

Fast, Accurate Micro-Architecture Simulation

Ramkumar Srinivasan

Jeanine Cook

*Klipsch School of Electrical and Computer Engineering
New Mexico State University
Las Cruces, NM 88003, USA*

(ram,jcook)@nmsu.edu

ABSTRACT

Computers are fast becoming an integral part of everybody's life, influencing almost everything we do in our day-to-day life. The workhorse in any computer system is its microprocessor (CPU). Modern day microprocessors pack several million transistors into a tiny package that is capable of crunching numbers for a variety of applications ranging from missile guidance to gene-sequencing. Microprocessor design is becoming an increasingly challenging task due to the complexity involved.

As in other fields, simulation is extensively used in the design of microprocessors. Every aspect of a newly proposed design is first studied on simulators before any realization in silicon is made. It is estimated that detecting design flaws during the simulation phase saves millions of dollars for every newly built chip. Thus, a good simulator saves money, time and effort. One of the most popular simulators used in computer architecture research is SimpleScalar [1], which gathers a large set of processor performance statistics by 'executing' various standardized programs (benchmarks) on a simulated processor model. However, the main drawback of micro-architecture simulators like SimpleScalar is that they take an extremely long time to execute some of the contemporary benchmarks such as SPEC2000 [2]. Simulations often run into many weeks before statistics are available. Various techniques have been proposed by researchers to speed up computer architecture simulations. However, most of these methods sacrifice accuracy resulting in significant errors in the final statistics. In this research, we propose a novel method that has a potential to greatly speed-up architecture simulations without sacrificing much accuracy.

We propose to parallelize SimpleScalar. By doing so, several machines that are geographically separated can participate in the simulation of a benchmark. Simulation speed-up achieved by this method is proportional to the number of participating machines. On our tests, we achieve a speed-up of close to 10 using 16 machines with a near-zero error in the results.

Keywords

Distributed simulation, Micro-architecture simulation, SimpleScalar

1. INTRODUCTION

Simulation is a very important tool used in various branches of science and engineering. Simulations are often used in scenarios where (1) experimenting on the actual system may be disruptive or infeasible, and (2) evaluating the performance of radically new and yet-to-be-built systems is required. A good simulator provides a convenient and flexible mechanism to experiment on the system that the simulator models.

One branch in computer engineering that is driven by simulations is computer architecture. Architecture research heavily relies on simulators for the evaluation of new ideas. Quantitative evaluation of future processors is impossible without simulation. Every chip manufacturer studies many important aspects such as power consumption, thermal dissipation patterns apart from performance of a newly proposed design before any realization in silicon is done. There is however a major problem that inhibits the use of simulators. The complexity of a simulator is directly tied to the complexity of the system we are trying to model. Given the high degree of complexity of modern processors, the simulators that model them are also very complex. This level of complexity makes the simulator very slow. It is estimated that micro-architecture simulators run at a speed that is 1000 times slower than an actual processor.

SimpleScalar is an openly available micro-architecture simulator suite that is used extensively by the architecture research community. Even a complicated and extensively used simulator like SimpleScalar does not model every aspect of a processor. However, the degree of modeling done by SimpleScalar is sufficient for most architecture studies. One of the simulators in the SimpleScalar suite is *sim-outorder*. *Sim-outorder* models a superscalar, dynamically scheduled, out-of-order processor. Dynamically scheduled, out-of-order processing is the fundamental idea used in almost all modern day processors. It is a technique that allows a processor to execute the instructions of a program in a different order than that they appear in the program. This can be done as long as program correctness is maintained. *Sim-outorder* executes a program as a normal processor would and displays performance statistics of the various sub-components of the micro-architecture at the end of the simulation run.

In order to provide a common platform to evaluate the performance of a new system, various standardized pro-

