DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time

Sylvain Girbal
LRI, Paris South University and CEA
France

Gilles Mouchard
LRI, Paris South University and CEA
France

Albert Cohen
INRIA Rocquencourt
France

Olivier Temam
LRI, Paris South University
France

ABSTRACT

While architecture simulation is often treated as a methodology issue, it is at the core of most processor architecture research works, and simulation speed is often the bottleneck of the typical trial-and-error research process. To speedup simulation during this research process and get trends faster, researchers usually reduce the trace size. More sophisticated techniques like trace sampling or distributed simulation are scarcely used because they are considered unreliable and complex due to their impact on accuracy and the associated warm-up issues.

In this article, we present DiST, a practical distributed simulation scheme where, unlike in other simulation techniques that trade accuracy for speed, the user is relieved from most accuracy issues thanks to an automatic and dynamic mechanism for adjusting the warm-up interval size. Moreover, the mechanism is designed so as to always privilege accuracy over speedup. The speedup scales with the amount of available computing resources, bringing an average 7.35 speedup on 10 machines with an average IPC error of 1.81% and a maximum IPC error of 5.06%.

Besides proposing a solution to the warm-up issues in distributed simulation, we experimentally show that our technique is significantly more accurate than trace size reduction or trace sampling for identical speedups. We also show that not only the error always remains small for IPC and other metrics, but that a researcher can reliably base research decisions on DiST simulation results. Finally, we explain how the DiST tool is designed to be easily pluggable into existing architecture simulators with very few modifications.

Categories and Subject Descriptors

C.1.1 [**Processor Architecture**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*; C.4 [**Performance of Systems**]: *Measurement techniques*

General Terms

Design,Measurement,Performance

Keywords

Distributed simulation, Processor architecture

1. INTRODUCTION

Within architecture research works and articles, simulation (and methodology in general) is often treated as a minor issue, and devoting a research article to this topic is rather atypical. However, in the last editions of four major conferences in computer architecture (MICRO, ISCA, ASPLOS and HPCA), cycle-precise simulators are used in 72 out of 103 articles, i.e., 70% of articles: simulation *is* a critical tool at the core of our research works. All processor architects have experienced in their research studies that what they are able to simulate in a restricted period of time often determines the extent, precision and quality of their research work, and even in some cases, the number of solutions and ideas that they can evaluate.

Therefore, simulation and the ability to speed up simulation are not just methodology issues, they directly affect the research part of architecture studies, and thus deserve more attention.

In many cases, exploring a new idea is a trial-and-error process: one implements a variation of an idea, tests it, updates the simulator, tests it again, and so on. In practice, whenever a variation of an idea looks interesting, processor architects want to confirm their observations by simulating and validating it on a large number of benchmarks, e.g., part or all of the Spec benchmarks, for a range of architectural parameters. In that case, having lots of computing resources is useful because simulations can be run in parallel over a large number of machines. However, between these convergence steps, the research process often consists in analyzing what is happening in a few or even a single benchmark, leading to a fast-paced sequence of modify-run simulation cycles. Because this part of the research process can be the most time-consuming, researchers are often willing to temporarily trade some accuracy for speed, providing research decisions are not significantly altered by the loss of accuracy. As a result, researchers often cut down on trace sizes to get trends faster, and later on, run longer traces on more benchmarks and more parameters when the research process seems to converge again. However, because of increasingly large data structures such as large L2 caches, trace size reduction can become too unreliable and thus an unsatisfactory alternative, as we later show in this study.

Because of the increasing processor complexity, processor simulation is now also used for program tuning [4, 18, 23] where simulation time is even more critical. Simulators provide a very detailed understanding of program behavior on complex processor architectures, but in that case, the optimization process is usually entirely sequential: a single program is tuned on a single architecture.

Finally, as processor complexity and application size increase, simulation time increases as well. For instance, simulating a Spec95 benchmark running on a 5-stage pipeline MIPS R2000 processor requires 7 hours in average on a Pentium 4 1.6GHz, while simulating a Spec2000 benchmark running on a 4-way out-of-order superscalar processor (using the SimAlpha simulator [7]) requires 356 hours in average on the same machine.

The most frustrating aspect of processor simulators is that their complexity and sequential nature does not lend well to parallelization. As a result, whenever running one or a few simulations, having lots of computing resources does not enable to cut down simulation time.

Several studies have demonstrated that sampling methods can bring significant speedups without loosing much accuracy [14, 17, 16]. However, they do not propose a simple and comprehensive method to control the associated accuracy issues; indeed, the accuracy decreases relatively quickly with the speedup since the principle is again to reduce the trace size in order to decrease simulation time. Conte et al. [5] has analyzed in details the effect of resetting processor state in sampling techniques, but the method has not gained a wide acceptance in the community because of what is often seen as an excessively complex overhead for a methodology issue. Several studies [20, 21, 11, 2, 10, 8] have proposed identifying representative program regions for simulation, but in practice, an exhaustive search of architecture research articles in the major conferences shows that few research groups effectively use such techniques yet, again because of their complexity, their restricted scope or the lack of widely distributed tools. Still, the *Basic Block Vectors* [21] approach is promising because it is completely hardware-independent; however, while it is quite accurate for global metrics such as IPC, it is less accurate for large memory components like L2 caches because it is based on relatively small trace sizes, and more important the mechanism is not designed to adjust the trace size to the memory components size; finally, the technique is data-set sensitive, and the analysis must first be conducted for each program/data set pair.

Nguyen et al. [15] proposed another approach for speeding up simulation: to split the whole trace into N separate chunks that are distributed over N machines. As a result of splitting, simulation accuracy decreases because the processor state is implicitly reset at the beginning of each chunk as with sampling techniques. Consequently, each chunk needs to perform a simulation warm-up to refresh the processor state (especially the memory structures), but the minimum warm-up interval size varies strongly with each chunk, benchmark and simulator combination. To compute the warm-up interval size, Nguyen et al. [15] propose a heuristic which requires to know in advance the L1 miss ratio of each program. Haskins et al. [9] propose a more sophisticated probabilistic technique to determine the warm-up interval size, but they again need to run cache simulations first for each benchmark, and their computations are excessively time-consuming for set-associative caches. Thus, in most cases, these heuristics are impractical.

In this article, we present the Distributed Simulation Tool (DiST), a practical distributed simulation technique that automatically adjusts the warm-up interval size to satisfy user-defined error thresholds, relieving the user from most of the accuracy issues. As a consequence, DiST can strongly reduce simulation time at the cost of slightly reducing simulation accuracy while still remaining very easy to use. The technique focuses on limiting the associated accuracy issues, and it is designed to always privilege accuracy over speedup. While a researcher is always willing to speedup simulations, one is not ready to accept an *unpredictable* loss of simulation accuracy that can impair research decisions. We experimentally show that, for none of the SpecInt2000 and SpecFp2000, the IPC error is greater than 5.29% when we applied our technique to SimAlpha [7], a validated Compaq Alpha EV6 processor version of the SimpleScalar [3] simulator which is widely used in the processor architecture community. DiST achieves an average speedup of 7.35 with 10 machines. Moreover, we experimentally show that typical research architecture decisions — the relative performance comparison of two simulated hardware configurations — were never affected by the slight loss of accuracy, provided the observed metric variation is not smaller than the error. For all statistics, we have *always* observed that either the relative error is small or the number of events is negligible. Finally, the DiST tool has been specifically designed for easy plugging into existing simulators with minimal modifications: in order to plug the tool into SimpleScalar and SimAlpha, we only had to modify 10 source lines in each simulator.

Section 2 introduces the principles of the distributed simulation technique, Section 3 describes the DiST implementation, Section 4 outlines the experimental methodology, Section 5 presents speedups and analyzes accuracy issues, and Section 6 outlines that fast simulation enables new applications like program optimizations.

2. PRINCIPLES

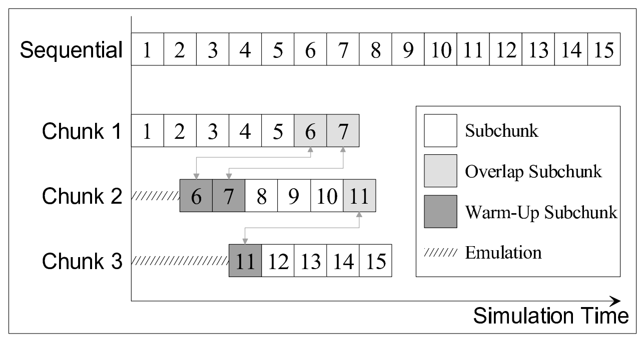

Figure 1: Chunks and subchunks schedule.

