications under this strategy may be benifical to alleviate the problem of system bus contention of neighbours cores.

Fig. 2 and Fig. 3 show violin plots of program execution times (CPU time) for the `wupwise` and `swim` applications 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 affinity configurations (no fixed affinity, compact, scatter). The Y-axis represents the 31 observed execution times for each configuration (with violin plot). 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 plot gives the observed median execution time and the thick line through the white dot gives the inter-quartile range. We 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 Fig. 3 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. Fig. 3 shows that when the benchmark is compiled with the `icc` compiler, it exhibits a variability too.
2. The variability is insignificant in almost all the benchmarks when the schedlung affinity is enabled (the observed variability is less than 1.5%). The variability disappears either when the threads shares L2 cache (*compact* binding) or not (*scatter* binding). Fig. 2 shows for the `wupwise` application compiled with `gcc` that the version with 2 threads runs ≈ 454 s when they shares the L2 cache (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*.

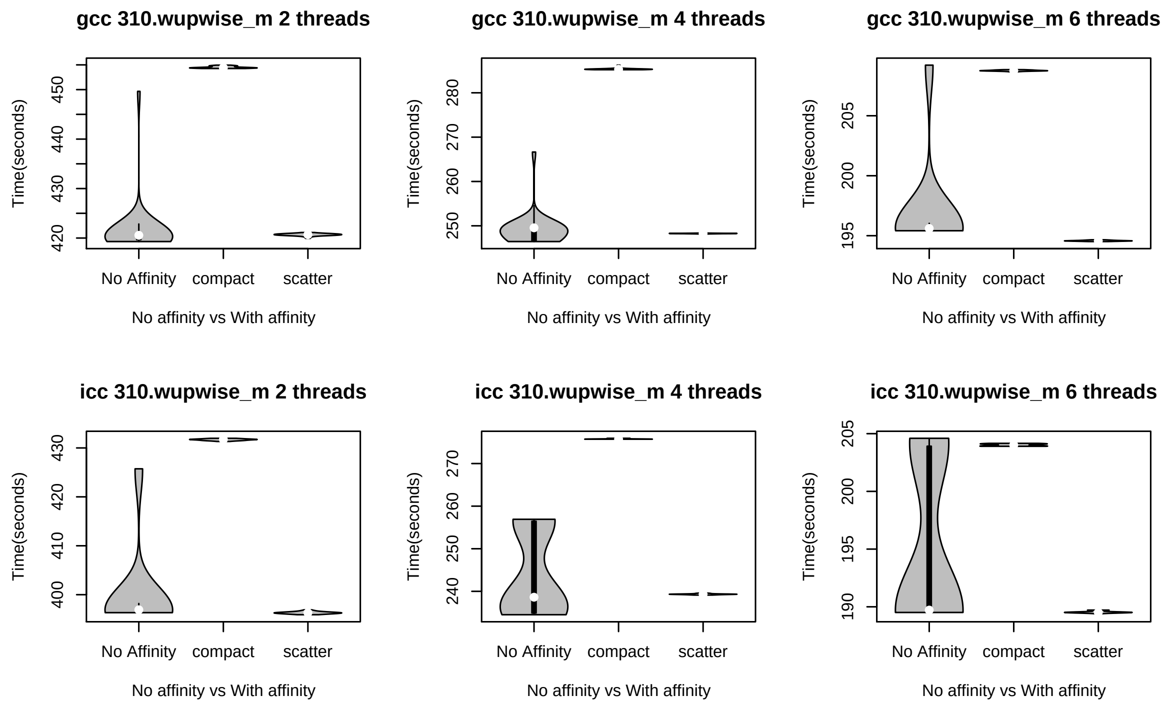


Fig. 2. Observed Execution Times of the `wupwise` Application (compiled with `gcc` and `icc`)