grams (also called benchmarks) have been developed. Each benchmark-suite is intended for the performance evaluation of a specific type of system. For example, the *SPLASH* [3] benchmark-suite is intended for the evaluation of parallel processors, *MiBench* [4] is intended for the evaluation of embedded processors. For general purpose micro-architecture evaluation, *SPEC2000* [2] is a commonly used benchmark suite. *SPEC2000* was developed by the Standard Performance Evaluation Corporation. The benchmark-suite includes 26 applications ranging from *gcc* - a very popular open-source C compiler to *crafty* a chess playing program. The *SPEC2000* benchmarks are designed to stress the various components of the micro-architecture. Apart from *SPEC2000*, there are several other benchmark suites that contain multimedia workloads, scientific workloads, that can be used for evaluating a micro-architecture. To get a fair estimate of the performance of a newly proposed design, it is useful to study the performance under numerous benchmark-suites. This is because the inability of a benchmark to stress a particular component of the micro-architecture is offset by the the presence of other benchmarks. This however is rarely done. Often, researchers limit themselves to a single benchmark suite like *SPEC2000*. Even while using *SPEC2000*, researchers often evaluate performance by partially executing only a few workloads of the suite. Statistics collected by executing a tiny fraction of the workload are assumed to be a good indicator of the performance. This, however, is found not to be true as shown in [5]. The performance statistics are shown to vary greatly throughout the execution of a workload. Certain portions of the workload exhibit very good behavior (characterized by quick code execution) while other portions exhibit significantly poorer behavior. Various reasons have been cited by researchers to justify performance evaluation based on limited execution of the workloads. We believe that a strong contributor to this problem is slow simulators. Most of the programs in the *SPEC2000* benchmark suite execute billions of instruction and take a couple of days to execute on SimpleScalar. All the workloads of the *SPEC2000* benchmark combined execute in excess of 7.29 Trillion instructions. This translates into 4 months of simulation time on the fastest desktop available today.

The organization of the paper is as follows: Section 2 analyzes existing methods to solve the simulation speed problem. Section 3 examines the basic principle and algorithm of our method. Section 4 presents the experimental methodology and the validation results of our method. Section 5 presents our conclusions and finally, Section 6 discusses the enhancements that can be done to improve the parallel simulator.

2. RELATED WORK

Various techniques have been proposed by researchers to speed up architecture simulations. These include sampling [6, 7], Workload input reduction [8, 9] and simpoints [10, 11].

Statistical sampling: Among the first to propose this technique was Conte [6, 7]. In this method, a workloads execution is divided into many tiny chunks. A few chunks are chosen for simulation based on a selection rule. Selection is done either randomly or in a regular order by selecting a chunk and dropping a few that follow the selected chunk.

Both these selection policies make an attempt to scatter the chunks throughout the workload in an attempt to capture the whole behavior of the workload. By selecting a sufficient number of chunks, this method can result in high simulation accuracy. However, very little speed-up is achieved if the number of chunks is large. Thus, a trade-off between speed-up and simulation accuracy has to be resolved while using this method.

Workload input reduction: All *SPEC2000* workloads take an input file for processing. The execution time (and the number of instructions executed by a workload) is proportional to the input file size. The larger the input file, the longer the workloads execute. For the *SPEC2000* benchmark-suite three sets of input are available. These are reference, train and test input sets. Test input is the smallest of the three and is intended to verify that a workload executes correctly. Train input sets which are bigger than test inputs, are used for profile-driven compiler optimizations. Test and train inputs are not intended for performance evaluation. Reference input sets are the largest of the three input sets. These are meant for the performance evaluation. Since reference input files are large, performance evaluation takes a long time. It is proposed in [8] that the reference input files can be tailored and its size reduced considerably without affecting the overall workload behavior. Reduction of the input data set is done by manually clipping the reference input files. After each quantum of reduction, various statistics are compared with those obtained using the unmodified reference input set. This process of reduction and comparison is continued until a much smaller input file within tolerable error bounds is achieved. In [9] it is shown that on many workloads the reduced input set is indeed representative of the reference input sets, however, for certain workloads like *vpr* the behavior is dissimilar. Another fundamental problem with this approach is that it takes significant time and effort to come up with the reduced input set for a workload. Also, this approach is not suitable for workloads that do not take input files.

SimPoint: Extensive work was done by Sherwood et al [10, 11] to identify a few tiny chunks of a workload that sufficiently represent its entire execution. To do this, the workload is profiled using ATOM [12], a versatile code profiler. Using ATOM, a vector representing the number of times each static branch instruction is executed is formed. This is called a Basic Block Vector (BBV). BBVs are then computed for short segments of the code. The segment whose BBV closely matches the BBV of the entire workload is taken to be representative of the entire workload. This work was later extended in [11] to include multiple segments that together more closely represent the behavior of the complete workload. Statistical error obtained using multiple simpoints directed simulation is shown to be small on certain metrics like IPC. However, the shortcomings of this method are:

- Error on certain other processor statistics (Level 2 cache hit rate) can be significantly large. This is because short segments of code cannot stress large memory structure such as level 2 cache enough so as to get an accurate performance measure.

- The simulation time can be significant if the chunks appear towards the end of a workload’s execution, since it takes time for a simulator to get to the chunk. This is found to be true for the simulation points identified for most of the *SPEC2000* workloads.
- Dependence on specialized hardware/software. ATOM is the key component required for identifying the simulation points. However, ATOM is designed for Alpha machines running DEC-OSF/HP-Tru64 Operating system. These resources are expensive and may not be at the disposal of many researchers.
- Significant processing time is required to identify the simulation points of a workload.

