Runtime variability in scientific parallel applications

Wim Heirman, Joni Dambre, Dirk Stroobandt, Jan Van Campenhout
ELIS Department, Ghent University, Belgium
wim.heirman@elis.ugent.be

***Abstract*—Simulation remains an important component in the design of multicore processor architectures, just as in uniprocessor design. In contrast to single-threaded applications, however, many multi-threaded programs are not deterministic: in multiple runs, even on the same architecture, different execution paths can be taken. This results in variability of simulation results, which is a well-known phenomenon in real systems, but is nearly universally ignored in simulation experiments. In this paper, we review existing work on simulation variability. We extend this work, which has been focused mainly on commercial workloads, and show that it also applies to scientific workloads. We characterize variability for the SPLASH-2 benchmark applications, and look at how variability can impact the optimization of the interconnection network. Both previous and our own results show that studies aiming to prove the optimality of architectural or other modifications should keep this variability in mind, and make sure that observed improvements are statistically valid in relation to the inherent variability to avoid drawing the wrong conclusions. Although this problem is in no way solved to satisfaction, we review some possible solutions, such as the use of statistics, using different types of metrics, and sample-based simulation.**

***Index Terms*—Simulation, Variability, Parallel processing, Multiprocessor interconnection**

I. Introduction

Due to several problems prohibiting further performance increase in uniprocessor designs, multi(-core) processor architectures are becoming more and more prevalent. The exponential increase in performance we have come to expect can in this way be continued, provided applications can be parallelized effectively. Just as for uniprocessor designs, exploring the performance of different design variants of a multicore processor will at some point in the design have to rely on execution-driven simulation, which is the only method that allows the most accurate measurements to be made. In single-threaded programs, such as the commonly used SPECint and SPECfp benchmarks, the control flow is based entirely on input data, which is fixed between simulation runs. For multi-threaded programs, however, this is no longer the case. For instance, the outcome of race conditions on synchronization variables can be altered, by slightly speeding up one thread or delaying another. This means that, in different simulations, a different processor will be allowed to move into the critical section first, causing a bifurcation in the dynamic instruction streams that can propagate throughout the execution. Methods of load balancing such as global task queues can cause this effect to have grave repercussions, since tasks may be claimed by different processors, altering the load balance in the program significantly.

A slowdown of one of the processors by just one clock cycle is enough for this to happen. It can easily be caused by several effects, including interference by background processes, or by even minimal changes in the architecture. In Java, even sequential programs are inherently non-deterministic due to run-time variations in just-in-time (JIT) compilation or garbage collection, requiring a statistically rigorous approach if accurate performance evaluation is desired [1]. Concerning the simulation of parallel architectures, Alameldeen and Wood [2] analyze the non-deterministic behavior of commercial on-line transaction processing (OLTP) workloads. They show that execution times can deviate by up to 9% between simulations. When subtle improvements in the architecture, compiler or operating system are to be measured, this variability can easily drown out the actual improvements. This can lead to the wrong conclusions being drawn: an architectural modification may seem to improve performance, when this improvement was actually due to random variations!

This behavior is especially apparent in commercial workloads. They are request-driven, the workload needed to complete each request can have large variations. Therefore, load balancing is required to evenly divide requests and their resulting workload among processors. However, several important scientific and technical workloads also use load balancing, so the same effects will play there. And even in programs without load balancing, the different outcome of synchronization race conditions can still introduce runtime variations. Finally, downscaling benchmark input sizes, as is commonly used to reduce simulation times to reasonable values, enlarges the problem, since less samples are being taken.

Sampling techniques, used to limit the runtime further, can have an even worse effect.

In this paper, we measure the variability of the SPLASH-2 benchmarks, which are commonly used in multiprocessor evaluation. We show that the variability in total program runtime is rather high, and can easily reach 10%. Still, program runtime, or an average number of transactions per second for OLTP and web server workloads, are used most often as performance metrics, probably because they are easy to measure (also on real systems), and because they provide an absolute measure that can be understood by the end-users of a machine – who would like to know how long their applications will run, or how many concurrent visitors their website will support. The variability in these high-level metrics is, however, high enough to drown out the improvement made by several important architectural modifications. One solution can be to use a (well-chosen) low-level performance metric, such as average miss latency. They often exhibit less variability, or show a larger performance delta, which makes them capable of distinguishing – in a statistically valid way – between architectural modifications with higher versus those with lower performance. In simulation especially, where the obvious solution of running longer benchmarks or doing multiple measurements to characterize and average out variability, is not a viable alternative, some solution will have to be found.

The remainder of this paper is organized as follows: Section II provides an overview of existing work on variability in (mainly commercial) parallel programs. From Section III onwards, we extend this work towards scientific applications. Section III itself describes our methodology, including the simulation platform and describes how variability was introduced into our simulations. Section IV shows the results of our measurements and analyzes their variability. In Section V we study the impact of variability on fidelity and relative accuracy, or the ability to quantize improvements made by architectural modifications. Section VI looks at how the problem of variability can be coped with, while Section VII summarizes our conclusions.