With DiST, simulation speedups derive from splitting the simulation process into a set of N chunks and distributing them over the same number of machines. Implementing simulation splitting and distribution is rather straightforward, except for the accuracy issue.

Cycle-level simulation models are not 100% accurate: they almost all include some approximations, e.g., imprecise description of the memory system, missing or too many internal data paths, impact of the operating system not considered, etc. Therefore a small additional source of inaccuracy may be acceptable. To be tolerated in practice, it must fulfill two constraints: the user must have confidence that the additional error is small, and moreover, it must have as little impact as possible on her/his research decisions. The goal of our technique is to meet these accuracy constraints with minimal user overhead. And we experimentally show in the next section that both of these conditions are satisfied with DiST. The main challenge is to automatically control the level of accuracy and to privilege accuracy over speedup by trading speedup for accuracy whenever necessary.

The principles of distributed simulation are the following. Assume one wants to run a simulation of N instructions with 3 machines: then, each machine will simulate $\frac{N}{3}$ instructions. The first $\frac{N}{3}$ instructions are simulated on the first machine, the second $\frac{N}{3}$

instructions on the second machine and so on, see Figure 1. Each machine runs the program from the beginning but starts simulating only at the first instruction of the specified chunk, thus, each machine must *emulate* (functional simulation only) the program until it has reached this point. So, the first machine performs no emulation, the second machine emulates the first $\frac{N}{3}$ instructions and the second machine emulates the first $\frac{2\times N}{3}$ instructions. Considering the speed gap between emulation and simulation, a typical distributed simulation schedule will look like Figure 1. However, at the end of emulation, before simulation starts, all processor structures are empty (caches, branch prediction tables...) so the first simulated instructions are likely to exhibit unusual behaviors (excessive cache misses, branch wrong predictions...) that would flaw statistics. For that purpose, the processor state is usually warmed-up/refreshed by simulating a certain number of instructions without recording the corresponding metrics; in other words, simulation starts before the chunk first instruction is reached. Because this warm-up interval can have a strong impact on both accuracy and simulation time, the biggest difficulty is to find an appropriate warm-up interval size.

Dynamic warm-up mechanism. In DiST, instead of running a fixed number of warm-up instructions *before* chunk m, we let chunk $m-1$ perform additional instructions *after* it has completed all its instructions. Therefore the last instructions of chunk $m-1$ are the same as the first instructions of chunk m. During that *overlap* period, we constantly compare the simulation results of chunk $m-1$ and chunk m until they converge. When they become almost identical, we can reasonably assume that the processor state in chunk m has been properly warmed-up and chunk $m-1$ stops. More precisely, to perform these comparisons, we split each chunk into a set of subchunks of fixed size (typically a few million instructions, see Section 4). For instance, in Figure 1, chunks 2 and 3 respectively need 2 and 1 warm-up subchunks; symmetrically, chunks 1 and 2 perform 2 and 1 overlap subchunks. During the overlap period, the chunks m and $m-1$ dump the simulation statistics of all the subchunks, and an offline process gathers the data and compares the statistics of the subchunks corresponding to the same instructions. The user can specify, through a simple scripting language described in Section 3, which statistics are used for the comparison and what is the convergence threshold. Typically, in many of the simulations of this study, the convergence is based on the IPC metric with a 98% convergence threshold, i.e., the IPC of the subchunks from chunks m and $m-1$ cannot differ by more than 2%. Whenever a subchunk meets this criterion, the warm-up stops. Then, all the statistics of "inaccurate" warm-up subchunks of chunk m, i.e., the subchunks simulated before the convergence occurred, are discarded and replaced in chunk m by the statistics of overhead (overlap) subchunks of chunk $m-1$. Note that, assuming all chunks run at a similar speed, the overhead subchunks have no impact on the distributed simulation time, because the last chunk has no such overhead subchunks, see Figure 1. Finally, another asset of the technique is that simulation accuracy is implicitly priviledged over simulation speedup: in the worst case where a chunk $m-1$ never converges with chunk m or any later chunk, chunk $m-1$ will run sequentially for the remainder of the simulation.

Overall, the main asset of dynamic warm-up over previous static warm-up techniques is not speed but accuracy: using dynamic warm-up, it is possible to distribute simulation and ensure a reasonable level of accuracy with minimal user input.

3. IMPLEMENTATION

Using distributed simulation raises several implementation issues: fine-tuning the dynamic warm-up mechanism, applying distribution to an existing simulator, combining the distributed statistics of each chunk without modifying the statistics procedures of an existing simulator, specifying the local accuracy constraints. At the end of the section, we also we provide a brief description of the software environment.

Using existing simulators. Simulator development is a painstaking and time-consuming task because of a long modeling, debugging and validation process. Consequently, DiST is designed so as to require minimal simulator modifications. To plug DiST into a simulator, we only need to force the simulator to dump statistics at periodic intervals instead of at the end of the simulation. To plug DiST into SimAlpha [7], we only had to modify 10 source lines, see http://www.microlib.org/DiST for more details. Similarly, DiST was plugged into SimpleScalar [3] and a PowerPC G3 Simulator based on SystemC [22].

A simulator is DiST-compatible if it provides a mechanism for fast-forwarding in the simulation: either an emulator or a checkpointing mechanism. Though we used emulation for fast-forwarding in this study, checkpointing is even more reliable and efficient because it guarantees that all distributed threads will not be sensitive to operating system effects, and besides, it saves the emulation phase.

Combining distributed statistics. During simulation, DiST collects each subchunk statistics from the simulator; upon termination all collected local statistics must be combined to obtain the statistics of the full run. Depending on statistics, the task can be trivial or require some overhead. For instance, cumulative statistics like the number of misses need only be summed up to get the full run statistics. But ratios must be recomputed, i.e., if chunk 1 dumps the ratio $\frac{A1}{B1}$ and chunk 2 dumps $\frac{A2}{B2}$, the ratio for the full run is $\frac{A1+A2}{B1+B2}$, not $\frac{A1}{B1}+\frac{A2}{B2}$. For that purpose, the tool implements a simple scripting language to define statistics. We only assume that, upon completion, a simulator dumps a text file which contains all statistics, and that statistics are listed one per line. See Figure 2 for the data L1 cache statistics in the SimAlpha output (unmodified).

DL1.hits	1471654945	# total number of (all) hits
DL1.misses	25916780	# total number of misses
DL1.accesses	1497571725.0000	# total number of accesses
DL1.miss_rate	0.0173	# miss rate (i.e., misses/ref)

Figure 2: An example of simulator output (SimAlpha).

In the tool, `DL1.hits`, for instance, is directly used as a variable name to characterize the statistic, so the only additional constraint on the statistics file is to have unique names for each statistic. The statistics file lists variables which are either followed by a comment (in which case they are cumulated), or by "=" and any mathematical expression followed by a comment.

```
DL1.hits : "total number of (all) hits"
DL1.misses : "total number of misses"
DL1.accesses = DL1.hits + DL1.misses : "total number of accesses"
DL1.miss_rate = DL1.misses / DL1.accesses : "miss rate (i.e., misses/ref)
```

Figure 3: An example script for combining statistics.

For instance, the fourth line in Figure 3 specifies that the `DL1.miss_rate` full run statistic is equal to the ratio of the full run `DL1.misses` statistic over the full run `DL1.accesses` statistic, which is itself defined in the above line as the sum of two full run statistics.

Therefore to get full run statistics, the user needs only get the output file and write the appropriate expressions for some of the statistics. At run-time, the tool will parse the simulator local outputs and match them with these expressions. Consequently, it is not necessary to modify the statistics procedures and the output file of a simulator to plug DiST, except if several statistics have the same name.

Implementing local constraints. We use the same scripting language and statistic parsing mechanism to implement local accuracy constraints. For instance, if we want to impose a 1% local accuracy constraint on Data L1 cache miss rate, we will insert the line of Figure 4 into the local constraints file.

```
abs(~DL1.miss_rate – DL1.miss_rate) / DL1.miss_rate < 0.01
```

Figure 4: An example of local accuracy constraint.

`~DL1.miss_rate` denotes the statistic of the new chunk, e.g., $chunk_m$, which is compared with the statistic of the terminating chunk, i.e., $chunk_{m-1}$. The comparison is repeated on each subchunk until the condition is fulfilled, as explained in Section 2. The different constraints are listed in the local constraints file, see Figure 4, and the tool performs a logical AND on all these constraints.