We believe sampling/chunking is a disruptive process and can never fully represent the complete execution of a workload, no matter how cleverly done. The design and composition of the *SPEC2000* benchmark involved years of coordinated effort. The input file to the benchmarks is a result of serious debating. Chunking such benchmarks defeats the very purpose of the suite. Also, in all speed-up techniques discussed, simulation accuracy and speed-up tend to be inversely related. Trying to maximize simulation accuracy inevitably results in poor speed-ups. This coupled with the need for pre-processing of the workloads partially explain the limited adoption of these techniques in architecture research.

3. BASIC PRINCIPLE AND ALGORITHM

The need of the hour is to come up with a simulation speed-up technique that is fast, accurate and that can be readily applied for simulating any workload without requiring any pre-processing of the workloads.

We approach the problem from a different angle. We propose that paralyzing the micro-architecture simulator is an accurate and scalable answer to the simulation problem. The parallel simulator runs on several machines that are networked even if they are geographically separated. A parallel simulator is capable of executing the complete workload at a much quicker rate than the sequential simulator. Errors are expected to be low since the complete workload is executed. This is indeed found to be true. The statistical error from the parallel simulator is found to be several orders of magnitude lower than the error reported on any previously published simulation speed-up technique. The potential for simulation speed-up is limited by the the number of machines taking part in the simulation. However to achieve such speed-up, additional capabilities needs to be added to the parallel simulator. Another advantage of this technique is that it can be used readily for the simulation of future workloads as no special processing is required. The widespread popularity of the SimpleScalar simulator prompted us to parallelize it initially. On our test setup involving 5 machines, we achieve an average simulation speed-up of around 3.33 with statistical errors less than 0.001% on IPC. A team of French researchers published a similar technique [13] when we were writing this paper. Their version of the parallel simulator however results in a significantly larger IPC error of around 2.60%.

Simulator	Purpose
sim-bpred	Branch predictor simulator
sim-cache	Functional cache simulator
sim-fast	Functional simulator with no error checking
sim-safe	Functional simulator with error checking
sim-outorder	Detailed out-of-order issue, superscalar simulator
sim-profile	Profiler based on a functional simulator

Table 1: Simulators in the SimpleScalar suite. Table shows the purpose of each simulator in SimpleScalar suite

Micro-architecture simulators can be classified into two categories: Trace-driven simulators and execution-driven simulators. Trace-driven simulators take as input a trace or sequence of information collected during the execution of the workload on the target machine. Execution-driven simulators take in a program as input and execute it as a normal processor would. This makes execution-driven simulators versatile and flexible enough to model future processors.

SimpleScalar is an execution-driven simulator suite. The SimpleScalar toolkit contains several simulators. A few simulators are meant for studying particular subcomponents of the micro-architecture. For example, the *sim-bpred* simulator is used for studying branch predictors, the *sim-cache* simulator is used for studying cache behavior. Table 1 enumerates the purpose of each simulator in the SimpleScalar suite.

The *sim-outorder* simulator of the toolkit, models the complete micro-architecture. *Sim-outorder* has two modes of execution: detailed simulation and functional simulation. In the detailed simulation mode, every building-block of the micro-architecture like the branch predictor, data/instruction cache, functional units and the register update unit are all modeled. Thus, simulation statistics can be gathered on every subcomponent of the micro-architecture. Functional mode on the other hand, does not model the micro-architecture. It instead interprets and executes each instruction of the workload. Since no modeling of the micro-architecture is done in functional mode, no component level performance statistics can be obtained. The relatively small amount of processing done in the functional mode make it very fast. On a Pentium-IV 2.2 GHz machine, detailed simulation executes around 300 to 400K instructions per second while, functional mode executes around 6 Million instructions per second (about 20 times faster than detailed simulation). Functional simulation mode is often used to quickly fly-past the initialization phase of a workload. The initialization phase is that portion of the workload where the workload is performing mainly initialization and warming-up of the data structures that define the core of the workload. Performance statistics during the initialization phase are erratic and not indicative of the performance of the new design. Functional simulation can also be used for architecture independent studies such as workload characterization and measuring intrinsic data locality. Parallelizing micro-architecture simulators poses a challenging problem. This is due to the instruction level dependency that exists in all programs being executed by the simulator. That is, the execution of a *dependent instruction* that depends the outcome