II. Existing work

A. Non-determinism in commercial workloads

The first to consider variability in architectural simulations of parallel programs were Alameldeen and Wood [2], [3]. They state that *variability is a well-known phenomenon in real systems, but [it] is nearly universally ignored in simulation experiments.* Accurate architectural simulations, required when microarchitectural research is performed on large multiprocessors, use a detailed simulator such as Simics/GEMS [4], [5] in which the ratio of simulation time versus simulated time can be up to one million. Due to this enormous slowdown, the common approach of averaging over multiple measurements is not very practical. Moreover, since the simulator itself is usually deterministic, extra provisions must be made such that the variations, visible in real systems due to different initial states and small external perturbations, are actually visible in simulation.

To this end, Alameldeen and Wood add artificial level-2 cache latency that is uniformly distributed between 0 and 4 ns. Although the average increase in miss latency is therefore always 2 ns in each simulation, the effects of load balancing, scheduling and synchronization as described before, cause the execution time to deviate by up to 9% between simulations. When the performance of two architectures or system configurations is to be compared, this variability can often be as big as the (average) performance difference between both architectures. This makes it easy to derive the wrong conclusion, i.e., "architecture A is better than architecture B," while in reality B is better but the simulation of A happened to be faster because random variations reduced the synchronization times. They define the wrong conclusion ratio (WCR) as a percentage of comparison pairs that reach an incorrect conclusion, which can in realistic situations be as high as 30%.

B. Coping with variability

To cope with this variability, Alameldeen and Wood suggest that each researcher should perform multiple simulations, in which divergent behavior is triggered using a method such as adding random cache latency. From these multiple results, variability can be measured which allows one to construct a confidence interval around each measurement. Also, they argue in [3] that one should move away from measuring the instructions executed per clock cycle (IPC), since this metric has too much variability, and use application level metrics such as program runtime or time per transaction instead.

Wenisch et. al. [6], on the other hand, measure the number of *user* instructions per clock cycle (U-IPC), and find that *it* has a lower variability than transaction throughput. Moreover, they find a linear relationship between both – showing that a (short) measurement of U-IPC can reliably replace a much longer measurement of transaction throughput. In [6] they also propose a methodology for statistical sampling of multiprocessor simulations. This allows a technique called *matched-pair sample comparison*, which will be discussed later.

Lepak et. al. [7] introduce a methodology that can, under some assumptions, remove the variances intrinsic to non-deterministic workload. This way, IPC can be measured fast and reliably – although the method is not applicable for all benchmarks and all types of architectural studies.

C. Non-determinism in scientific benchmark suites

Woo et. al. [8] mention the non-determinism of some of the benchmarks in their description of the SPLASH-2 benchmark suite. Since they want to measure their benchmarks' inherent program properties, such as data sharing and communication-to-computation ratio, they use a fixed timing model that should not trigger the non-determinism. While this approach is good for measuring fundamental, architecture independent properties, it does not point out how variability can be coped with when architecture-induced performance changes are to be measured.

PARSEC is a much newer benchmark suite for the evaluation of chip-multiprocessors, which aims to be a more up-to-date replacement for SPLASH-2. It is described by Bienia et. al. in [9]. They recognize Woo's argumentation that fundamental program properties can be measured accurately without a timing model, avoiding non-determinism, but they also refer to Alameldeen's work and use the same approach to characterize variability in PARSEC's inherent properties. They claim these measurements remained constant within 0.04%, except for two (out of twelve) benchmarks in which much higher deviations were observed.

However, these results were obtained using the largest data set, `simlarge`, which on a real machine executes in about 15 seconds. Since they used a cache simulator based on dynamic instrumentation, the resulting simulation times were acceptable. Microarchitectural researchers, requiring cycle-accurate simulation tools such as Simics/GEMS, face simulation times that will be longer by several orders of magnitude. This, in practice, limits them to the `simsmall` data set which has a native runtime of about one second. By how much this shorter data set would increase variability is yet to be investigated. Moreover, these measurements again concerned *inherent* properties, which are expected to be architecture-independent. Variability of architecture-dependent properties, such as performance metrics, is not necessarily of the same magnitude.

III. Methodology

A. Multiprocessor architecture

For our own experiments on the variability of the SPLASH-2 benchmarks, we assume a 16-processor distributed shared-memory architecture with hardware-enforced cache coherence using a directory-based protocol. The architectural modifications we will study pertain to the interconnection network. We look at the performance of three different networks: 4×4 mesh and torus networks, and additionally a reconfigurable network as described in [10]. This network adds eight extra links to the torus topology, that are reconfigured every millisecond to alleviate congestion on those parts of the network that carry the most traffic at that point in time.

B. Simulation platform

Our simulation platform is based on the commercially available Simics simulator [4]. It was configured to simulate a multiprocessor machine resembling the Sun Fire 6800 server, with 16 UltraSPARC III processors clocked at 1 GHz and running the Solaris 9 operating system. Since we focus on performance of the interconnection network, in-order processors are used to limit simulation times: all executions execute in a single clock cycle, except for memory accesses, which are simulated realistically using two levels of caching and an interconnection network for remote memory accesses and coherence traffic. The coherence controllers and the interconnection network are custom extensions to Simics. They model a full bit-vector directory-based MSI-protocol and a packet-switched network with contention and cut-through routing.

