Study of Variations of Native Program Execution Times on Multi-Core Architectures

Abdelhafid MAZOUZ
University of Versailles Saint-Quentin,
France.

Sid-Ahmed-Ali TOUATI
INRIA-Saclay,
France.

Denis BARTHOU
University of Bordeaux,
France.

***Abstract*—Program performance optimisations, feedback-directed iterative compilation and auto-tuning systems [1] all assume a fixed estimation of execution time given a fixed input data for the program. However, in practice we observe non-negligible program performance variations on hardware platforms. While these variations are insignificant for sequential applications, we show that parallel native OpenMP programs have less performance stability.**

This article does not try to quantify nor to qualify the factors influencing the variations of program execution times, that we let for a future work. This article demonstrates three observations: 1) The performance variations of sequential applications is insignificant. 2) OpenMP program execution times on multi-core platforms show important variations. 3) The distribution of the execution times is not a Gaussian distribution in almost all cases. We finish by a discussion explaining why considering the minimal or the mean execution time within a sample of experiments is not the best estimation of program performance.

I. INTRODUCTION

Every computer scientist working on the field of program performance optimisation knows how to compute a speedup. Let P be a program and I an input data set. The execution time of the program P given an input data set is noted by $T(P, I)$. Computing the speedup of a program, given a fixed data input, is easy [2]: if P' is a modified version of P, then the speedup is computed by $s(P', I) = \frac{T(P',I)}{T(P,I)}$. 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 function calls).

Contrary to emulated or virtualised programs (such as Java byte-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 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.

Till now, program variations of large sequential applications such as SPEC CPU are materially neglected. In the SPECPU2006 run rules, we can read: *¨A central idea of SPEC benchmarking is to create tests that are repeatable: if you run a benchmark suite multiple times, it is expected that results will be similar, although there will be a small degree of run-to-run variation ¨*. Indeed, we confirm in this study that large SPECPU2006 applications have minor variations with the train input data. This of course does not guarantee that the variations of sequential applications would always be negligible. For instance code kernels and toy benchmarks have small execution times, consequently the statistical variance may be greater.

With the introduction of multi-core architectures, programming in parallel is expected to take an increasing importance in the future. OpenMP paradigm is one of the most used parallel programming model on desktop machines: it assumes a shared memory between threads, is used with simple pragmas that allows or not the invocation of threads, etc. In this article, we report extensive experiments on a representative sample of OpenMP benchmarks (from SPEC) executed on an Intel multi-core machine. We show that, contrary to sequential applications, the variations of program execution times are sensitive from a human usage point of view. Consequently, considering an good estimation of $T(P, I)$ becomes of crucial importance and impose us to revisit the old performance analysis methodology.

By now, the variability of $T(P, I)$ must be correctly taken into account for auto-tuning systems [1]. Also, it must be considered to report confidence intervals for speedups. This advice already exists in the literature since numerous decades [2], [3] but not really followed in practice: the reason is that the exact definition of the confidence interval of a speedup is not known till yet. As far as we know, a correct formula based on statistical and probability theory that estimates the confidence interval of the fraction $\frac{T(P,I)}{T(P',I)}$ is not defined. And defining such formula is not easy, because the shape of the theoretical distributions of program execution times is not known yet.

Our article is organised as follows. Sect. II presents some related efforts dealing with programs execution times variability. Sect. III presents the heart of our experimental study. Sect. IV develops some discussion before concluding.

II. RELATED RESEARCH ACTIVITY

Collective optimisation [4] 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* [5]. 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 test of student. 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.

Last, program execution times variability has been shown to lead to wrong conclusions if some execution environment parameters are not kept under control [6]. For instance, the experiments on sequential applications reported in [6] show that the size of Unix shell variables and the linking order of object codes both may influence the execution times. In our article, we do not provide explanation about the external factors influencing the variability of program execution times. We fix the execution environment as advised in [6], and we observe the variations that comes from complex interaction phenomena between software and hardware. Identifying such factors is not the topic of this article and we start our effort by analysing the performance variability.