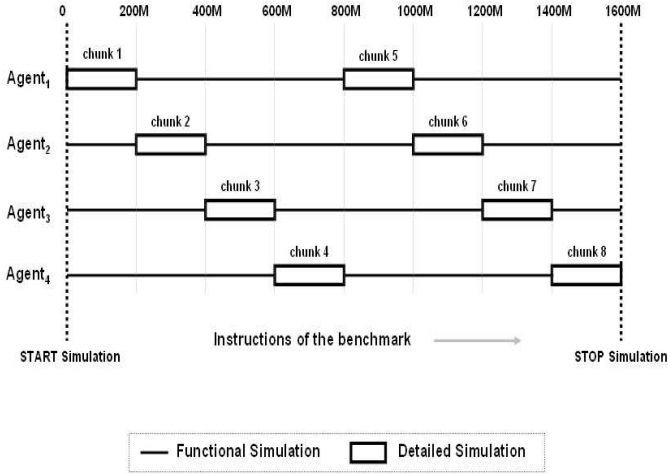


Figure 1: Basic principle of the parallel simulator. Each *agent* does a detailed simulation of a chunk, Transfers the performance statistics to the central server and quickly jumps to the next chunk by *fast-forwarding*

of a *producing instruction* can complete only after the outcome of the later is known. Thus, individual instructions cannot be scattered to different machines for processing. Parallelizing must be done at a much coarser granularity. A workload’s execution is divided into many chunks. Chunks are typically a few million instructions long. Each chunk is then distributed among the various machines taking part in the simulation. The machines taking part in the simulation are referred to as *agent* henceforth. Each *agent* does a detailed simulation of the assigned chunks and transfers the performance statistics to a central server. The server collects and concatenates the statistics from the individual *agents* to form the composite statistics.

The size of the chunks must be picked carefully. If the chunks are too large then the simulation load may not be distributed evenly across all the *agents*. This is because with large chunk size there may not be sufficient chunks for all the *agents*. On the other hand, a small chunk size implies a large number of chunks which results in a significant overhead. If we have t *agents* to execute a workload of N instructions, the optimal chunk size would be $\frac{N}{t}$. However, it is impossible to determine N apriori. It was observed that a chunk size of 200 million instructions provides good performance on the workloads we simulated. To understand the basic concept behind the parallel simulator, let us look at an example. Assuming that the number of instructions a workload executes is N ($N = 1600$ million instructions), the number of *agents* is t ($t = 4$, that is *agent*₁ to *agent*₄), and the chunk size is C_s ($C_s = 200$ million instructions). *Agent*₁ begins detailed simulation of the first 200 million instructions. *Agent*₂, *agent*₃ and *agent*₄ perform functional simulation to reach their respective chunks. This is referred to as *fast-forwarding*. The number of instructions *fast-forwarded* by *agent* _{i} ($2 \leq i \leq 4$) is given by $(i - 1)C_s$. After *fast-forwarding*, the *agents* begin the detailed simulation of the chunk. After the detailed simulation of the assigned chunk, the *agents* collect the simulation statistics in a buffer and move onto the next

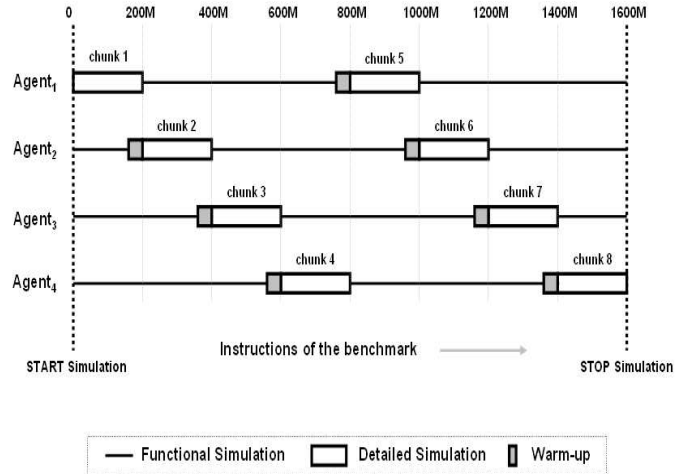


Figure 2: Incorporating *warmup*. Before the execution of a chunk, each *agent* does detailed simulation for a few instructions in order to restore an accurate state to the various components of the micro-architecture.

chunk by *fast-forwarding* $(t - 1)C_s$ (in our example, 600 million) instructions. This process continues until the workload completes execution. On completion, all the *agents* transfer the accumulated statistics to the central server. This is illustrated in Figure 1.