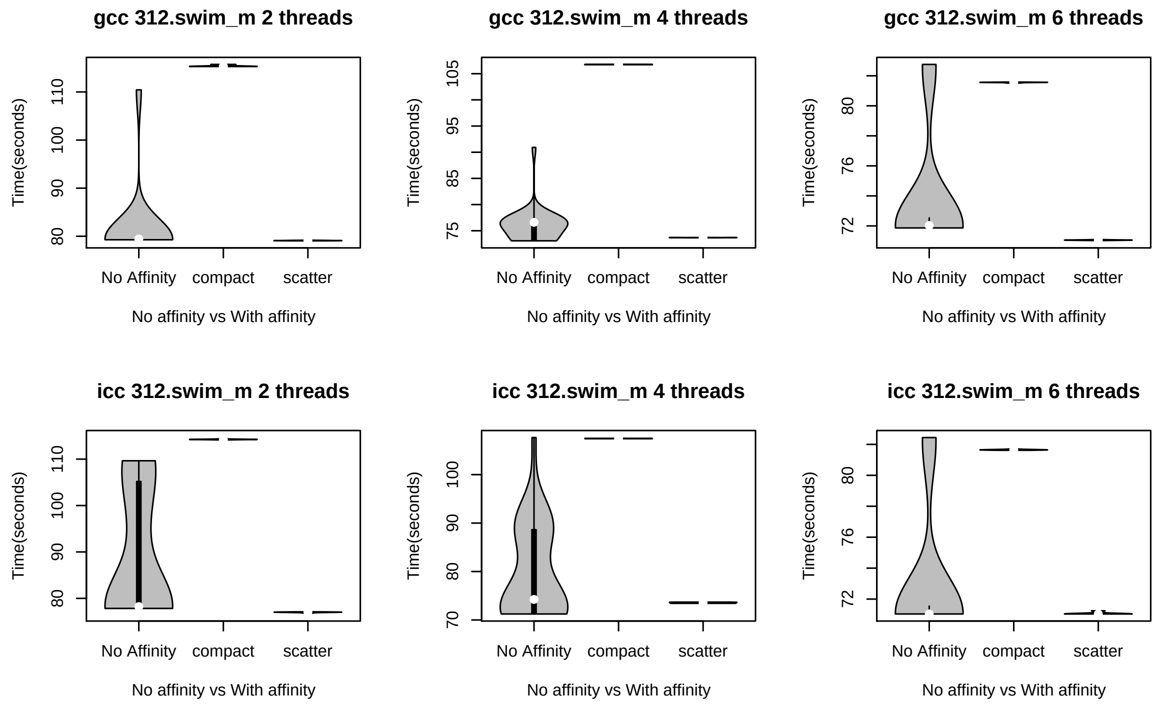


Fig. 3. Observed Execution Times of the `swim` Application (compiled with `gcc` and `icc`)

3. The `art` benchmark (compiled with both `gcc` and `icc`) and the `apsi` (compiled with `gcc`) exhibit a less sensitivity to fixing scheduling affinity. We observed that even when we set up the binding feature, variability in execution times still appear (see Fig. 6 and 7 where the variability exceeds 5%). In other words, fixing the affinity between the threads does not remove the performance variability of all the benchmarks.
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 Fig. 8 and 9).

In order to check the origin of the performance variability observed when we disable the affinity, we study the impact of thread placements on the cache effects. For instance, we run `swim` and we report his number of last level cache misses (L2 cache misses). Fig. 5 shows violin plots summarising the number of L2 cache misses when `swim` runs with 2 threads. We observe clearly that the variability of the execution times observed in Fig. 4 is closely related to the number of L2 cache misses. Indeed, when we binded the threads of `swim` explicitly to the system cores, we observed insignificant variability in the execution times. But when letting the 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 L2 cache misses number. This situation shows that `swim` is sensitive to cache affinity.