III. Experimental Results

This section presents our synthesis of multiple months of experiments. We describe our experimental setup, then we study the normality of the distribution of program execution times. We then show the weak variability of SPEC sequential applications while confirming the results presented in [6]. Finally, we demonstrate a larger variability of OpenMP program execution times on a multi-core architecture.

A. Experimental Setup

Our experiments have been conducted on a Linux workstation. The kernel version x86_64 2.6.26 is patched with `perfmon kernel 2.81` (`libpfm` and `pfmon` version 3.8). The micro-architecture of the processors is Intel Core 2, quad-core Xeon E5345, FSB 1333. The core frequency is 2.33 GHz. The machine has two chips, each one has four cores. Regarding the cache sizes, the L1 caches are private to each core and have a size of 32 KB for instructions and 32 KB for data (separate). The L2 cache level is shared between each couple of cores. In our machine, there are two L2 caches on each chip. The size of each L2 is 4 MB, for both instructions and data. Inside a chip, two cores share 4 MB of L2. The main memory size is 4 GB and the number of TLB entries is 512.

B. Experimental Methodology

The test machine was entirely dedicated during the experiments to a single user. The OS services were all inactive, except basic ones such as `sshd`. We used the build system and scripts of SPEC CPU2006 and OMP2001 to compile and optimise applications, launch them, measure execution times, check validity of the results and report the performance numbers. The compiler used are `gcc 4.1.3` and `4.3.2`, and Intel `icc 11.0`. The compiler optimisation level we used are `-O2` and `-O3`. The option `--fno-strict-aliasing` was used for `perlbench` benchmark because of a technical error in that code (reported to SPEC and patched for future versions). `gcc` was not able to compile the omp 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 Intel `icc 11.0`). We also disactivated the random dynamic stack allocation (this is an option in a Linux Kernel).

The SPEC system measuring the execution times relies on the function `gettimeofday` that returns the real time with a precision of micro-second.

Numerous software configurations were experimented:

1) For SPEC CPU 2006, we varied the size of the Unix shell environment, as described in [6]. We also experimented two code optimisation levels (`-O3` and `-O2`).
2) For SPEC OMP 2001, we fixed the Unix shell environment, and we varied the number of threads as: no threads (sequential version), 1 thread (omp version with a unique thread), 2, 4, 6 and 8 threads.

The input data used for experiments are the *train* set. The successive executions are performed sequentially in a back-to-back way. No more than one application was executed at a time. We configured the SPEC runspec to perform 30 runs for each software configuration. This high number of runs allows us to report statistics with a high confidence level.

C. Normality Check Results

Each software configuration is executed 30 times, yielding to report 30 distinct execution times of the same binary program and train input data. We checked if these 30 distinct values follow a normal distribution or not. Surprisingly enough, most of the execution times distributions are not Gaussian: Using the standard Shapiro-Wilk normality check, and a high confidence level equal to 95%, the normality check fails in almost all SPEC CPU 2006 applications (except for `462.libquantum`) and in all OMP2001 applications. When

the confidence level is reduced from 90% to 50%, the normality check succeeds in three SPEC CPU applications out of 11 (`462.libquantum`, `482.sphinx3` and `445.gobmk`) and in three OMP applications within 9 (`310.wupwise_m`, `324.apsi_m`, `330.art_m`). The normality check demonstrates that the Gaussian function cannot be used as a general distribution function for program execution times.

D. Variability of Spec CPU 2006 Execution Times

We use the violin plot[1] to report in Fig. 1 the execution times of each application, and for each Unix shell variable size. Here we illustrate four cases with representative variability, the article page size limitation does not allow the full report of all experiments.

The X-axis of Fig. 1 represents the size of Linux shell environment as studied in [6]. For each size, the Y-axis report the 30 execution times. We deduce from all these figures that: 1) we confirm that the size of the Linux shell environment may influence the execution times and 2) when we fix the execution environment, the variations of the execution times are minor (within one second). These observations are valid for all the SPEC CPU 2006 applications that we experimented.

By using the Student's t-test, we also report the mean confidence interval of these benchmarks in Fig. 2. We can see that these intervals are sufficiently tight. We deduce that the sample mean of the execution times of SPEC CPU benchmarks does not vary in a sensitive way.