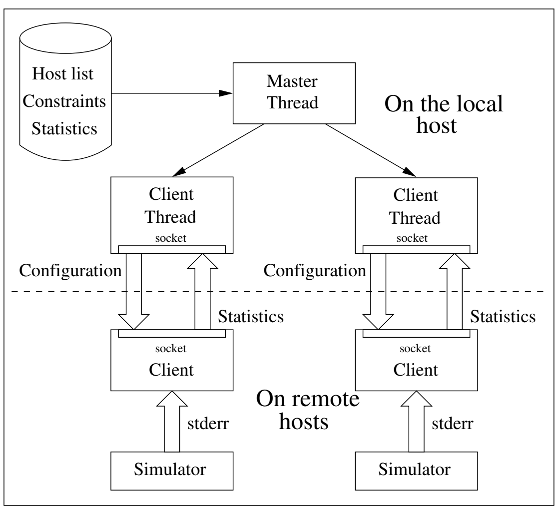

Figure 5: The DiST architecture.

DiST. DiST itself uses a client/server architecture, see Figure 5. We assume the simulator object code and the benchmarks data reside on all the machines (hosts). When a simulator is started through DiST, it spawns one client to each target host (one chunk per client) using a remote shell service. Each client connects to the server which determines the chunk size, assigns chunks depending on the number of available processors and the processor speed. The server is multithreaded: one master thread plus one thread per client; the client gets its client configuration upon start-up and sends back statistics to the server after each subchunk; the server determines when a client should stop by comparing the overlap and warm-up subchunks of two consecutive clients; finally, the master thread also combines statistics when all chunks have completed.

Note that distributing several chunks of the same benchmark trace over different machines raises file sharing issues (they write in the same benchmark output files) which are resolved through copying by the server.

Also, note that since each client only sends one message per subchunk, the network traffic is very low and does not induce any contention, even with a large number of machines (several tens). For instance, with SimpleScalar, 16-million instructions subchunks for a 4-billion trace only induce a few hundred 100-byte messages. Consequently, DiST is compatible with slow networks.

DiST versus workload management systems. Many environments, like Condor [13], propose to exploit multiple computing resources by distributing jobs across several machines. For instance, Condor schedules, pauses, and migrates queued jobs to optimize CPU utilization. However, Condor is not designed to manage interprocess communications, especially for the cooperative jobs of a client/server application like DiST. Still, we might augment DiST with Condor, and use the Condor library to distribute our specific jobs more efficiently across the available computing resources.

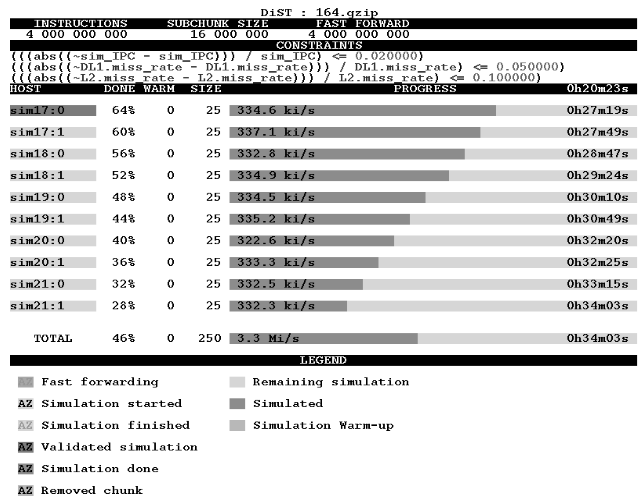

Figure 6: DiST graphical user interface.

GUI. Finally, DiST comes with a simple text-based graphical user interface, see Figure 6, which is helpful for monitoring simulation progression, the amount of warm-up for each chunk and to estimate the total simulation time.

4. EXPERIMENTAL FRAMEWORK

Simulation environment. In the next section, we use SimAlpha and the Spec2000 benchmarks to evaluate distributed simulation. We chose SimAlpha [7] over SimpleScalar [3] because the simulated architecture is closer to that of a true processor (Figure 7 describes SimAlpha baseline configuration), and because the ratio of emulator speed over simulator speed is higher than SimpleScalar so that the speedup threshold is higher as well, see Section 5. We used 4-billion instructions traces for each benchmark, skipping the first 4 billions. We performed the main experiments (speedup and accuracy) on 22 Spec2000 benchmarks (4 could not be run on our machines). We ran our experiments on several different machine sets: we had access to 10 Pentium III 733 Mhz machines with 128MB, 10 Pentium III 500 MHz machines with 256MB, and we had restricted access to 40 Athlon XP1800 with 1GB. Because of these access restrictions, we could run several, but not all, our ex-

Parameter	Value
Processor core	
Fetch width	up to 4 instructions per cycle
Issue width	up to 4 integer ops per cycle
	plus 2 floating ops per cycle
Functional units	4 ALUs + 2 FPUs
Commit width	up to 11 instructions per cycle
Branch Prediction	
Predictor	21264 predictor (hybrid)
BTB	512 entry/4-way associative
Return Addr Stack	32 entries
Memory Hierarchy	
L1 Data Cache	32 KB/2-way associative/LRU
L1 Instruction Cache	32 KB/2-way associative/LFU
L2 Cache	2 MB/direct mapped/LRU
ITLB	128 entries/fully associative
DTLB	128 entries/fully associative

Figure 7: ***Baseline configuration*.**

SpecINT2000	SpecFP2000
175.vpr	172.mgrid
186.crafty	178.galgel
197.parser	179.art
252.eon	187.facerc
255.vortex	188.ammp
300.twolf	200.sixtrack

Figure 8: ***Reference set*.**

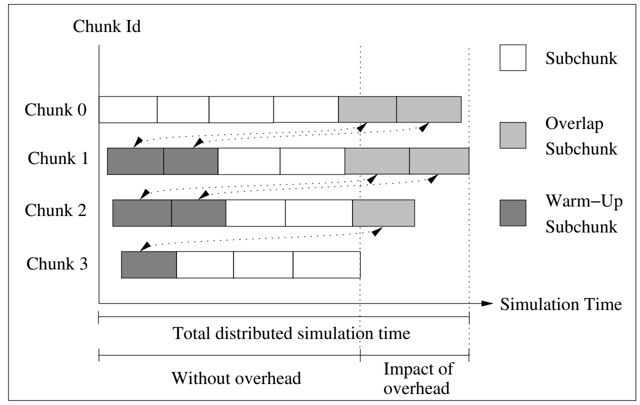

Figure 9: Impact of subchunk size on total distributed simulation time.

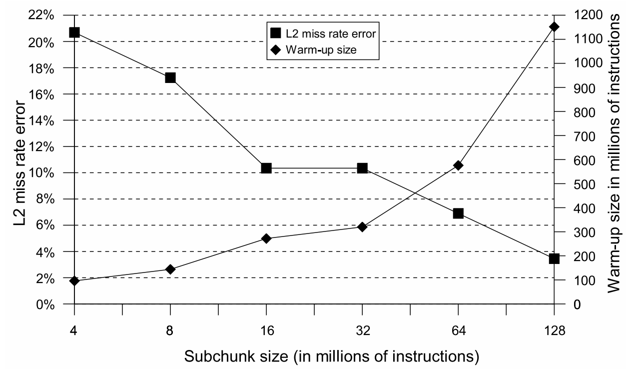

Figure 10: Impact of subchunk size on speedup and accuracy; ***10 PIII 500*.**

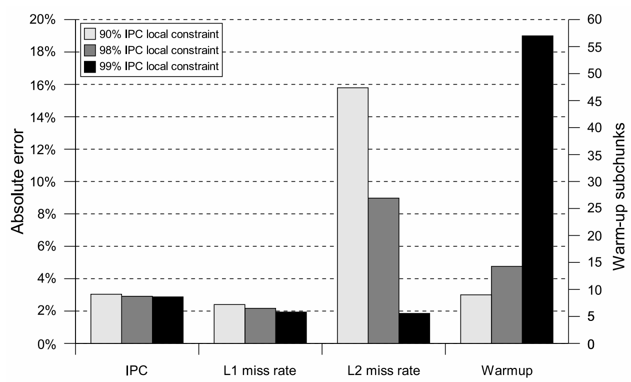

Figure 11: Choosing a convergence threshold based on IPC.

periments on the 40 Athlon. For each figure, we indicate on which set of machines the experiments were run. Note that 10 of the 22 Spec2000 benchmarks induce excessive swapping during simulation on the Pentium III 733 Mhz with 128MB due to limited memory resources, so that the execution time, and thus the speedup, could not be accurately estimated. Consequently, we have defined a *full set* corresponding to all 22 benchmarks, and a *reference set* of 12 benchmarks that could run on all machine sets, see Figure 8 (fortunately, the reference set is a balanced mix of SpecInt and SpecFp codes). Because of these machine constraints, and subsequently, for the sake of the comparisons, some experiments are only provided for the reference set.