A first-touch memory allocation was used that places data pages of 8 KiB on the node of the processor that first references them. Each thread is pinned down to its own processor using the Solaris `processor_bind()` system call. This way the thread stays on the same node as its private data for the duration of the program which reduces communication, and already avoids most of the non-determinism caused by the scheduler. More details on this simulation platform can be found in [10].

C. Benchmarks

The SPLASH-2 benchmark suite [8] consists of a number of scientific and technical applications using a multi-threaded, shared-memory programming model. Thread creation and synchronization are done using the UPC PARMACS [11] macro's, employing the `solaris.threads` threading model. Because some of the default benchmark sizes were too big to simulate their execution in a reasonable time, smaller problem sizes were used (see Table I). Only the parallel part of each benchmark is included in our measurements. To make sure that the measured variability was not caused

Benchmark	Input size
barnes	8192 particles
cholesky	tk15.O
fft	256K points
fmm	8192 particles
lu	512×512 matrix
ocean.cont	258×258 ocean
radiosity	room
radix	1M integers, 1024 radix
raytrace	teapot
water.sp	512 molecules

Table I
SPLASH-2 BENCHMARKS AND INPUT SIZES USED IN THIS STUDY

by the interference of background tasks, we verified that no other processes were active on our simulated machine during execution of the benchmarks.

Since our scaling down of the problem size influences the working set, and thus the cache hit rate, the level 2 cache was resized from an actual 8 MiB on a real UltraSPARC III to 512 KiB. Also, the associativity was increased to 4-way (compared to 2-way for the US-III) after we experienced excessive conflict misses in Solaris' internal structures with the 2-way caches. Overall, this resulted in realistic, 93–97% hit rates for the L2 caches. 50–60% of L2 misses were cataloged as coherence misses (resulting in communication among different processors), the remaining 40-50% were cold, conflict or capacity misses.

D. Introducing variability

To measure runtime variability, we repeated the simulation of each benchmark a number of times. This in itself is not enough, since the simulator used is totally deterministic. Instead, we delay the starting of the benchmark for a small amount of (simulated) time, by running a number of UNIX `ls` commands before starting the benchmark. This happens even before the initialization phase of the benchmark, so it is not included in our measurements which only begin at the start of the parallel phase of the benchmarks (caches are also not simulated until this point). In theory, no difference should thus be observed, because we run the same benchmark on exactly the same simulated architecture. However, this small delay causes the machine to be in a slightly different state: process identifiers, various usage counters, and internal timers have all been incremented and will take less time to overflow – changing the points in time when subtle interferences to the benchmark (process switches, virtual memory management, etc.) occur.

In contrast to [2], who by changing the latency of each L2-cache miss, keep adding variation throughout the simulation; we on the other hand introduced variation only by changing the initial conditions. The fact that we still observe a significant runtime variability, even without changing the architecture, shows that this effect is fundamental to the parallel programs that were used. A simulation that does introduce architectural modifications will thus certainly have to cope with the same conditions.

IV. MEASURING VARIABILITY

Using our simulation platform described in Section III, ten simulations are performed for each of the SPLASH-2 benchmarks on each of the three types of interconnection networks. For each of them, we measure a number of performance metrics. Figure 1 plots the results, showing the average, standard deviation, minimum and maximum measurements. All metrics have been scaled to the average recorded on a mesh network for the same benchmark.

The top plot shows the total program runtime of the parallel phase, for each of the applications. Clearly, a large variability is present there, which is often as large as the improvement that is made by using a better interconnection network. Next, the average instructions per clock cycle (IPC) is shown (totaled over all processors). Note that since we studied changes in the interconnection network, an in-order processor model was used in which each instruction executes in a single clock cycle, except for memory references. The IPC measured here is therefore solely dependent on the memory access latency, which in turn depends on cache performance and network speeds and congestion. Again, a variability is present that can be of the same magnitude as the average improvement between architectures. The third and fourth plots show the number of executed instructions, in total and in user mode only, respectively, summed over all 16 processors. Those in itself are not performance metrics, but these measurements clearly shows that different execution paths are taken in each simulation, even though the benchmark, operating system and simulated architecture are kept the same in each case. For most of the benchmarks, the number of user instructions executed is relatively constant, showing that most of the variation is in the number of instructions executed in kernel mode – most of which are synchronization instructions. A notable exception is `raytrace`. According to [8], it is implemented using distributed task queues with task stealing. This clearly makes for large variations in program flow.