From the experiments presented in this section, we deduce in overall that the variability of the execution times of all SPEC CPU2006 applications is negligible (less than 1 second) whatever the optimisation level we used (`-O2` or `-O3`). A human user would not feel a difference when he executes multiple times the same program with the same input data.

The next section shows that this situation is not always the case if we use OpenMP applications on a multi-core architecture.

E. Variability of Spec OMP 2001 Execution Times

We use violin plots to report in Fig. 3 the execution times of each application compiled with `gcc`. The Unix environment size was fixed. We chose four applications to highlight that the variability is significant. 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 30 observed execution times for each software configuration. We conclude three 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 deep disparity. Consider for instance the case of `swim` in Fig. 3. The version with 2 threads runs between 76 and 109 s, the version with 4 threads runs

[1]The Violin plot is similar to box plots, except that they also show the probability density of the data at different values.

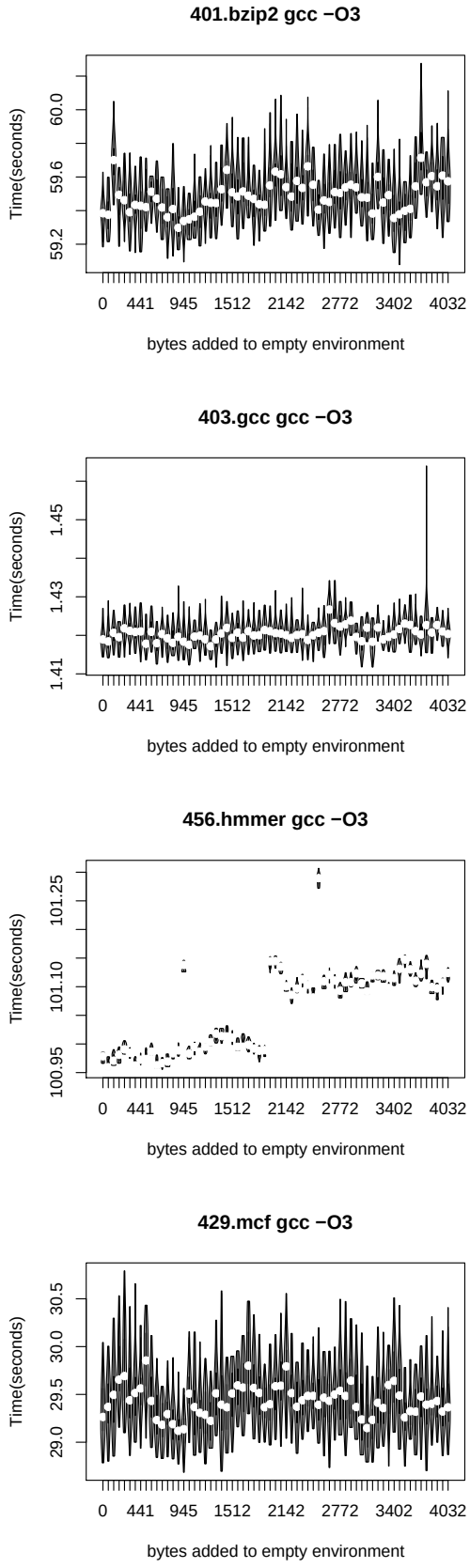

Fig. 1. Observed Execution Times of some SPEC CPU 2006 Applications (compiled with `gcc`)