DiST parameters. The main parameterization issues of DiST are finding the appropriate subchunk size and defining the convergence threshold. Intuitively, the larger the subchunk, the more relevant and accurate the comparison between warm-up and overlap subchunks. Therefore, the larger the subchunk, the smaller the error. But with a too large subchunk, the warm-up period and thus the simulation time of a machine m can increase up to the point that it affects the overall distributed simulation time, see Figure 9. Therefore, we need to find a subchunk size value which realizes a reasonable tradeoff between accuracy and speedup. Figure 10 suggests that 16-million instructions subchunks achieve both low error and low warm-up overhead.

We experimentally found that in many cases only constraining the IPC metric error was sufficient to achieve reasonable accuracy for most processor components and metrics. Imposing an error constraint on another metric is only necessary when the purpose of simulation is to study a specific component. Experiments also showed that a 98% local accuracy constraint on IPC proved a reasonable value for all benchmarks: for instance, at 90% or less, accuracy significantly decreases and the dynamic warm-up mechanism is not exploited, while beyond 99%, the warm-up overhead increases significantly so that speedup decreases, see Figure 11.

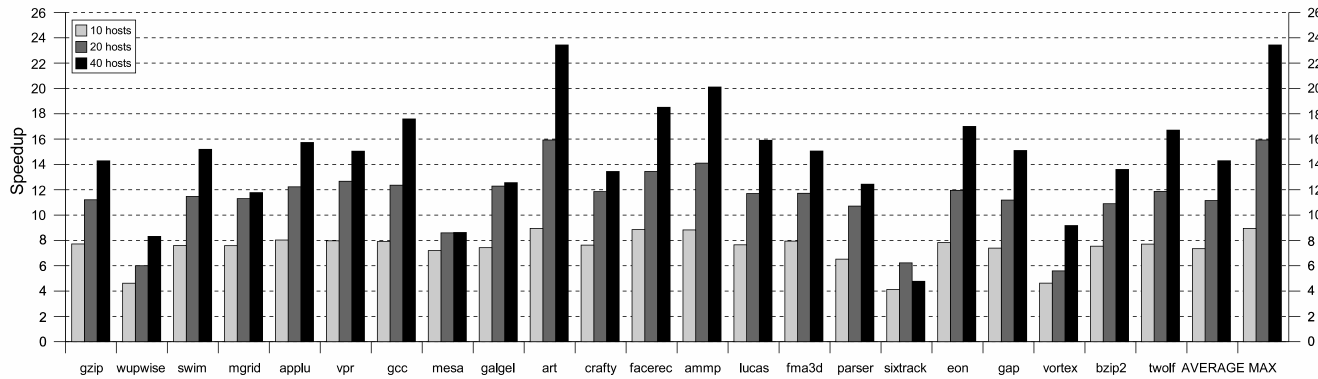

Figure 12: Speedup with 10, 20 and 40 machines; *full set, Athlon XP 1800+.*

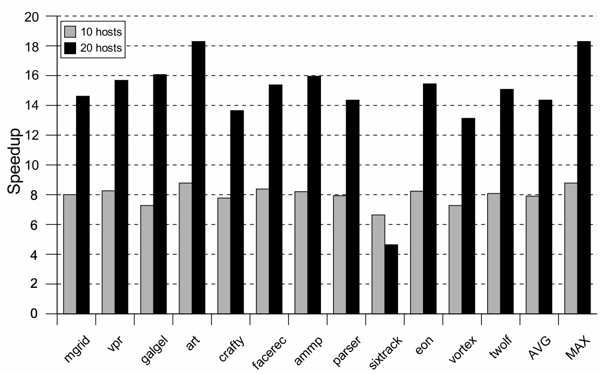

Figure 13: Speedup with 10 and 20 machines; *reference set, 10 PIII 500 and 10 PIII 500 + 10 PIII 733.*

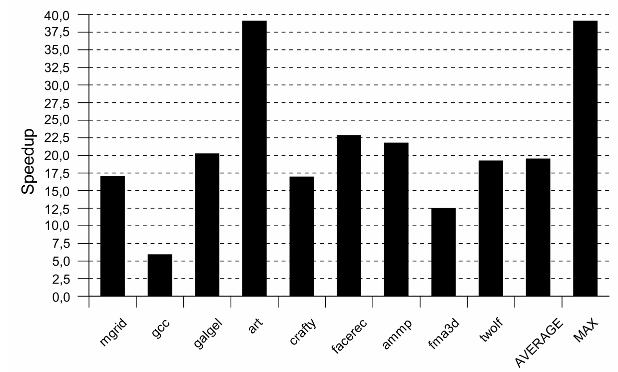

Figure 14: Speedup with 40 machines; *restricted set, 16 billion instructions, fastforward of 4 billions, 40 Athlon XP 1800+.*

Definitions. In the paragraph below, we define several metrics.
Global error on a metric is defined by the following formula:
$$\frac{metric_{dist} - metric_{seq}}{metric_{seq}}$$
where $metric_{seq}$ is the metric provided by the sequential simulation on a single processor, and $metric_{dist}$ is the combined metric computed by DiST after all chunks have executed.
Sequential time is the sequential simulation time on a single processor.
Distributed time is the time interval between the beginning of the first chunk and the end of the last running chunk, see Figure 9.
Speedup is the ratio of Sequential time over Distributed time.

5. SPEEDUP AND ACCURACY

Speedup. Distributed simulation can achieve very significant speedups. Figure 12 shows that DiST achieves an average speedup of 7.35 using 10 processors, 11.15 using 20 processors and 14.29 with 40 processors, for a 98% local accuracy constraint on IPC for a 4-billion simulated trace.

With a larger (16-billion) trace, DiST better benefits from a large number of machines, and with 40 processors the average speedup increases to 19.5 with a maximum speedup of 39.07 for *179.art* spec-code, see Figure 14. As an example, the average sequential simulation time for a 16-billion trace on an Athlon is 20 hours and it decreases to 1 hour using 40 machines.

The average speedup is not the same for all benchmarks because of the variable size of the warm-up phase. As the number of machines increases, the warm-up overhead increases while the trace size remains constant, thus the speedup does not increase linearly because there are proportionnally more warm-up subchunks in each chunk (from 6% with 10, 12% with 20 machines to 22% with 40 machines for 4-billion traces).

With 10 Pentium III 500 MHz, the speedup is 7.82 with an average sequential simulation time of 15 hours, see Figure 13. It increases to 14.35 using a heterogeneous set of 10 Pentium III 500 MHz (256MB) plus 10 Pentium III 733 MHz (128MB), where the reference performance is given by a Pentium III 500 MHz (256MB).

Threshold speedup. Besides the trace size limitation, the ratio of the speed of the emulator over the speed of the simulator is a speedup upper-bound. Let us consider a simple case where emulation and simulation speed remain constant during the whole simulation. v_s denotes the simulator speed (number of instructions per second), v_e denotes the emulator speed, I is the number of instructions per chunks, and N is the number of chunks (i.e., the number of processors). $t_s = I/v_s$ is the time needed to simulate one chunk, and $t_e = I/v_e$ is the time needed to emulate one chunk. Then, $T_{seq} = N \cdot t_s$ is the sequential time, and $T_{dist} = (N-1) \cdot t_e + t_s$ is the distributed time (time to emulate $N-1$ chunks and simulate the last chunk). The speedup is then equal to $\frac{T_{seq}}{T_{dist}} = \frac{N \cdot t_s}{(N-1) \cdot t_e + t_s}$. Note that when $N \to \infty$, the speedup converges towards $t_s/t_e = v_e/v_s$, i.e., the ratio of the emulator speed over the simulator speed.