There is one problem with this simplistic approach. To get to a chunk, each *agent* uses the functional simulation mode. In functional mode, no modeling of the micro-architecture components are done. Thus, various structures such as cache and branch prediction tables are empty when detailed simulation of a chunk begins. This results in erroneous performance statistics for the chunks. The start of each chunk is likely to be characterized by poor behavior due to high branch mis-prediction and cache misses. Work done by Hakin and Skadron [14, 15] suggests a technique to overcome this problem. They observe that the memory reference and branch instructions just before the start of a chunk primarily determine the state of the associated structures. This implies that doing detailed simulation for a fixed number of instruction before the start of a chunk would ensure that the state of the various structures are consistent. This process is referred to as *warmup*. A slight modification to the example described before is required to incorporate *warmup*. Let W represent the number of instructions to be executed for *warmup*. *Agent* _{i} ($2 \leq i \leq 4$) would have to *fast-forward* $(i - 1)C_s - W$ instructions in order to reach their first chunk. Also, after execution of a chunk, each *agent* would have to *fast-forward* $(t - 1)C_s - W$ instructions to get to the next chunk, where t is the number of *agents* and C_s is the chunk size. This concept is illustrated in Figure 2. No performance statistics are collected during *warmup*.

Implementation: We modify the *sim-outorder* simulator of the SimpleScalar toolkit to enable switching back and forth between the functional and detailed simulation modes. Several standard tools such as MPI exist to facilitate communication between parallel programs. These tools requires

Workload	Purpose
gzip	Lempel-Ziv coding (LZ77) based compression algorithm
gcc	C Language optimizing compiler for the Motorola 88100 processor
crafty	Chess playing program
vortex	Single-user object-oriented database transaction benchmark
twolf	Lithography artwork tool used for the production of microchips
mgrid	Multi-grid Solver for a 3D Potential Field
equake	Simulates of seismic wave propagation in large basins
ammp	Models large systems of molecules usually associated with Biology
lucas	Performs Lucas-Lehmer test to check primality of Mersenne numbers
apsi	Weather prediction

Table 2: Workloads used and their purpose. The workloads shown in this table were used for evaluating the parallel simulator

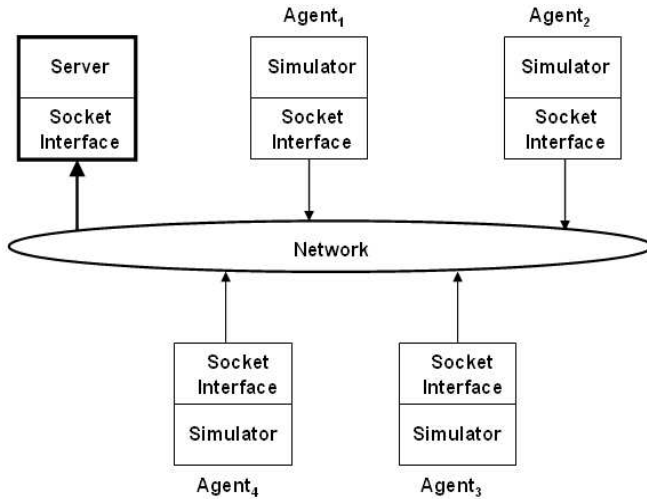


Figure 3: Block diagram of the Parallel Simulator

that a central server is capable of starting and stopping processes in all the *agents*. Typically in LAN clusters such as a Beowulf, this constraint is a perfectly valid. However on heterogeneous machines connected over the Internet, this constraint is impossible to impose. Socket based communication is free of such constraints and is useful for distributed computing environments. A socket based communication layer is interfaced to the simulator to enable communication of the performance statistics to the central server. When the simulator is started on each *agent*, the *id* of the *agent* ($1 \leq id \leq t$) and the number of *agents* taking part in the simulation *t* is passed through the command line. These parameters are used by the *agents* for *fast-forwarding*. The central server collects the statistics from all the *agents* and forms a composite. Figure 3 shows the block diagram of the parallel simulator. The algorithm used in each *agent* is shown below:

```
agent_main(id, t)
{
  fast_forward((id - 1)CHUNK_SIZE - W);
  while(1)
  {
    if not the very first chunk of the workload
    {
```