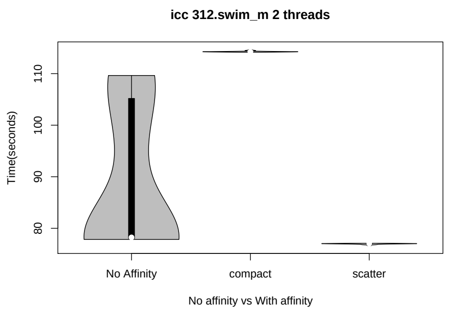


Fig. 4. Observed cycles count in `swim` running with 2 threads

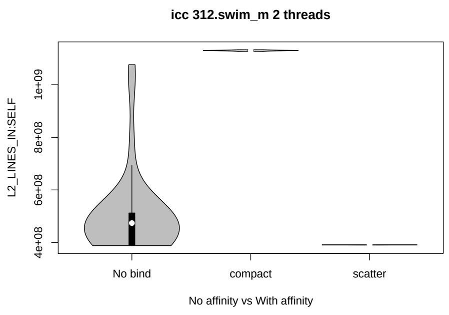


Fig. 5. Observed L2 cache lines misses in `swim` running with 2 threads

When affinity is not fixed, the direct effect is a high variation in the number of L2 cache misses. We figured out also that another important factor contributing to the performance variability is the thread migration operated by OS kernel. Indeed, we traced the mapping of threads to cores each time a new parallel region is entered. The analysis of the tracing of the mapping event allows 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. However, it is possible that thread migration improves execution times: this is the case for instance when date reuse and L2 cache sharing are less important.

The performance variability of the `swim` application is not the unique case of sensitivity to thread affinity, the other benchmarks exhibit similar behaviour, see the cases of `wupwise, applu, equake, apsi, fma3d, ammp`.

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

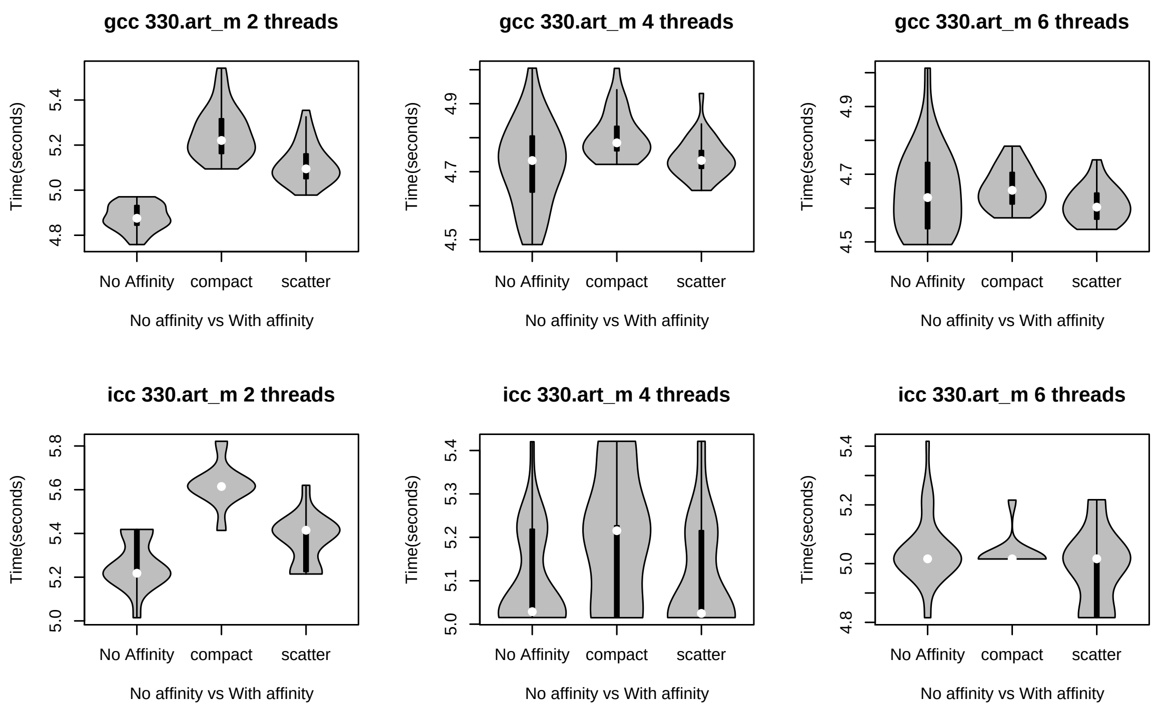


Fig. 6. Observed Execution Times of the `art` Application (compiled with `gcc` and `icc`)