We noted this speed ratio varies with each simulator-emulator pair on both the Athlon and the Pentium III, and for each benchmark. On the Athlon, the average ratios are 31 for SimAlpha, 23 for

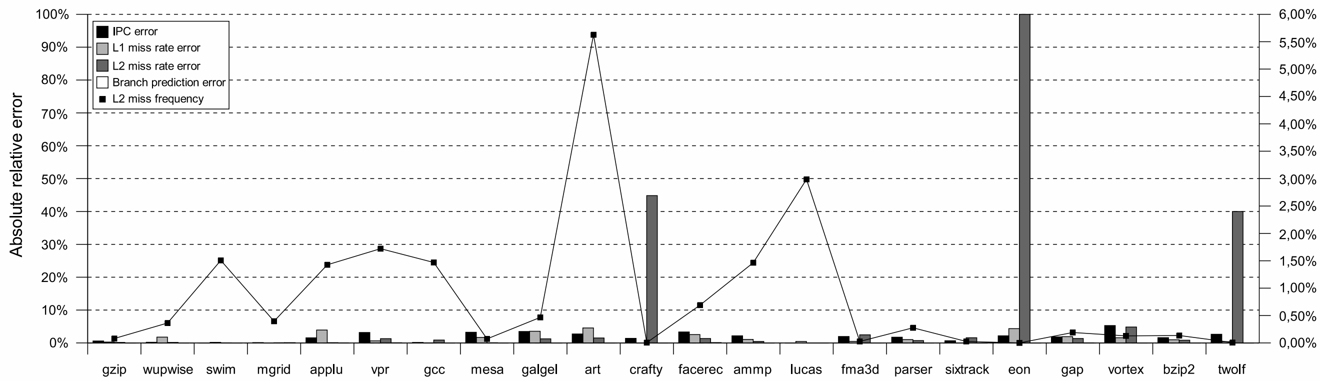

Figure 15: DiST accuracy; *full set, 40 Athlon*.

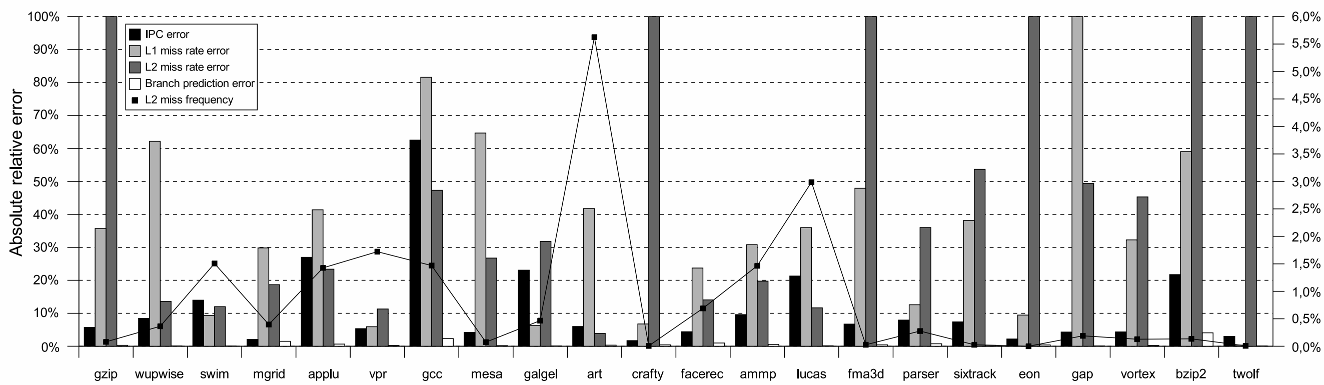

Figure 16: Small (100 millions) trace accuracy; *full set, 1 Athlon*.

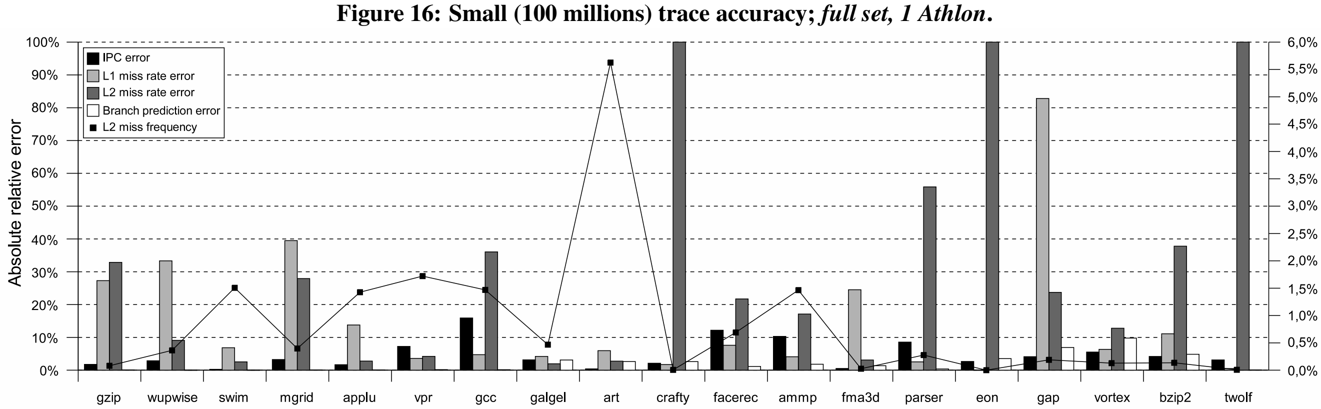

Figure 17: Trace sampling accuracy; *full set, 1 Athlon*.

SimpleScalar and 185 for a PowerPC G3 simulator based on *SystemC*. And on the Pentium III, they are respectively 48, 36 and 239. The speed ratio also varies widely accross benchmarks, e.g., from 32 to 94 for SimAlpha on the Pentium III. Still, we did not have enough machines and use long enough traces to reach the speedup threshold of any of these simulators. Assuming a large number of machines, the best way to take advantage of distributed simulation and DiST is to develop fast emulators [12, 19]. It is also possible to get rid of the emulation phase altogether and use checkpointing, such as the EIO checkpointing implemented in SimpleScalar [3]. In that case, DiST performance is only bounded by the number of available computing resources.

Accuracy. Even though distributed simulation can provide very significant speedups, researchers will effectively use it only if they have reasonable confidence in the results accuracy. For that purpose, we have applied distributed simulation on all the Spec2000 benchmarks (see Section 4), and we found that not only the average distributed simulation IPC error is fairly small at 2.60%, but that it also *always* remains smaller than 5.17%, using a local accuracy constraint on IPC at 98%, see Figure 15. Besides IPC, the error on other important metrics like branch prediction, L1 miss rate and L2 miss rate also remains smaller than 10% in most cases.

Small error or negligible number of events. We have always observed that either the relative error is small or the number of events is negligible. Consider Figure 15 where the bargraphs represent the absolute value of the relative error (which can be positive or negative) and the line represents the L2 miss frequency; the left vertical axis corresponds to bargraphs and the right vertical axis corresponds to the line. The L2 miss frequency is defined as the ratio of the number of L2 misses over the trace size. In the three cases where the error is not negligible, i.e., the L2 data cache miss rate of *eon* (266%), *crafty* and *twolf*, the absolute number of events measured by each metric is in fact negligible — 1985 misses in *eon* — so that a tiny variation of the number of events is enough to induce a large variation of the relative error. Such a tiny number of events has a negligible impact on performance, and therefore the variation is unlikely to bias research decisions.

***Dynamic* warm-up is necessary, especially because of large memory structures.** Even though DiST achieves almost the same level of accuracy for all benchmarks, i.e., 2.60% error on IPC in average, the dynamic warm-up mechanism often proves useful because the warm-up size varies significantly with each benchmark, see Figure 18 where we have measured the average warm-up size (number of warm-up subchunks per chunk). The dynamic warm-up mechanism proves even more useful when we vary the L2 size. Consider Figure 19: clearly, large memory structures have the strongest impact on the warm-up size, i.e., increasing the L2 size can increase the warm-up size, but the effect varies strongly from one program to another. While we effectively observed that the amount of warm-up increases with L2 size on 8 of the 12 benchmarks, for 2 benchmarks the warm-up size is unchanged, and for 2 other benchmarks, *vortex* and *sixtrack* the warm-up size varies unexpectedly. Therefore, the dynamic warm-up mechanism is necessary for both achieving the required accuracy and preserving the speedup by avoiding excessive warm-up overhead. Naturally, when the warm-up interval increases with the L2 size, the speedup slightly decreases, i.e., DiST implicitly privileges accuracy over speedup.

Confidence in research decisions. To further increase confidence in DiST, we have made several experiments to show that research decisions are unlikely to be influenced by the small loss of accuracy. For that purpose, we have selected three processor components often targeted by researchers (branch prediction, L2 cache, register bank), and for each component, we vary a single parameter. For each component and each parameter value, we focus on the *direction* of the performance variation, i.e., positive or negative, with respect to the default configuration parameter, and the *amplitude* of this performance variation. A research decision is not affected by distributed simulation if the variation *direction* is the same as for sequential simulation, and to a lesser extent, if the variation *amplitude* is similar as well. We have observed that both the variation direction and amplitude are almost always the same for sequential and distributed simulation. Implicitly, these experiments suggest that a research decision based on distributed simulation, e.g., choosing the optimal parameter value for a processor component, is usually the same as the decision based on sequential simulation.