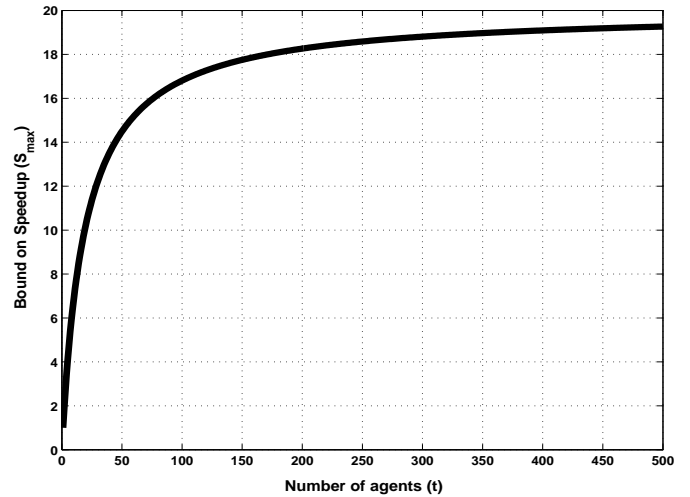


Figure 4: Theoretical bound on speed-up. The curve depicts the theoretical speed-up that can be achieved by using different number of *agents*

```
do_detailed_simulation(W);
discard_statistics();
}
do_detailed_simulation(CHUNK_SIZE);
accumulate_statistics_in_buffer();
fast_forward((t - 1)CHUNK_SIZE - W);
}
send_statistics_to_server();
}
```

The algorithm used in the server is shown below:

```
server_main()
{
  collect_statistics_from_all_agents();
  form_composite_statistics();
  display_statistics();
}
```

Theoretical bound on speed-up: We now look at the bound on achievable speed-up using the parallel simulator.

Let: N represent the number of dynamic instructions executed by a workload, t represent the number of *agents* available for simulation, W represent the number of instructions executed during *warmup*, R_d represent the simulation speed in the detailed simulation mode, R_{ff} represent the simulation speed in the functional simulation mode.

Assuming perfect load balance (that is the workload is evenly split across all *agents*), the number of instructions (L_d) for which detailed simulation is done by each *agent* is given by,

$$L_d = \frac{N}{t} + W$$

The number of instructions (L_{ff}) that an *agent* has to *fast-forward* is given by,

$$L_{ff} = N - L_d$$

The total simulation time (T_p) of the parallel simulator is given by,

$$T_p = \frac{L_d}{R_d} + \frac{L_{ff}}{R_{ff}}$$

The total simulation time (T_s) of a sequential simulator is given by,

$$T_s = \frac{N}{R_d}$$

Speed-up (S) achieved by parallelizing the simulator is given by,

$$S = \frac{T_s}{T_p}$$

The maximum speed-up (S_{max}) that can be achieved by a parallel simulator is given by,

$$S_{max} = \lim_{t \rightarrow \infty} S = \frac{R_{ff}}{R_d}$$

From the above expression it is clear that the maximum speed-up that the parallel simulator can achieve is upper-bounded by the simulation speed of the functional mode, which is typically 20 times faster than detailed simulation. We are planning to study a native execution technique to break the speed barrier. This would enable us to achieve speed-ups proportional to the number of *agents* available for simulation. Figure 4 shows a plot of the theoretical speed-up bound using different number of *agents*.

4. RESULTS

Evaluation of the parallel simulator is done on a cluster of eight machines. Each machine is equipped with dual AMD 2 GHz processors. All the machines in the cluster are connected using a gigabit Ethernet interface. Ten workloads from the *SPEC2000* benchmark suite are used for evaluating the parallel simulator. The functionality of the workloads used is described in Table 2. Testing was limited to ten workloads because simulation data from a sequential simulator, which takes a very long time, is required to make a comparative analysis between the parallel and sequential simulator. Executing these ten workloads using the sequential simulator, took 31 days.

Acronym	Meaning
IPC	Instructions Per Cycle
Addr hits	Correct branch address prediction
Dir hits	Correct branch direction predict-on
IL1 hits	Level 1, Instruction cache hits
DL1 hits	Level 1, Data cache hits
UL2 hits	Unified Level 2 cache hits
ITLB hits	Instruction Translation Look-aside Buffer hits
DTLB hits	Data Translation Look-aside Buffer hits

Table 3: Acronyms used in the error plots

Evaluation of the parallel simulator was done using two metrics: simulation accuracy and simulation speed.