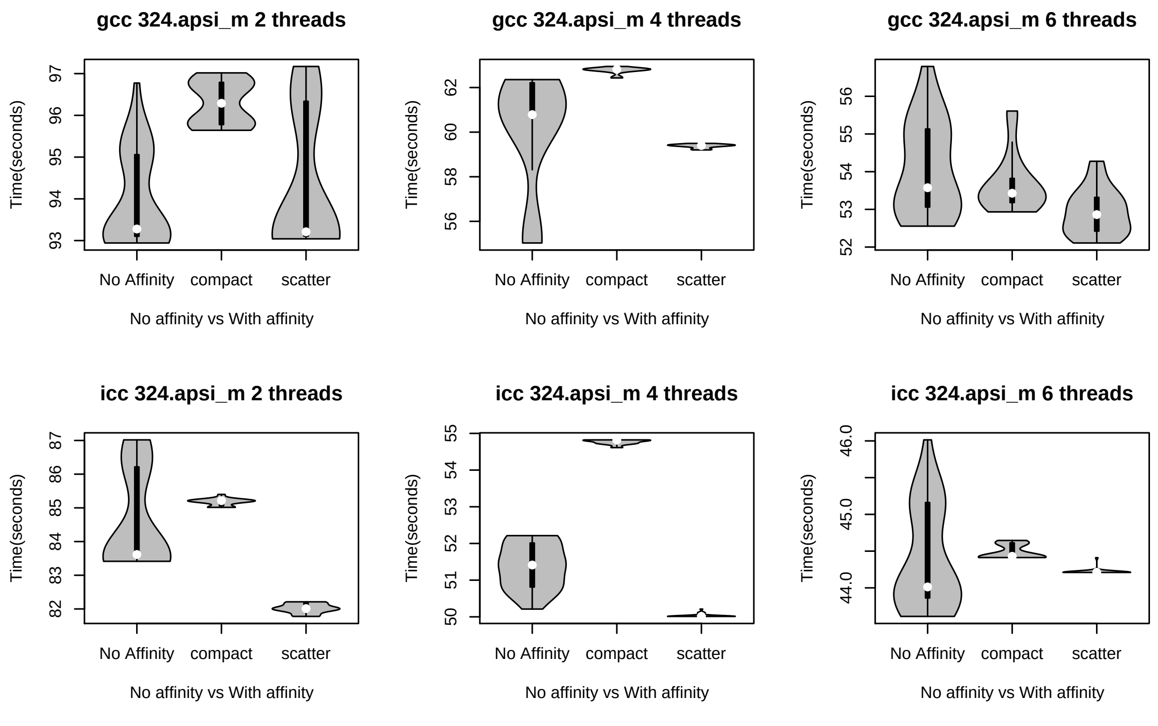


Fig. 7. Observed Execution Times of the `apsi` Application (compiled with `gcc` and `icc`)