For our experiments, we vary the size of the L2 cache, the EV6 branch predictor tables size, and the number of physical registers in SimAlpha [7]. Figures 20, 22 and 24 show the performance variation for each parameter value with respect to the default parameter value, using sequential and distributed simulation. Figures 21 and 23 show the variation of the corresponding processor component metric when applicable: respectively L2 miss rate and branch prediction rate.

The variation direction is almost always the same, and the variation amplitude is usually similar, except when the absolute number of events is negligible as for the L2 miss rate in *eon*. Only when the performance difference is of the order of the error (a few percent), the comparison becomes less precise. Consider the branch prediction experiment in Figure 22 and benchmarks *vpr*, *gcc*, *crafty* and *parser*: the IPC variation amplitudes are very small and the variation direction differs. However, the variation direction and amplitude of the component *metric* — branch prediction rate — are almost the same as for sequential simulation for *all* benchmarks, see Figure 23, so that a research decision based on this metric would be correct. Generally speaking, the scale of the performance variation serves as a safeguard for exploiting distributed simulation results: when it becomes of the order of one percent, i.e., below the typical distributed simulation error, the researcher knows the results may not be trusted.

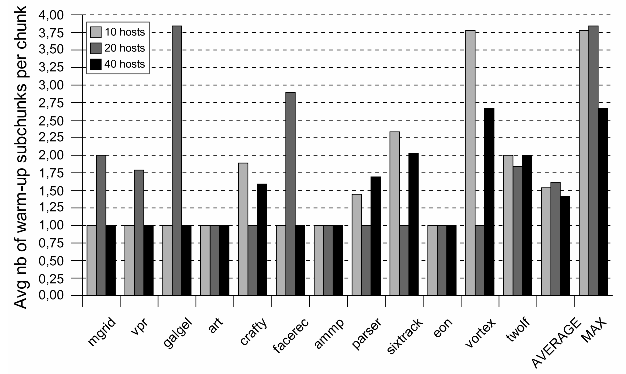

Figure 18: Average warm-up size per chunk; *reference set, Athlon*.

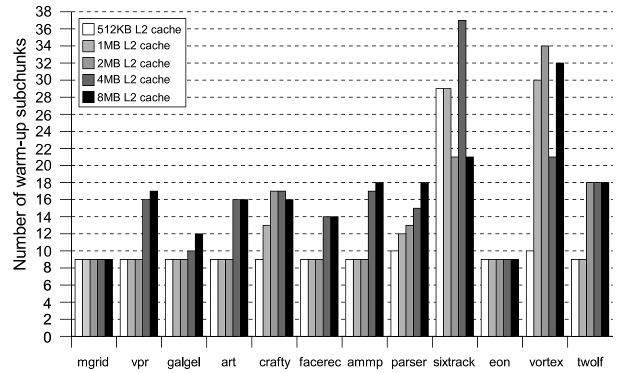

Figure 19: Influence of large memory structures on speedup and accuracy; *reference set, 10 PIII 500*.

DiST versus trace size reduction or trace sampling.

Trace size reduction. As mentioned before, trace size reduction is a simple means often used for speeding up simulation. To compare the accuracy of the trace size reduction technique with DiST, we decreased the trace size so that the simulation time is the same as DiST, i.e., 100 million traces for all benchmarks. Then, we measured the average error for each 100-million chunk within the 4-billion trace used for DiST; because the error can either be positive or negative depending on the chunk, we measured the average absolute value of the relative error over all 100-million chunks, see Figure 16. We can see that the error variation is far bigger than with DiST, between 3.28% (*twolf*) and 120.48% (*gcc*) for IPC, and between 5.74% (*art*) and 1766% (*fma3d*) for L2 miss rate; note that in many cases, when the error is large the number of events is *not* negligible, e.g., number of L2 misses. Consequently, it may be difficult to trust research results and decisions based on such (or smaller) trace sizes (note that 100-million traces are not uncommon in research articles).

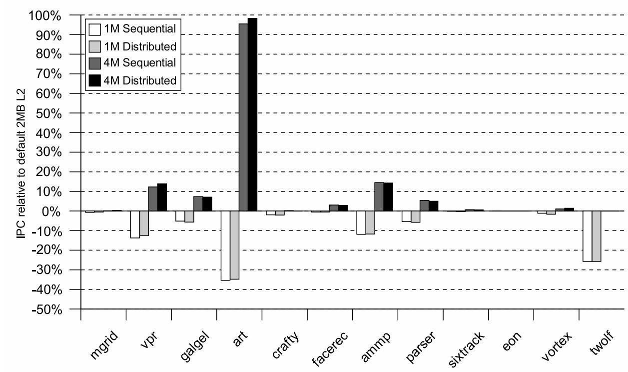

Figure 20: Varying L2 size (IPC); *reference set, 10 PIII 500.*

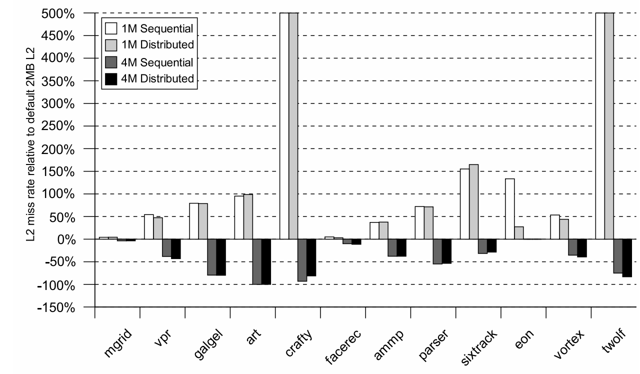

Figure 21: Varying L2 size (L2 miss rate); *reference set, 10 PIII 500.*

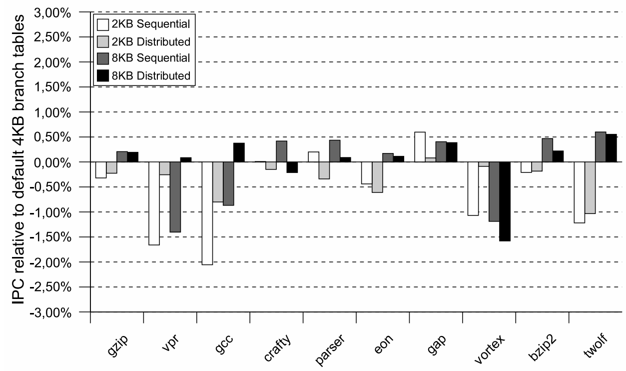

Figure 22: Varying branch prediction tables size (IPC); *reference set, 10 PIII 500.*

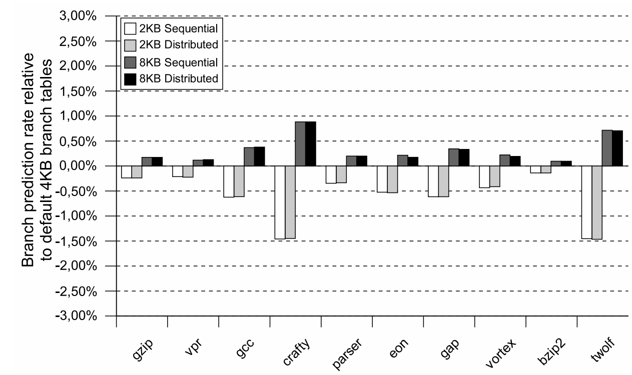

Figure 23: Varying branch prediction tables size (branch prediction rate); *reference set, 10 PIII 500.*

A recent study [21] proposes a novel approach to trace size reduction by carefully picking the starting points. The technique is efficient and accurate (3% IPC error on 100-million traces compared to full runs) except for large memory components like L2 caches (more than 20% on L2 miss rate) because it is based on small and fixed-size traces. Augmenting this technique with DiST dynamic warm-up mechanism to automatically determine the appropriate trace size has the potential to achieve both efficiency and accuracy over a large share of the design space.

Trace sampling. Trace sampling was also mentioned as another and more sophisticated technique for reducing the trace size [14]. Instead of picking a trace of T consecutive references, the trace is split into a number of randomly located intervals within a larger trace. Because the resulting simulated instructions span over a larger share of the program execution, the trace is usually more representative. On the other hand, the program must be emulated between each simulated interval which slows down simulation compared to straight trace size reduction. We have applied trace sampling by splitting the trace into 40 intervals, reducing the interval size so that the speedup is the same as DiST. The distance between two intervals is randomly choosen. Even though trace sampling proves more accurate than straight trace size reduction, Figure 17 shows that it is significantly less accurate than DiST.