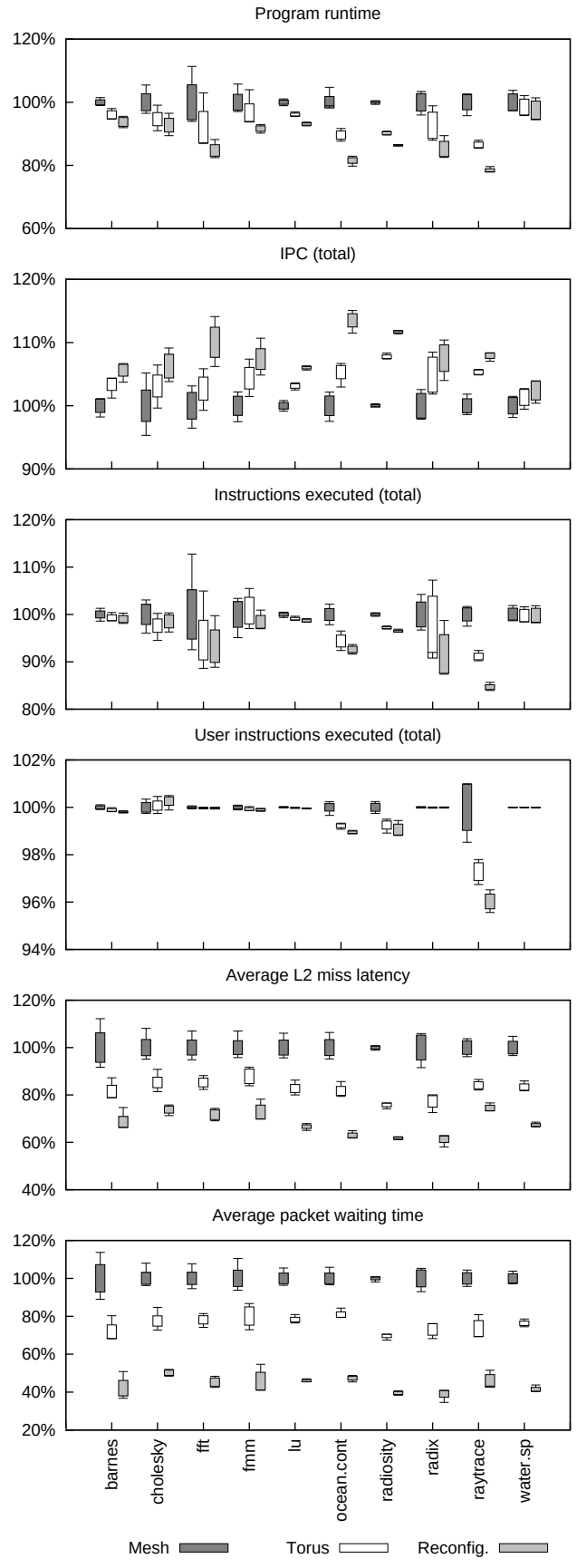

Figure 1. Variability of possible performance metrics for the SPLASH-2 benchmarks, for ten measurements each, on three different interconnection networks.

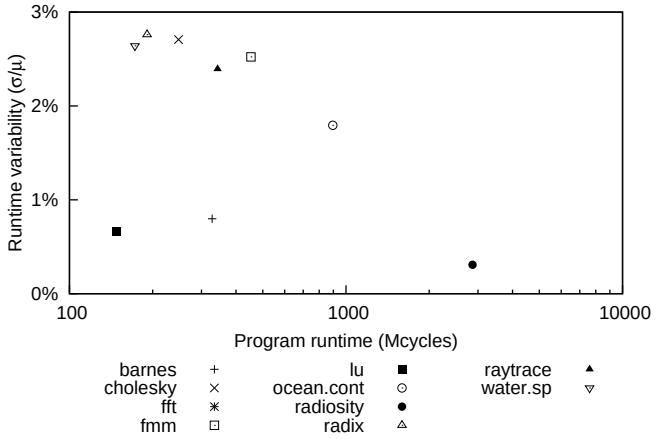

Figure 2. Runtime variability as a function of average runtime: short benchmarks do not necessarily have more variation than long ones.

The next graphs show low-level performance metrics pertaining to the memory system and interconnection network, who's performance were the objective for this study. The fifth plot shows the average level-2 miss latency, pertaining to those memory accesses going through the interconnection network. Again, variability is present, although its magnitude is now in most cases much smaller than the average improvement. The same observation holds for the last graph, which shows the average time network packets spend in a buffer, waiting for a network link to become free – this value largely determines packet latency and is a measure for the congestion on the interconnection network.

When comparing the various benchmarks, significant differences in variability can be observed. These can usually be traced back to the source code. For instance, `radiosity` computes an equilibrium distribution of light, using an iterative algorithm. The speed of convergence, and thus the number of iterations that are needed, depend on the exact order certain computations are made. This is again affected by the relative timing of the different threads, which is not deterministic. The total program runtime can therefore vary to a large degree. This is visible in Figure 1: the number of user instructions executed changed, showing that there was not just variation in synchronization times, but also in the computational part of the application. `ocean.cont` on the other hand requires few synchronizations, and thus experiences less variability. Another class of applications is represented by `water.sp`. It has a low communication-to-computation ratio, and thus is not affected much by changes in the network topology. Its average runtime therefore changes little between architectures, but still very much so between runs. Memory access latencies do change between architectures.