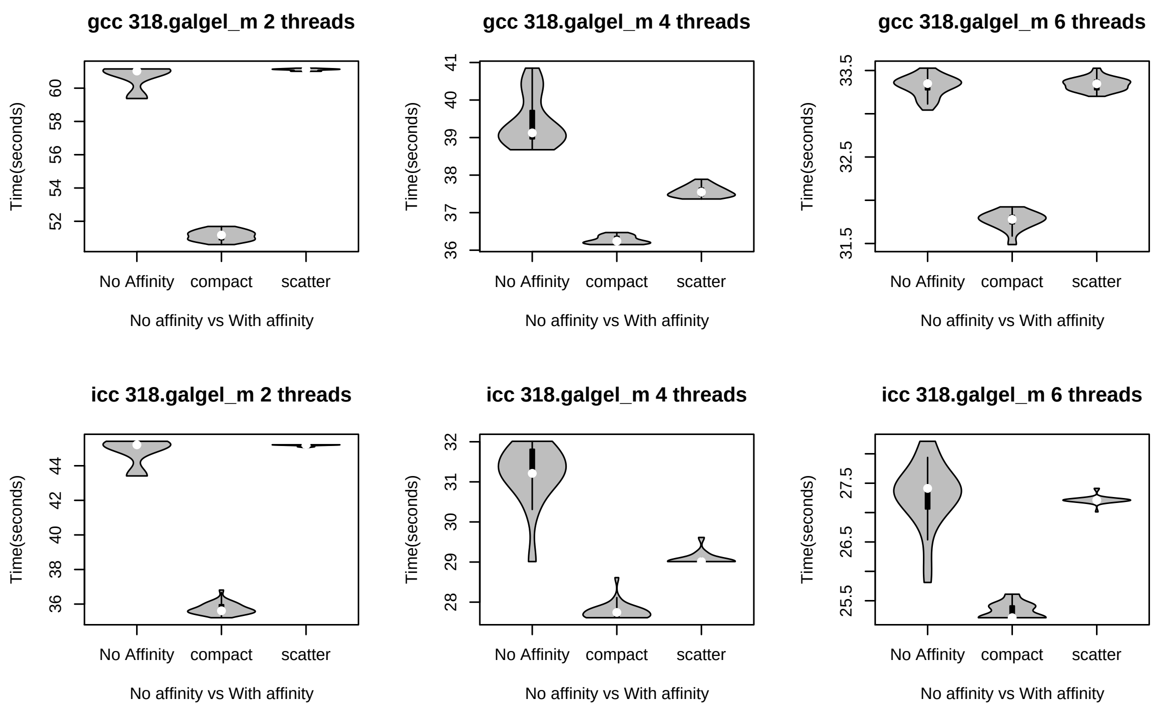


Fig. 8. Observed Execution Times of the `galgel` Application (compiled with `gcc` and `icc`)

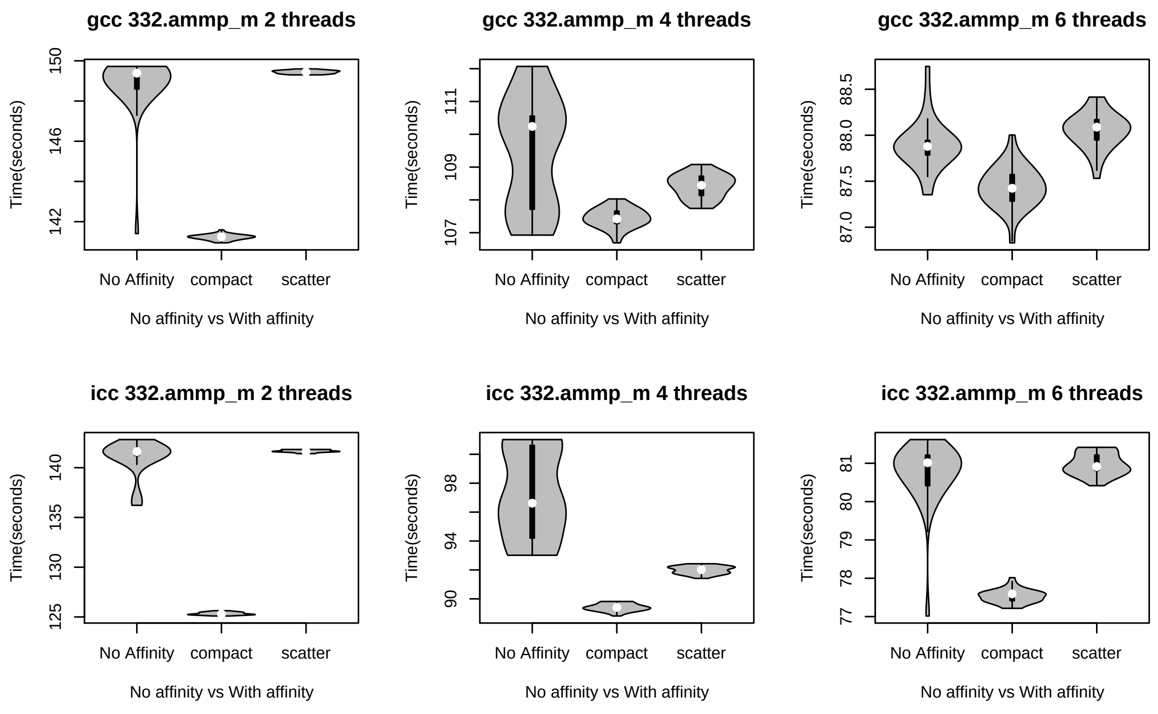


Fig. 9. Observed Execution Times of the `ammp` Application (compiled with `gcc` and `icc`)

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. There are many work in the litterature that tries to find a good thread binding, which is not the purpose of our study here. The next section explores the performane variability when SPEC OMP are executed in parallel with other co-running processes.