Simulation accuracy: We measure the accuracy of our technique by measuring the accuracy of several simulation metrics, including the unified level 2 cache performance, which has been a poor performer in previously published speed-up techniques. For instance, the error on level 2 cache performance can be as high as 90% in the single simpoint technique [10] and 25% in the multiple simpoint technique [11]. We used the following simulation parameters for the accuracy study: Number of *agents*, $t = 5$, chunk size, $C_s = 200$ million instructions and *warmup* interval, $W = 100K$ instructions. Figures 5 and 6 show the absolute relative error on key simulation statistics between the sequential and parallel simulator. The absolute error (E_{abs}) shown in the figures is calculated according to the formula given below:

$$E_{abs} = 100 \times \frac{|M_s - M_p|}{M_s}$$

where, M_s is the actual value of the statistics obtained from the sequential simulator and M_p is the value obtained from the parallel simulator. The acronyms used on the x-axis of the figures are explained in Table 3. It is clear from Figures 5 and 6 that the error on the various statistics is small. Figure 7 shows the minimum, maximum and average error for each metric, across all ten workloads. From the figure it can be seen that the IPC error has a mean value of 2.73×10^{-4} % and a maximum of 1.19×10^{-3} %. This is about 2000 times less error than that reported for IPC in [11, 10, 7, 13].

Simulation speed-up: Figure 8 shows the speed-up achieved by the parallel simulator for executing each of the ten workloads, using 5 *agents*. Speed-up (S) is measured using the following formula:

$$S = \frac{T_s}{T_p}$$

where T_s is the simulation time of a workload on the sequential simulator and T_p is the simulation time on the parallel simulator. It is observed that an average speed-up of 3.33 can be achieved using 5 *agents*. Table 4 shows the simulation time in hours for each workload on the sequential and the parallel simulator. We next looked at the variations in simulation speed-up as the number of *agents* is increased. Figure 9 shows this variation for the *twolf* workload using the training input set. This figure also shows the theoretical bound on speed-up for each configuration. It can be observed from the figure that the ‘actual speed-up’ curve closely follows the

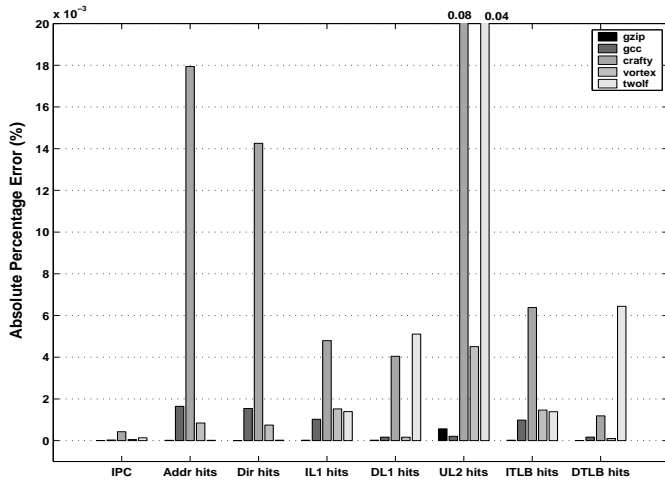


Figure 5: Error on statistics for the *SPEC2000* Integer workloads

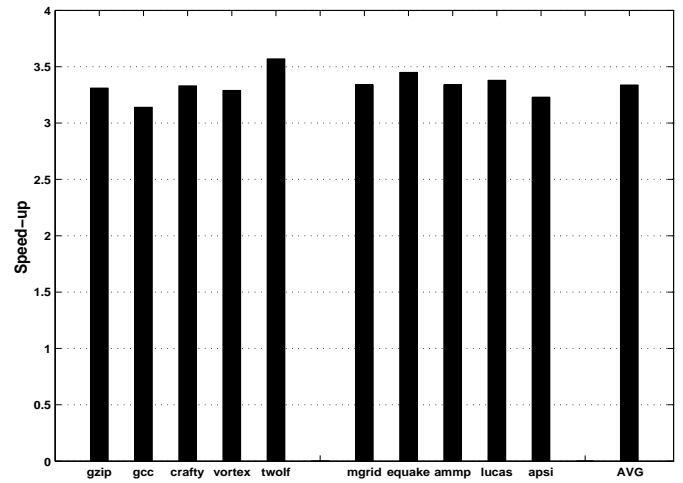


Figure 8: Speed-up achieved. Parallel simulation was done using 5 *agents*. An average speed-up of 3.33 is observed