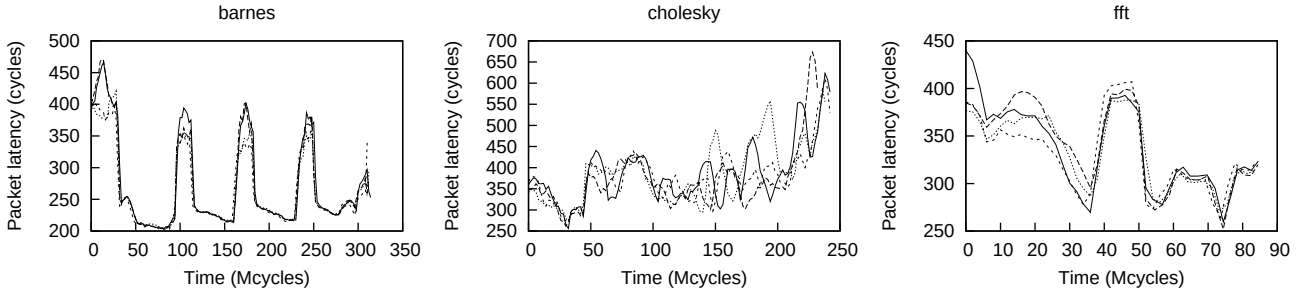

Figure 3. Illustration of divergence among different simulation runs: packet latency (moving average over 10M cycles), for four simulations with the same architecture but different initial conditions.

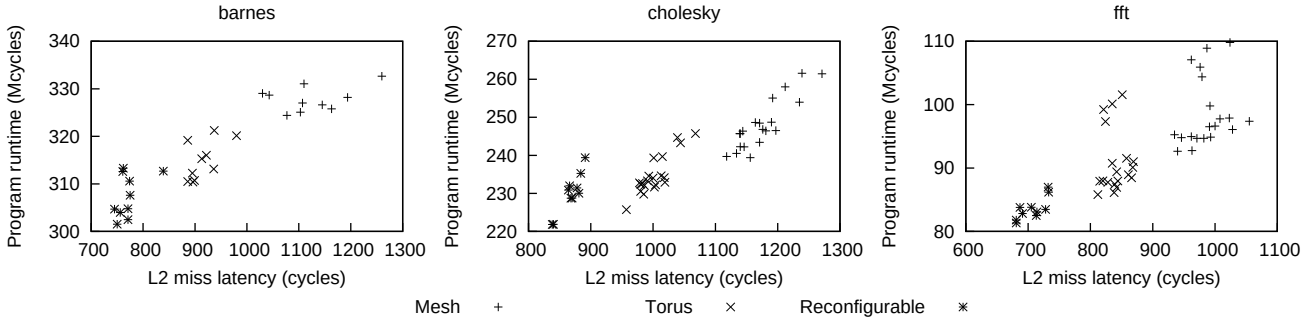

Figure 4. A network is only statistically significantly better if its improvement of a performance metric is more that the variability of that metric. Program runtime alone cannot make this distinction, L2 miss latency can.

Figure 2 plots the benchmark's variability versus its average runtime. In general, there is not much correlation between both, so variability is clearly dependent on the benchmark's algorithm, not just on input size. Still, for (much) longer programs we would expect them to have less variability, and at least for `radiosity` this is indeed the case.

In Figure 3 an illustration is given of the divergent behavior of different simulation runs. All use the same simulated architecture and initial conditions, but now the placement of threads on the different processors was changed – influencing the communication cost between different threads. The graphs plot a moving average of network packet latency. For `barnes`, divergences seem to be concentrated into events about 60M cycles apart. These can be traced back to different stages in the benchmark, each with different communication requirements, and followed by a global synchronization step. `cholesky` and `fft`, on the other hand, have much finer grained synchronization, and thus more points at which variations can occur.

Usually, performance measurements are done to compare different architectural implementations. For instance, in this paper we compare the performance of a mesh, torus and reconfigurable interconnection network. Figure 4 shows all simulation results for some of the benchmarks (ten measurements for each benchmark, network combination), plotting two different performance metrics – program runtime and L2 miss latency – for the three types of network topology. Projection onto the Y-axis, that is, looking only at program runtime, results in an almost complete overlap of the *mesh* and *torus* clusters for `cholesky`, the same thing happens with the *torus* and *reconfigurable* networks with the `fft` benchmark. Clearly, even multiple measurements of program runtime would not allow an accurate comparison of the different networks. On the other hand, when looking at the L2 miss latency, such a distinction is possible since there is a clear separation between the clusters.

Overall, we can conclude that some low-level metrics have less variability than the total program runtime. This makes them more reliable for quantizing architectural performance improvements. Moreover, they provide more insight into the reasons for an increase or decrease in performance. On the other hand, they do not really relate to the *user experience* such an architecture will provide, which is more easily expressed by metrics such as program runtime, frames per second (for visual applications or games), or queries per second (web or database servers). Also, some architectural improvements do not help overall performance much, as is the case for network improvements when only

applications with a low communication-to-computation ratio (such as the `water.sp` benchmark) are run – this effect will not show up on a low-level metric. Still, the variability in some high-level performance metrics, in the context of multicore and multiprocessor architectures, can sometimes be such that their use is limited to a mere illustration of improvements, rather than providing a solid base for making scientific comparisons.

V. Fidelity and relative accuracy

When designing an experiment that is to compare the performance of architectures, two properties can be of importance. They relate to how the outcome of the experiment, the *measured value* of for instance program runtime or average miss latency (which are influenced by variability), compares to the *actual performance* of the architecture (which may be defined as the total runtime of a given benchmark, averaged over an infinite number of executions to cancel out the variability).

One property is *fidelity*: is the experiment able to answer the question "is architecture A better than architecture B?" For this, the experiment needs to provide a metric that changes monotonically with actual performance, i.e., when the performance of A is higher than that of B, the experiment always needs to yield a value for A that is higher (or always lower) than the measurement for B. In a noisy environment, where the magnitude of the noise is as high as, or higher than the absolute difference between both architecture's performances, this may not always be the case!