Finally, note that, whenever speeding up simulation is vital, it is possible to combine trace size reduction techniques *and* DiST.

Confidence in research decisions. For instance, when varying the branch prediction table size, trace size reduction and trace sampling perform significantly worse than DiST. These techniques detect almost no amplitude variation, compare Figure 23 with Figures 26 and 25.

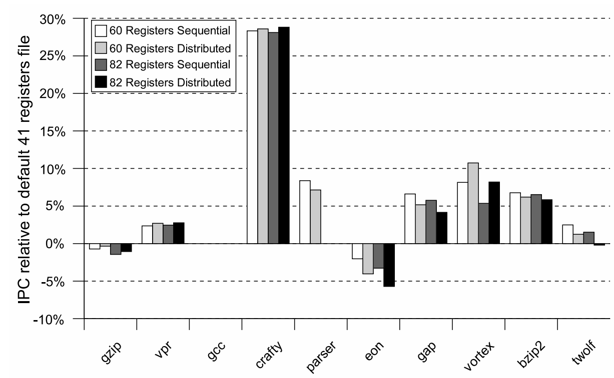

Figure 24: Varying the number of physical registers (IPC); *reference set, 10 PIII 500.*

A simple mechanism for occasional validations during the research process. All experiments show that DiST results are trustworthy and that overall, it is a significantly more trustworthy technique for speeding up simulation than trace reduction techniques. Nevertheless, when a researcher uses DiST for a particularly long sequence of analysis steps without extensive validation on traditional simulation and/or many benchmarks, there always

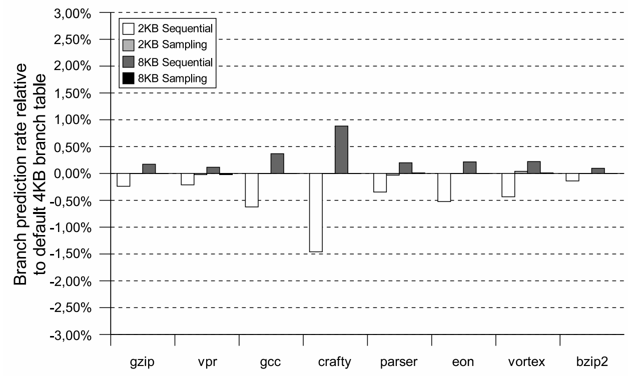

Figure 25: Trace Sampling: Varying branch prediction tables size (branch prediction rate); *1 PIII 500*.

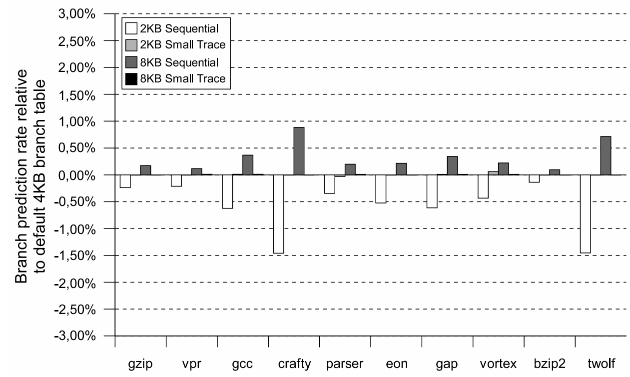

Figure 26: Trace Size Reduction: Varying branch prediction tables size (branch prediction rate); *1 PIII 500*.

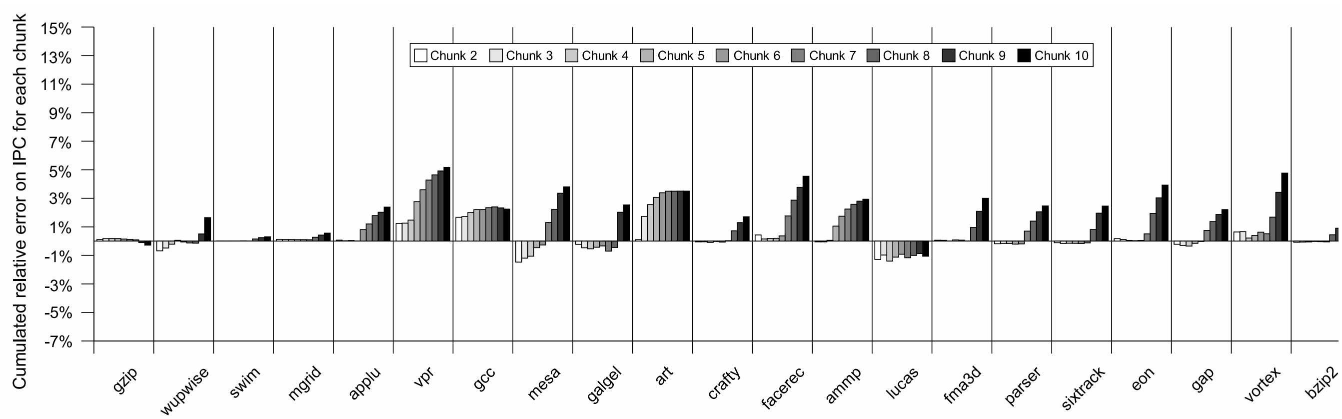

Figure 27: Evolution of global versus local error (IPC); *full set, 10 PIII 500*.

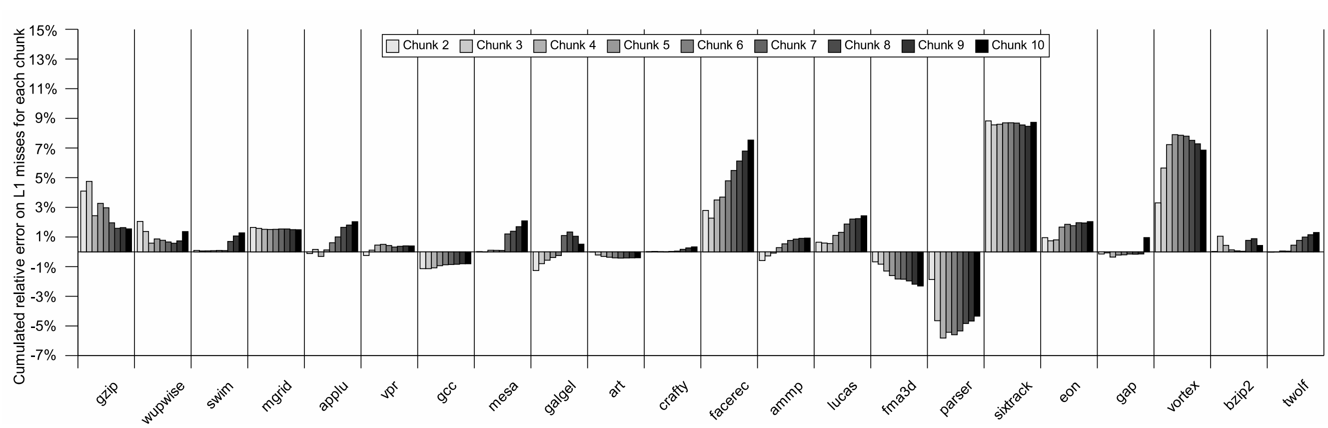

Figure 28: Evolution of global versus local error (L1 miss rate); *full set, 10 PIII 500*.

exists a tiny risk that several days of analysis are based on a trend wrongfully derived from erroneous simulation results, as when using small traces. For the sake of efficiency, DiST is fitted with a simple backup mechanism: the first chunk is always run until completion, i.e., until the end of the full trace. The researcher starts making research decisions/analyses as soon as DiST has completed, and when the full trace completes, she/he can check the true DiST error. This validation is optional: when starting a new simulation, the user can decide to kill the validation thread if it has not yet completed, in order to have all machines available for achieving the maximum speedup.

Accuracy versus speedup: error increases with the number of machines, but slowly. Because accuracy is enforced only locally, the error can increase with the number of machines. Consider chunk m. Since the convergence threshold is necessarily smaller than 100% (or convergence would almost never occur), each new chunk can slightly degrade accuracy, i.e., convergence is achieved without a perfect match between the statistics of $m-1$ and m. Over time, these errors may cumulate, and the higher m, the higher the loss of accuracy. Therefore, while the "local" error remains fixed, bounded by the local accuracy constraint, the "global" error may diverge. In practice, even though we effectively observe that, for some codes, the "global" IPC error increases with the chunk number, see Figure 27, the progression is generally very slow. Moreover, for other metrics such as the L1 miss rate, the "global" error remains almost always constant, see Figure 28.