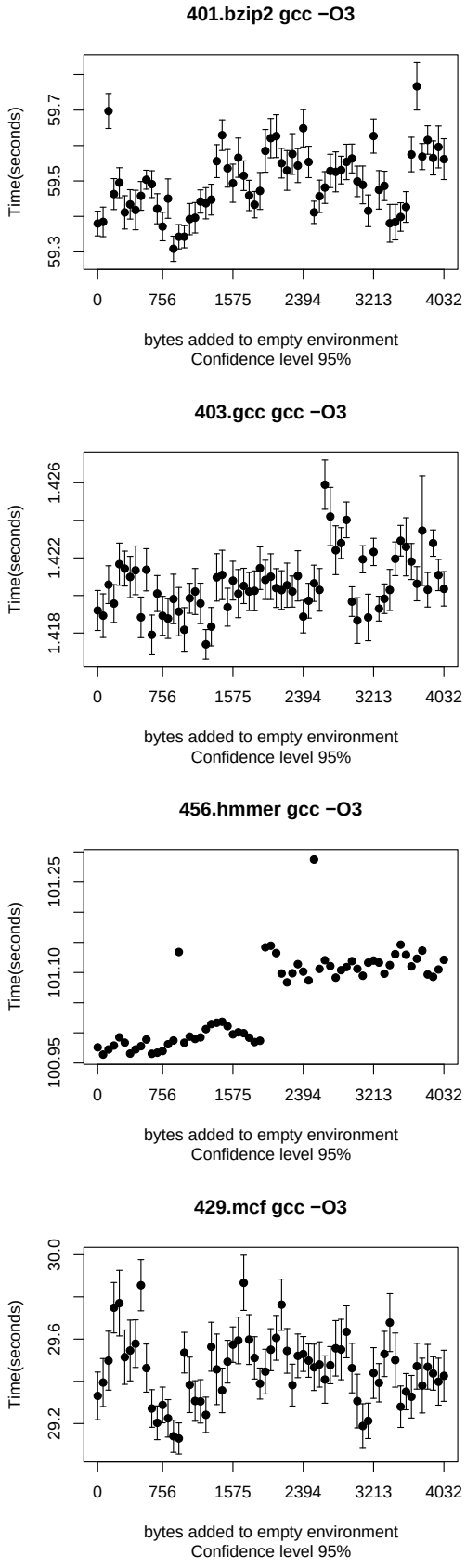

Fig. 2. Mean 95% Confidence Interval of some SPEC CPU 2006 Applications (compiled with `gcc`)

between 71 and 90 s. This variability is also present when `swim` is compiled with `icc`, see Fig. 4. The example of `wupwise` in Fig. 3 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, because as we can observe 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 multi-core 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 Fig. 4: 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.

By using the Student's t-test, we report the mean confidence interval of these benchmarks in Fig. 5. Contrary to SPEC CPU 2006 applications, we can see here that these intervals are not always tight. For instance, the case of `swim` shows a large mean confidence interval, that does not allow for instance to clearly distinguish between the average performance of 4 and 6 threads. Also, the mean confidence interval of `galgel` with 4 threads is [108,112], which cannot be considered as tight for a sample mean. The other applications have tighter sample mean confidence intervals, the question becomes now if the sample mean is a good measure or not for the performance of a program. The next section discusses this matter.

IV. Discussion

Many reported performance numbers in the literature consider the minimal observed execution time within a small sample (between 3 and 10 executions). While it may be argued that the observed minimum can be justified to have an idea of the best observed performance, we think that it is not true from a statistical and computer science point of view:

1) The minimal execution time may be rare, see the case of `galgel` application with 2 threads in Fig. 3. If we consider the min, then we may conclude that the version with 2 threads is better than the sequential version. But we can clearly see that the sequential version is better than the 2 threaded version in overall.
2) The minimal execution time belongs to a distribution of execution times obtained while input data and compiler optimisations do not change. In any case, the execution time variability, accounting for this minimum time, is not yet under control and has to be taken into account in measurements of speed-ups.