On the other hand, ignoring the presence of variability for a moment, the low-level metrics in Figure 1 can provide a good fidelity with respect to program runtime: a network with a lower miss latency always results in a shorter program runtime. Fidelity thus does not require a one-to-one relation between measured value and actual performance. Any monotonic relation, even a non-linear one, suffices. In this case, an appropriate low-level metric such as L2 miss latency, due to its lower variability, may provide better fidelity of an architecture's performance, as expressed by the total runtime of a set of benchmarks, than by measuring runtimes directly.

Another important property is *relative accuracy*, this is the accuracy to which the experiment can predict the magnitude of changes in the actual performance between architectures. While an experiment with good fidelity can only say that architecture A is better than B, an experiment with high relative accuracy can reliably say "how much better." Here, a linear relation between measurement and actual performance is desired. A systematic error is allowed though, as this will only reduce

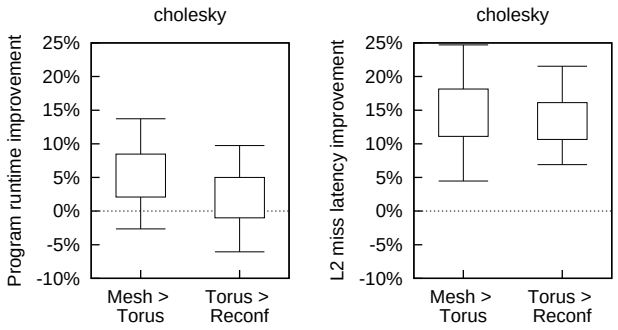

Figure 5. Measured relative improvements of program runtime (left) and L2 miss latency (right) for `cholesky`.

the *absolute accuracy* but leaves relative accuracy intact. Since the changes in performance are usually not very high, however, a variability that has a small magnitude compared to the absolute performance may suddenly become very big when compared to an equally small change in average performance. Variability can thus be devastating for relative accuracy.

This is shown in Figure 5, which plots the relative change in program runtime (left) and in L2 miss latency (right) when changing the network topology, for the `cholesky` benchmark. Since we have 10 measurements for each network, there are 100 combinations that allow us to compute a measured improvement between two network architectures. Out of these 100 combinations, we plotted the average, the standard deviation ($\mu \pm \sigma$), and the minimum and maximum measurements. Note that the variability in program runtime, for one network architecture, was only about 5%. But comparing two entities that are, on average, only 5% apart, and each have a 5% error, results in a terribly low relative accuracy! In this graph one has the luxury of averaging over several simulation runs, it is therefore visible that, when going from mesh to torus or from a torus to a reconfigurable topology, there is an increase of about 5% or 2% in program runtime, respectively. However, this conclusion would not have been possible using just one simulation for each network: in such an experiment the relative change could be anywhere between -3% and +14% for the mesh to torus case. This range is not much in the way of relative accuracy. Worse, the fact that, if the initial conditions of both simulations proved unfavorable, a negative number can be found, shows that a single measurement of program runtime cannot even provide the fidelity required to compare mesh and torus networks (this can be seen by the fact that the number 0 is in the confidence interval: the outcome can be both positive or negative). The L2 miss latency does a lot better here: its variation is much lower, and the sign of

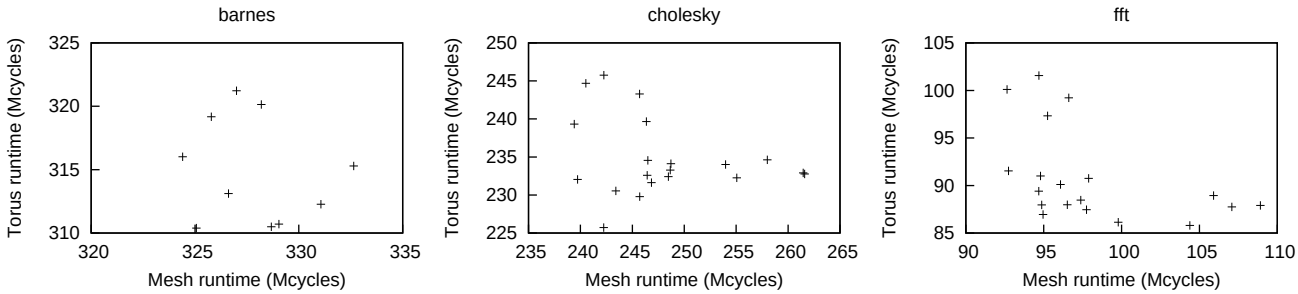

Figure 6. Comparison of runtimes on mesh and torus networks, for pairs of simulations with the same initial conditions. Since variation is also introduced by the difference in network architecture, variability cannot be canceled out in this way.

the improvement never changes. This is caused by the fact that the difference between networks is a lot higher, about 15%. Moreover, the variability in miss latency was, at less than 5% of the absolute latency, lower than the variability in program runtime.

Note also that there is no correlation between the variations of executions with the same initial condition, as is visible in Figure 6, so the variations do not cancel each other out if comparisons would only be made between runs with the same initial conditions.