4 Analysing the variability of SPEC OMP performances with co-running processes

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 concurrent applications. For each OpenMP benchmark we measure 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).

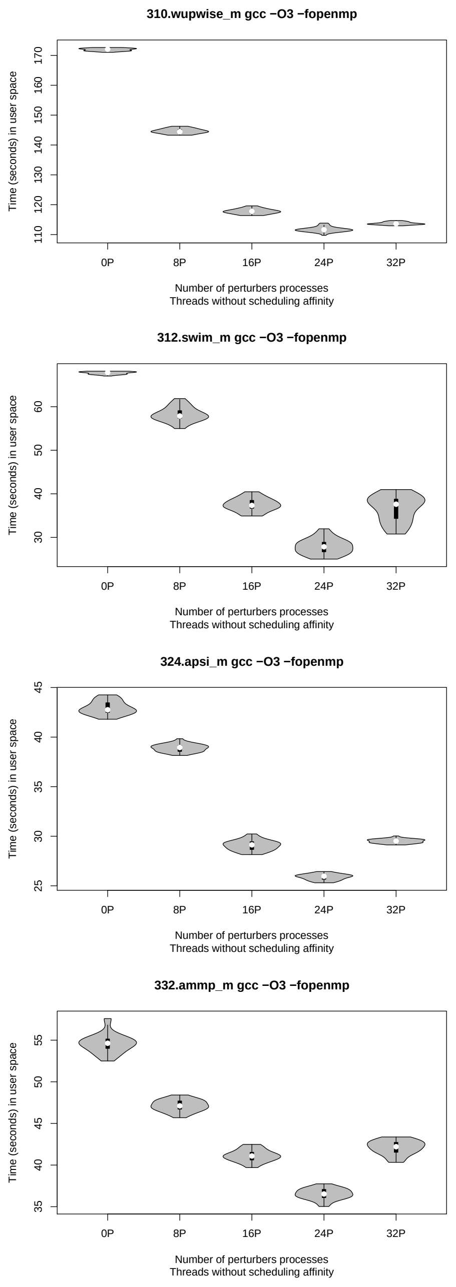
In these experiments we generate a system load by running some artificial co-running processes in background. These co-running processes are launched by one master process executing the `fork` system call a number of times equal to the number of processes that we need to generate at runtime. This number is supplied as an argument to the command line. The code executed by these co-running processes is a dummy non terminating loop without memory access (`do while(1);`).

4.1 Experimental setup

- The SPEC OMP2001 benchmarks are launched with 8 threads at runtime to occupy all the cores of the system.
- Each SPEC OMP2001 benchmark (8 threads) is run either as a single application on the machine (minimal system load) or in parallel with 8, 16, 24 or 32 co-running processes respectively (performance perturbation created in background). This leads to five distinct runtime configurations.
- We report here the results when SPEC OMP2001 and the co-running processes are launched without scheduling affinity to the system cores (no explicit binding of threads on cores). Similar experiences have been conducted when affinity is fixed, the conclusions remain similar.
- 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.

Fig. 10 and Fig. 11 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 together with the co-running processes. The Y-axis represents the 31 observed execution times for each software configuration. In each violin plot, we have still some important performance variability since thread affinity is not fixed. As expected, when the number of co-running processes increase, the total elapsed execution time increases too for each application.

However, a strange phenomenon appears. From Fig. 10, we can see that when we increase the number of co-running processes in background, the program execution times at user level of the OpenMP applications decrease. Meanwhile we record in Fig. 11 an increase of elapsed execution times as expected. Running the threads of the SPEC OMP2001 benchmarks and the co-running processes with a fixed scheduling affinity leads to the same conclusion (not plotted here): when we increase the number of co-running processes, we observe a decrease in program execution times at user level of the OpenMP applications. Why do we have such execution time decrease ? There are multiple factors that may explain such phenomena. For instance, increasing the number of co-running processes may increase the cost of thread synchronisations:



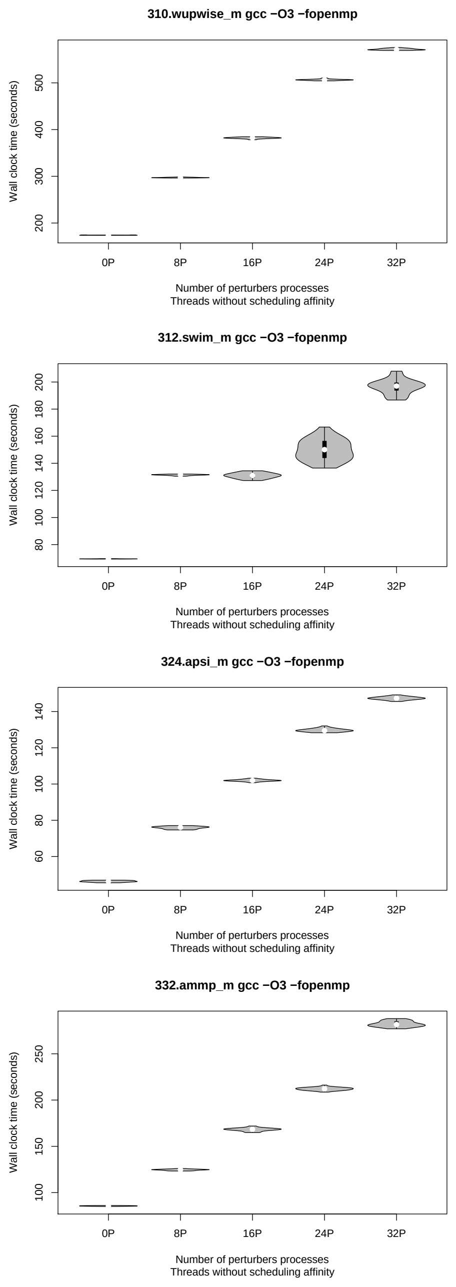


Fig. 10. Observed User Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

Fig. 11. Observed Real Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

the application threads being in sleep mode, the sleep period does not account for user level execution time. In our article, we do not investigate the interaction with the OS, we focus mainly on the interactions between the code and the underlying architecture. In order to make such analysis, we design micro-benchmarks that isolate the phenomena under study (since SPEC OMP 2001 are too complex to analyse at the micro-architectural level). The next section describes our study with the micro-benchmarks.

5 Micro-benchmarking with co-running processes

5.1 Memory-bound micro-benchmarks

The generic code of the micro-benchmarks is given in Listing 1.1. It is composed of three loops `Loop1` (parallel loop), `Loop2`, `Loop3` and one statement `S1`. The `L2` loop is added for repetition purpose to increase the measurement accuracy. The data set accessed by all the micro-benchmarks is equal to $\texttt{N} * \texttt{M} * sizeof(int64) = D\ bytes$ 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 L2 cache size preventing from frequently accessing the DRAM in case of L2 cache misses.