Accuracy can be based on other metrics than IPC. Note that accuracy needs not target IPC alone: if the purpose of simulations is to analyze a specific processor component, local accuracy constraints can target the relevant statistics. We can either replace the IPC constraint by a constraint on another statistic, or add a new constraint on another statistic.

For instance, let us assume we want to focus on L2 miss rates. We have run experiments with a relaxed IPC constraint (error less than 10% instead of 2%) and we find that the average error on L2 miss rates is equal to 3.24%; then, we add a local accuracy constraint on the L2 miss rate (error less than 2%), and run again the experiments: the average global error on L2 miss rate decreases to 2.17%, and even the average IPC error gets close to what we achieved with a strict 2% error constraint on IPC. Still, when the local constraints are too numerous or excessively tight, they either have a redundant effect or decrease the speedup. For instance, when adding the local 2% L2 miss ratio constraint to the 10% IPC constraint, the L2 error is improved but the average total number of warm-up subchunks increases from 13 to 48.

6. FAST SIMULATION ENABLES NEW APPLICATIONS

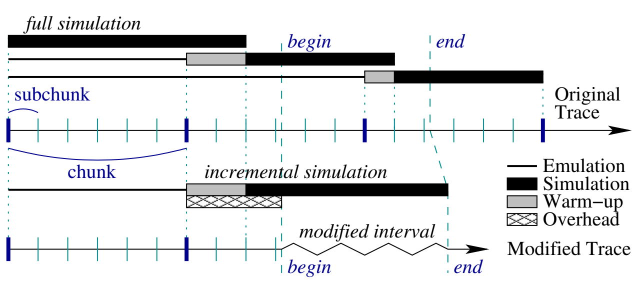

Figure 29: Iterative local program analysis and optimizations using DiST warm-ups.

Compilers are increasingly unable to cope with quickly growing architecture complexity, resulting in a rapidly increasing gap between peak and sustained performance. Currently program optimizations either rely on time profiling tools or hardware-counter based tools such as DCPI [1] and ProfileMe [6]. However, simulating program execution on the target processor architecture provides a wealth of information that can considerably help understand why performance degradations occur, how the different processor components interact and how the program behavior can be improved. Still, processor simulators are rarely used in program optimizations tasks because they are dauntingly slow. Depending on the number of available simulation machines and the speed ratio emulator/simulator, DiST can bring a simulation slowdown of several thousands back to a more acceptable few hundreds or even less. And a few additional modifications can make DiST even more appropriate to the trial-and-error process of optimizing a program. We briefly discuss one such modification in this section.

Most often, program optimization will focus on a given code section where optimizations are repeatedly applied and tested. Usually, after an initial simulation, the programmer applies (local) code transformations and then has to run again a full simulation to monitor the impact of the transformations. A few modifications to DiST enable *selective (re)simulation* of a small superset of the *modified trace interval*, see Figure 29; then, upon entering this interval, the processor state is almost the same as in the original simulation without effectively simulating prior instructions. Let us now describe the technique in more details.

The modified interval must be inferred from the exact occurrences of the bounding instructions in the original and modified traces. These instructions are simply tagged by user-inserted directives in the assembly code. A fully automatic alternative consists in relying on a binary matching tool to automatically locate the modified code segments. For instance, BMAT [24] is used to spot local changes in object code in the debugging process of a large application.

Then, we need to spot the chunk whose warm-up period immediately precedes — and does not intersect with — the modified interval. During the original simulation, we store the size of this warm-up period. To resimulate the interval, we can simply emulate until the beginning of this chunk, and then start simulating as in DiST. In order to determine when statistics become valid, the dynamically determined warm-up size is replaced with the recorded size. When simulation starts, the processor state is exactly the same as with DiST, without simulating prior chunks. The lower part of Figure 29 represents an incremental simulation run after a program transformation. Note that it is even possible to analyze the impact of the program transformation on the rest of the program by carrying on the simulation, assuming the transformation does not affect the program trace beyond the *end* point. This lightweight incremental approach is a step towards the pervasive use of accurate simulators in optimization frameworks and methodologies.

7. CONCLUSIONS AND FUTURE WORK

Because simulation speed is not just a methodology issue but can have a strong impact on design space exploration, we presented DiST, a distributed simulation technique that can speedup processor simulation while preserving a high accuracy. Accuracy is always privileged over speedup thanks to a dynamic warm-up mechanism that automatically adjusts the warm-up size of each distributed chunk to satisfy user-defined local accuracy constraints. DiST is much more accurate than the traditional trace size reduction technique used by many researchers to speed up some parts of the research process; and it is significantly more accurate than more sophisticated techniques like trace sampling. Moreover, we have experimentally shown that the technique is reliable and that research decisions based on distributed simulation are usually not affected by the slight loss of accuracy; besides, DiST comes with a backup mechanism for delayed validations. Speedup can scale with the number of available computing resources and is currently bounded by the trace size and the ratio emulator/simulator speed. Finally, the tool was designed so that it can be easily plugged to existing simulators with minimal modifications. We demonstrated a distributed version of SimAlpha and SimpleScalar which are publicly available with DiST at http://www.microlib.org/DiST.

DiST speeds up simulation and the speedup is bounded by the ratio of the emulator speed over the simulator speed. Using checkpointing, this speedup upper-bound disappears and thread behavior is not disrupted by differing system calls, so that augmenting DiST with EIO-checkpointing, for instance, has the potential of improving both speedup and accuracy.

Fast simulation enables new applications such as detailed program behavior analysis on complex processor architectures. Based on the warm-up principles of DiST, we can quickly and repeatedly analyze/modify a given code section without rerunning a full simulation. In the future, we intend to investigate further improvements of this application, particularly by fastening or even removing the emulation phase that precedes the target code section using checkpointing.

More generally, DiST is part of a broader methodology effort conducted by our research group to address several simulation methodology issues: fast and reliable simulator design using modular structures, speeding up simulations, improving simulator accuracy and using simulators as dynamic analysis tools for program optimization purposes.

8. ACKNOWLEDGEMENTS

We would like to thank the other members of the Computer Architecture Group at LRI, especially Nathalie Drach and Sami Yehia for many helpful suggestions.

9. REFERENCES

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone, July 1997.

[2] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, pages 41–49, May 1998.

[3] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, Department of Computer Sciences, University of Wisconsin, June 1997.

[4] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 195–205, Toulouse, France, 2000.

[5] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *International Conference on Computer Design*, pages 468–477, 1996.

[6] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. ProfileMe : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, Research Triangle Park, North Carolina, 1997.

[7] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *The 28th Annual Intl. Symposium on Computer Architecture*, pages 266–277, June 2001.

[8] L. Eeckhout, K. DeBosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *Int. Symp. on Performance Analysis of Systems and Software*, Liege, Belgium, April 2000.

[9] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proc. of the 2001 International Conference on Computer Design*, Austin, Texas, September 2001.

[10] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Technical Report RC20610, IBM T. J. Watson, Oct 1996.

[11] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization, International Conference on Computer Design (ICCD),*, pages 73–82, September 2000.

[12] T. Lafage, A. Seznec, E. Rohou, and F. Bodin. Code cloning tracing: A "pay per trace" approach. In *EuroPar'99 Parallel Processing*, Toulouse, France, August 1999.

[13] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proc. of the 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, San Jose, Calif., June 1988.

[14] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 248–259. ACM Press, 1993.

[15] A. Nguyen, M. Michael, A. Nanda, K. Ekanadham, and P. Bose. Accuracy and speed-up of parallel trace-driven architectural simulation. In *Proc. Int'l Parallel Processing Symp., IEEE Computer Soc. Press,*, pages 39–44, Geneva, Switzerland, April 1997.

[16] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proc. Thrird In. Symp. On High Perf. Computer Architecture*, San Antonio, Texas, February 1997.

[17] S. Nussbaum and J. Smith. Modeling superscalar processors via statistical simulation. In *PACT '01, International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, September 2001.

[18] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance - matrix-multiply revisited. In *Supercomputing 2002*, Baltimore, November 2002.

[19] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Goa, India, January 1999.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architecture and Compilation Techniques*, Barcelona, Spain, September 2001.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. of Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, Calif., October 2002.

[22] Synopsys. SystemC. http://www.systemc.org, 2000-2002.

[23] X. Vera, M. Hogskola, and J. Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, Boston, Massachusettes, February 2002.

[24] Z. Wang, K. Pierce, and S. McFarling. BMAT — a binary matching tool for stale profile propagation. *Journal of Instruction-Level Parallelism*, 2(1–6), 2000.