VI. Solutions

A. Using statistics: characterizing variation

Alameldeen and Wood suggest in [2] to use statistics in coping with variability: by performing multiple simulations, with different initial conditions or small perturbations in the architecture to trigger divergent behavior, variability can be characterized. Using confidence intervals based on this variability, conclusions can be made to within a preset level of certainty. Figure 7 gives an example of this approach. Up to 20 simulations are performed for mesh and torus networks each. Using the measured variability and Student's t-distribution, the 95% confidence interval is computed. This means that, with a certainty of 95%, the actual performance will be inside the range shown, which is centered around the average of a (small) number of measurements. If one wants to conclude, with a probability of 95%, that the torus architecture performs better than the mesh network, the confidence intervals of both should not overlap.

When shown in this way, it becomes clear that not all metrics behave in the same way. In this case, L2 miss latency allows statistically valid conclusions to be drawn using much fewer measurements. The fact that in some cases variability actually *increases* when doing more simulations, as between 5 and 7 simulations of the runtime of `fft`, may seem strange, but it is exactly due to the fact that these graphs are based on a small number of highly variable measurements. In fact, the seventh simulation of `fft` on a torus network (the one with lowest runtime), which is included in the graph only from point seven on the X-axis onwards, had a much longer execution time. This means that when only six simulations would have been done, one could not get an accurate value for the real variability of this type of measurement. And imagine what the error would be if simulation run number seven would have been the *only* one that was made, and the conclusions of a research paper were based solely on this one measurement!

Also bear in mind that the average difference in performance between the two networks chosen for this study is pretty high (around 10%). Often one likes to make statements about the performance of architectures or compiler options that represent much more subtle changes. Here, much less variability can be tolerated, so a much higher number of measurements must be made.

B. Low-level metrics

As has been shown above, certain low-level metrics prove to be more reliable to quantize the effects of certain architectural modifications, both because they suffer less from variability, and because the same modification usually causes a larger change in (the right) low-level metric than it does in program runtime. On the other hand, the improvement in high-level metrics (when measured correctly, i.e., using sufficient simulation runs to both characterize the variability and to obtain confidence intervals that do not overlap) should be taken into account, to decide whether the architectural improvement – which may seem significant at a low level – actually has any effect on the global performance of the benchmark in question. An example here is the `water.sp` benchmark: Figure 1 shows that, although changing the interconnection network does lower the L2 miss latency, the low communication-to-communication ratio of this specific benchmark causes the optimization to have no real effect on `water.sp`'s total runtime.

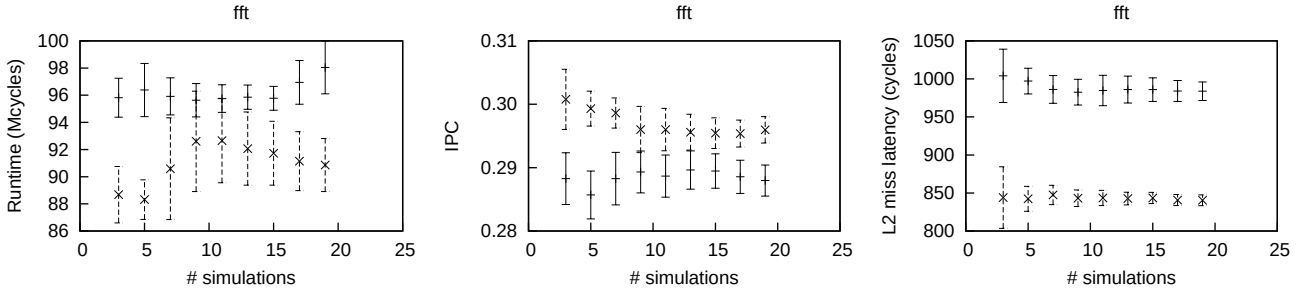

Figure 7. 95% confidence intervals for mesh and torus networks after a number of simulations, for program runtime (left), IPC (center) and L2 miss latency (right). Only when enough simulations are done such that the intervals no longer overlap, a valid statement can be made on which of the network architectures is better.

C. Simulator detail versus input set size

As may be correctly noted, the fact that our simulator uses an in-order processor model will influence the performance results obtained. However, since we wanted to study the effect of changes in the interconnection network, one may argue that, as long as those microarchitectural properties visible to the interconnection network (i.e., the L2 miss rate, the number of concurrently outstanding L2 misses, etc.) are simulated correctly, the load on the interconnection network will be more or less correct, allowing us to draw (within a certain level of accuracy) the correct conclusions. On the other hand, using a simpler simulator can greatly reduce simulation times. This allows one to use longer input sets, or perform multiple simulations of a short input set, both of which reduce variability and its associated inaccuracy.

We thus see the emergence of another trade-off: that between the error introduced by a less-than-cycle-accurate simulation model, and the error resulting from runtime variability. When the available simulation time is kept constant, several pairs of (simulator detail, input set size) can be constructed. One of those pairs will have the lowest combined error. We believe that for studies in which the interaction between processors is large, such as those where shared caches or interconnection networks are investigated, the optimum may be surprisingly far towards the low detail/large input set side. Further research into this trade-off is necessary, however.