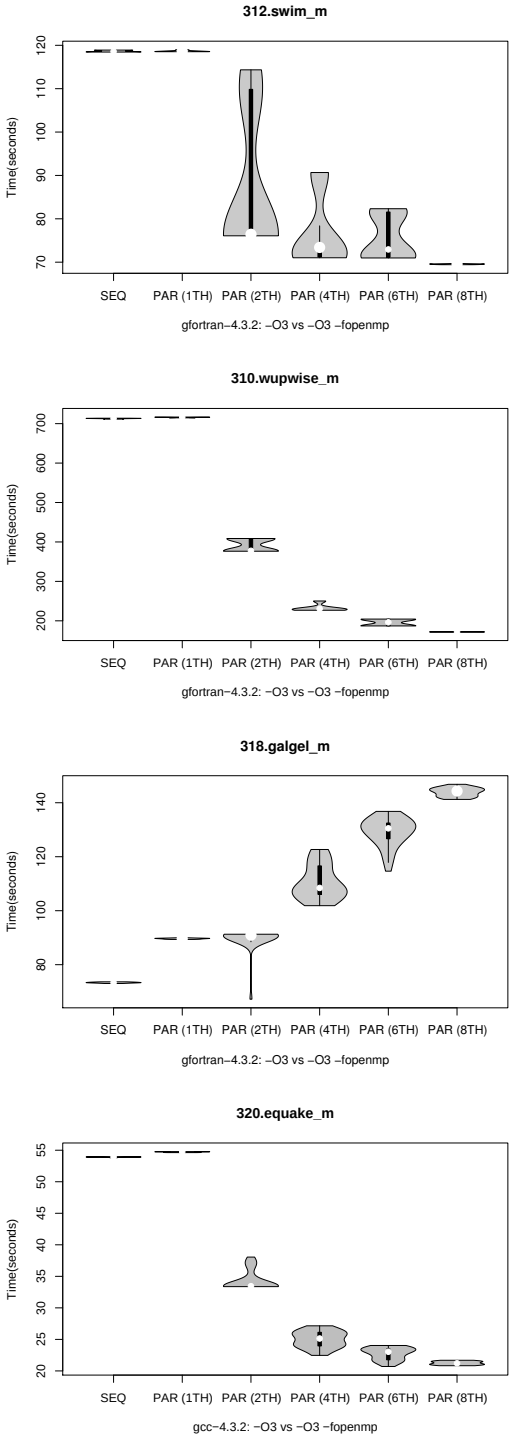

Fig. 3. Observed Execution Times of some SPEC OMP 2001 Applications (compiled with `gcc`)

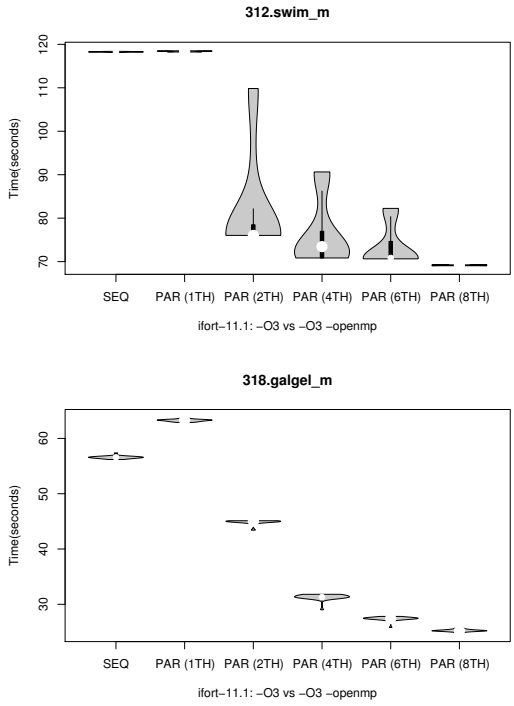

Fig. 4. Observed Execution Times of `swim` and `galgel` compiled with intel `icc`

3) In statistics theory, estimating the confidence interval of the mean and the median is based on the central limit theorem, allowing to have a correct estimation with an arbitrary confidence level. However, it is well-known that the estimators of extreme values like minimum and maximum are hard to define correctly. In other words, considering the minimum of a sample of experiments does not allow to compute a confidence interval if the variability is important. Consequently, the minimal value may have a huge variation when we consider distinct samples. This variation of the minimum when we consider multiple samples is a deep disadvantage for reproducibility and check of the results.

As we studied in Sect. III, we showed that the distribution of the execution times have bias and important variations in the case of OMP applications. Consequently, the mean of the observed execution times is not a good choice to report a unique value for the execution time. We confirm then the advices of [3], [7] to use the sample median instead of sample mean to report execution times.