Listing 1.1. 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]++;
}
```

Giving distinct values to the parameters `N` and `M` defines distinct micro-benchmarks that we name mb_N_M. 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 $\texttt{M} * sizeof(int64) = 1.5\ MB$. This 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 $\texttt{N}/8 = 24576$ `N` iterations, the innermost `M`-loops access to a chunk of size $\texttt{M} * sizeof(long) = 64\ B$ (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 have seen that threads sharing common L2 may create cache conflicts, thus data are ejected from L2.

The previous micro-bechmarks make extensive memory operations accessing cache levels. We also used a CPU-bound micro-benchmark, which is a parallel OpenMP implementation of a prime number code. Its code is listed in [7] page 28. We used this CPU-bound code to check the behaviour of performance variability under the influence of co-running processes if less memory operations are performed.

5.2 The co-running processes

We investigated three types of co-running process:

1. CPU-bound co-running processes which are simple non terminating loops (`do while (1);`).
2. CPU-bound with sleeping state: that is, the co-running process makes a call to the `usleep` function with various values (from 10 μs to 10 ms, this later value is the linux kernel time slice). It allows us to study the behaviour of the micro-benchmark when its co-running processes sleep completely or partially.
3. Memory-bound co-running processes: that is, the co-running process makes extensive accesses to the L2 cache to be in competition with the micro-benchmarks.

5.3 The experimental environment

The micro-benchmarks and the co-running processes are executed concurrently under multiple software and environmental configurations:

- The number of threads of the micro-benchmarks is either 8 (to occupy all the cores) or 4 (to test some affinity configurations).
- Affinity is fixed in order to experiment two situations: sharing of L2 cache between threads or not (when 4 threads are used, sharing or not the 4 L2 caches of the system).
- The number of co-running processes per core is varied from 0 to 4.
- We tested the case of active and inactive automatic hardware prefetching.
- We tested two values of memory pages: small size (4 KB) and large size (2 MB).

5.4 Results and analysis

The full results and analysis are given in [7]. This section presents a synthesis. First, we confirm the observation done in Sect.4: given a parallel OMP application, increasing the number of the background co-running processes may decrease the execution time at the user level (while the real execution time increases).

Indeed, our extensive experiments with the different micro-benchmarks and co-running process described in this section, run in different configurations (different affinity, page sizes, disabling/enabling automatic hardware prefetch) allow us to figure out the following explanations:

- The automatic hardware prefetching combined with a small pages size generates L2 cache conflicts, even if enough cache capacity exists to hold all the accessed data. Such L2 cache conflicts explain part of the performance variability observed in our experiments.
- The decrease of the execution time at the user level when we increase the number of co-running processes is due to the following fact: the co-running processes run in competition with the OpenMP threads. Consequently, the threads of the same application run with less competition between themselves, their access to L2 cache is regulated (smoothed) by the co-running processes. In other words, the contention on the L2 cache and on the memory bus is reduced thanks to the co-running process which consume fraction of the CPU time.

6 Related work

Collective optimisation [3] 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* [6]. 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 Student's t-test, 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. [2] 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. [1] 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 [9]. For instance, the experiments on sequential applications reported in [9] show that the size of Unix shell variables and the linking order of object codes both may influence the execution times.

Thread binding is another research subject that has been studied in the literature. For instance, the *autopin* tool [5] tries to find a good affinity between threads at execution time (dynamically). The aim is to have a better execution time of the application. However, the authors did not investigate the variability of the performances: indeed, we showed that thread pinning is important to stabilise performances in most of the cases, but some applications still exhibit non negligible performance variability even when the affinity is fixed.

7 Conclusion

Our extensive experimental study clearly shows 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). Since SPEC OMP does not make any recommendation for thread affinity (since multicore platforms did not exist yet in 2001), 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 article 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 total elapsed 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 competition, 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 executing machine to fix them. Consequently, when an end-user executes an application that is declared optimised, he may 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 [12].

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. Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.
2. Lieven Eeckhout, Hans Vandierendonck, and Koenraad De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *J. Instruction-Level Parallelism*, 5, 2003.
3. Grigori Fursin and Olivier Temam. Collective Optimization. In *The 4th International Conference on High Performance and Embedded Architectures and Compilers (HIPEAC)*, 2009.
4. 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.
5. T. Klug, M. Ott, J. Weidendorfer, C. Trinitis, et al. autopin–Automated Optimization of Thread-to-Core Pinning on Multicore Systems. *Transactions on High-Performance Embedded Architectures and Compilers. Springer*, 2008.
6. Hugh Leather, Michael O'Boyle, and Bruce 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.
7. Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study. Technical report, University of Versailles Saint-Quentin en Yvelines, July 2010. http://hal.archives-ouvertes.fr/inria-00514548.
8. Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Study of variations of native program execution times on multi-core architectures. In *CISIS '10: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems. MuCoCos workshop.*, pages 919–924, Washington, DC, USA, 2010. IEEE Computer Society.
9. Todd Mytkowicz, Amer Diwan, Peter F. Sweeney, and Mathias Hauswirth. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.
10. Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling.* John Wiley and Sons, New York, 1991.
11. Standard Performance Evaluation Corporation. SPEC CPU. http://www.spec.org/, 2006.
12. Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. The Speedup-Test. Technical report, University of Versailles Saint-Quentin en Yvelines, January 2010. http://hal.archives-ouvertes.fr/inria-00443839.