D. Matched-pair sample comparison

In [6], Wenisch et. al. propose SimFlex, a methodology for statistical sampling of execution-driven architectural simulation. To avoid the warming up of microarchitectural state before each sample, they first run a quick functional simulation of each benchmark, and construct at regular intervals *flex points*, these are snapshots of selected microarchitectural state which can be loaded before the execution of each sample. With some care, this state can be made general enough to apply to several proposed architectures – e.g., state for a large cache can easily be converted to the initial state of a smaller cache. Other, short-lived state such as that of the reorder buffer can be reconstructed by a short warming up period of only a few thousand instructions.

This methodology allows comparison of different architectures through matched-pair samples: since the simulation of a sample starts by loading the state from a flex point, the initial state is the same for each architecture under test. This results in a situation similar to that in Figure 6, but in this case, because the samples are very short, there won't be enough time for the simulations to diverge. This should eliminate most of the variability.

This method actually seems most promising, since it prevents variability – which needs to build up over time – to come into play. At the same time it greatly reduces simulation time. However, the technique only holds insofar as the same flex points are valid for each microarchitecture under test. Some architectural enhancements, such as widely diverging cache architectures, require different state to be stored. Also, all possible influences of the microarchitecture on the execution path of the benchmark – including systematic ones – are removed, which may not always be desirable. Finally, like all sampling methods it runs the risk of choosing a non-representative sampling set, which introduces its own range of inaccuracies.

E. Alternative solutions

Other solutions may be to use traces, of executed instructions, of memory references or of network packets. Here, the outcome of synchronization races is only determined once (while recording the trace), which removes this type of variability. The advantage is similar to the use of flex points, i.e., variation in the execution path of the application is avoided since the same path is recorded once and subsequently used for all

simulations. However, traces may not be useful for all types of measurements, since they too ignore all kinds of feedback between architecture and benchmark, which may be important. Also, a single trace only exposes the architecture under test to a single combination of synchronization outcomes. A different trace, which in a real system can occur with equal probability, might incur other behavior of the architecture. Again, a multitude of traces – representing the different scheduler and synchronization decisions possible at runtime – may need to be used, and the combined results interpreted.

Synthetic loads can be another option. For instance, [12] introduces a model of synthetic network traffic that is representable for the traffic patterns encountered on real multiprocessor or CMP interconnection networks. Since there are no synchronization issues here, its variability is much lower – even for simulation runs that are much shorter than even a benchmark with a reduced input set. Due to simplifications in synthetic loads, the absolute accuracy, and even the relative accuracy, of these solutions are not always very high, although their fidelity is usually a lot better than that of execution-driven simulation.

VII. Conclusions

Parallel applications, which are to become much more prevalent following the broader adoption of multi-core processors, have intrinsic non-determinism in their execution. Their behavior can therefore change when different initial conditions are presented, or when the architecture on which the program is run varies even slightly. Research into architectural improvements, based on measurements of the runtime of parallel programs on real systems or in simulations, should take these variations into account. This prevents one from making wrong conclusions when architectures with a performance difference similar in magnitude to the inherent variation in the performance metrics used, are compared. While a definitive solution to this problem has yet to be found, possible solutions may include (a combination of) the use of larger benchmark input sets or performing multiple close-to identical simulations and using statistics to characterize and average out variability. When total simulation time is limited, gain can be found in the use of low-level metrics (which can have less variability, or a larger separation between architectures under test), matched-pair sample comparison, or trading off simulation detail for input set size.

VIII. Acknowledgments

This work was supported in part by the Inter-university Attraction Poles program photonics@be (IAP–Phase VI), initiated by the Belgian State, and by HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation. We also thank the anonymous reviewers for their many helpful comments.

References

[1] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd International Conference on Object-Oriented Programming, Systems, Languages and Applications*. Montreal, Canada, Oct. 2007, pp. 57–76.

[2] A. Alameldeen and D. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA-9)*, Anaheim, Calif., Feb. 2003, pp. 7–18.

[3] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, Jul. 2006.

[4] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[5] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[6] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[7] K. M. Lepak, H. W. Cain, and M. H. Lipasti, "Redeeming IPC as a performance metric for multithreaded programs," in *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. New Orleans, Louisiana: IEEE Computer Society, Sep. 2003, p. 232.

[8] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, Jun. 1995, pp. 24–36.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," Princeton University, Department of Computer Science, Tech. Rep. TR-811-08, Jan. 2008.

[10] W. Heirman, J. Dambre, I. Artundo, C. Debaes, H. Thienpont, D. Stroobandt, and J. Van Campenhout, "Predicting the performance of reconfigurable optical interconnects in distributed shared-memory systems," *Photonic Network Communications*, vol. 15, no. 1, pp. 25–40, Feb. 2008.

[11] E. Artiaga, X. Martorell, Y. Becerra, and N. Navarro, "Experiences on implementing PARMACS macros to run the SPLASH-2 suite on multiprocessors," in *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, Jan. 1998, pp. 64–69.

[12] W. Heirman, J. Dambre, and J. Van Campenhout, "Synthetic traffic generation as a tool for dynamic interconnect evaluation," in *Proceedings of the 2007 International Workshop on System Level Interconnect Prediction (SLIP'07)*. Austin, Texas, Mar. 2007, pp. 65–72.