V. Conclusion

In this article, we report the experimental results of numerous months of benchmarking. We considered the applications of SPEC CPU2006 and SPEC OMP 2001. We used two compilers (`gcc` and `icc`) with two optimisation levels (`-O2` and `-O3`). We varied the Unix shell environment and the number of threads. We also desactivated the randomisation of the starting address of the stack option. We considered

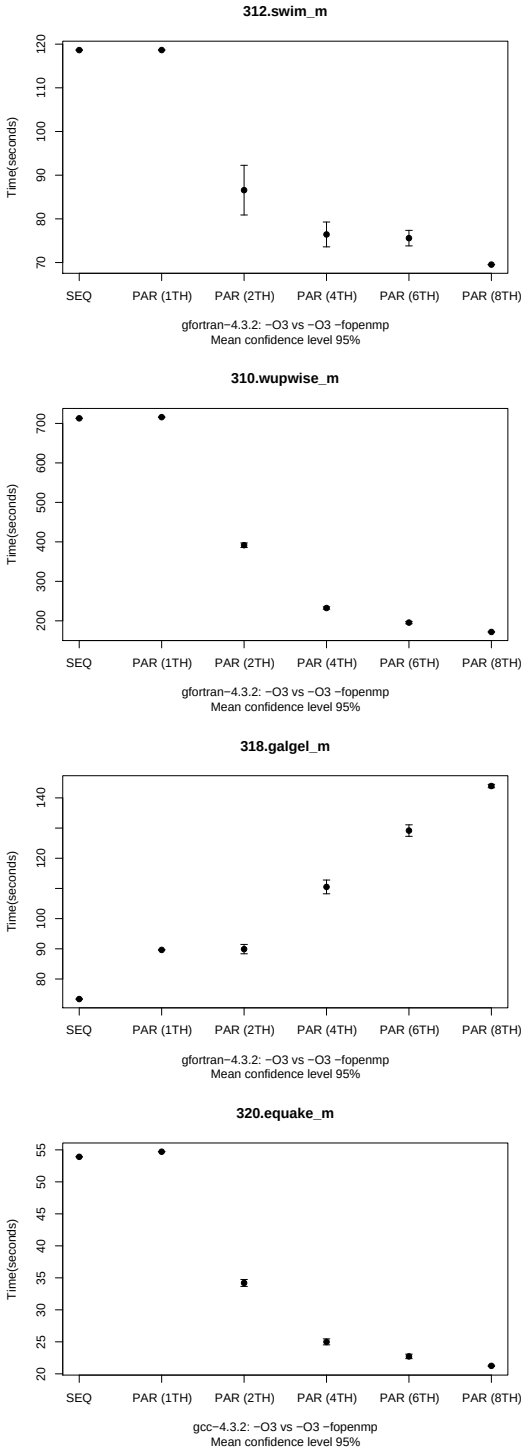

Fig. 5. Mean 95% Confidence Interval of some SPEC OMP 2001 Applications (compiled with `gcc`)

the train data input. For each combination of these software configuration, we conducted 30 runs on an 8 cores Linux machine allowing to have a high confidence level for our statistics [3].

Our experiments demonstrate that, contrary to sequential SPEC CPU applications, OpenMP applications have an important variability of execution times. Dealing with this variability will be an increasing challenge for auto-tuning systems [1]. We showed that the distribution function of the execution time does not follow a Gaussian function in most of the cases. We also showed that the distribution functions have bias, asking as to revisit the classical speedup evaluation. We made a discussion to give arguments against estimating the minimum and the mean, and we advise to consider the observed median execution time as proposed in [3], [7]

Our future work is oriented towards two directions: 1) Determine and quantify the factors influencing the variability of program execution times. 2) Study the distribution function of program execution times.

ACKNOWLEDGEMENT

This research result has been funded by the DIGITEO foundation (ANAPERS project, contract number 2008-12D).

REFERENCES

[1] T. Karcher, C. Schaefer, and V. Pankratius, "Auto-tuning support for manycore applications: perspectives for operating systems and compilers," *SIGOPS Operating System Review*, vol. 43, no. 2, pp. 96–97, 2009.

[2] John L. Hennessy and David A. Patterson and David Goldberg , *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002, ISBN-13: 978-1558605961.

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

[5] Hugh Leather and 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.

[6] Todd Mytkowicz and Amer Diwan and Peter F. Sweeney and Mathias Hauswirth, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009.

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