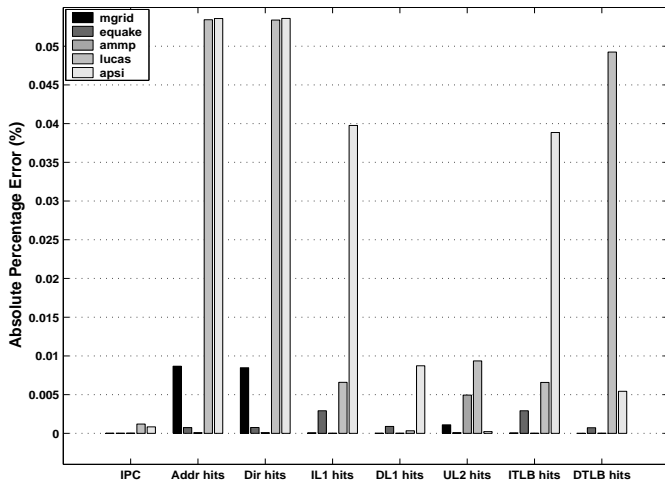


Figure 6: Error on statistics for the *SPEC2000* Floating-point workloads

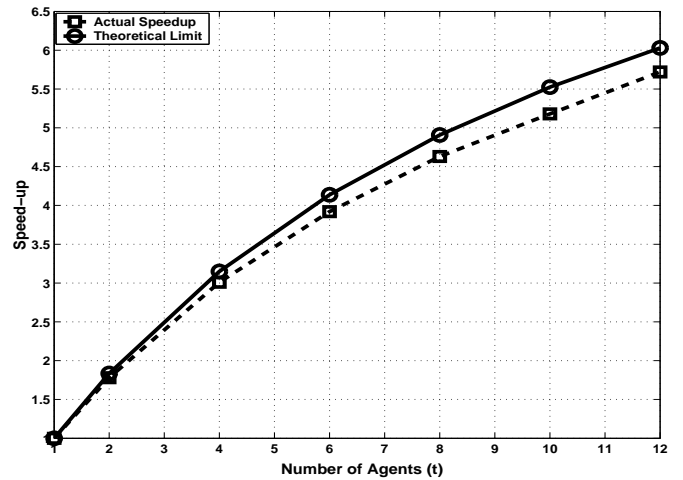


Figure 9: Speed-up with different number of *agents*. The plot shows the actual speed-up and the theoretical bound on speed-up by using different number of *agents* for the *twolf* workload

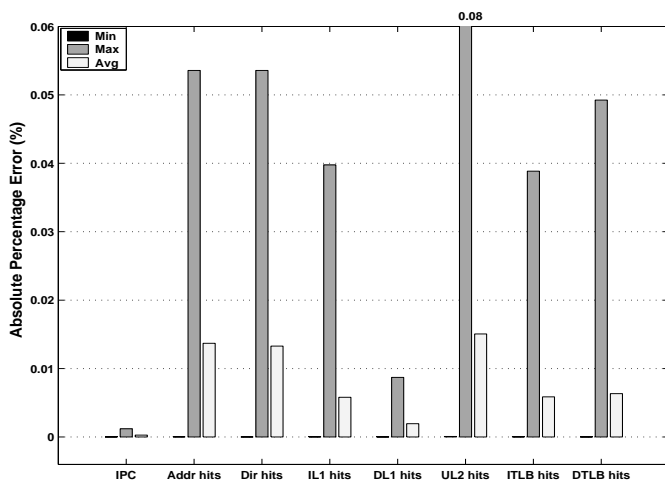


Figure 7: Minimum, Maximum and Average error

Workload	Sequential	Parallel
gzip	32.04	9.67
gcc	16.35	5.19
crafty	64.99	19.47
vortex	37.75	11.47
twolf	126.67	35.40
mgrid	134.48	40.26
equake	55.71	16.13
ammp	117.39	35.14
lucas	41.15	12.17
apsi	117.83	36.45

Table 4: **Simulation time.** Table shows the simulation time in hours for executing each workload using the sequential simulator and the parallel simulator. Five *agents* were used for the parallel simulation.

‘theoretical speed-up’ curve. Actual speed-up will always be lower than the theoretical bound due to overheads associated with communication and simulation mode switching.

5. CONCLUSIONS

SimpleScalar is a simulation toolkit that is used extensively in micro-architecture research. It collects numerous processor performance statistics by executing standardized benchmarks like *SPEC2000*. This capability is very important for evaluating new designs. The complexity of the simulator makes the simulations very slow. Executing all the workloads in the *SPEC2000* benchmark can take more than 4 months of simulation time. Various techniques have been proposed by researchers to speed-up micro-architecture simulation. Most of these techniques involve chunking or sampling the workload. In all these methods, simulation accuracy and speed-up are inversely related. Improving one metric invariably results in impairing the other metric. We propose a method to parallelize architecture simulator which we believe is an accurate and scalable answer to the simulation speed problem. We parallelized the *sim-outorder* simulator of SimpleScalar. The same technique can be used to parallelize other micro-architecture simulators. Quantitative evaluation of the parallel simulator based on simulation accuracy and speed-up was done. The accuracy of our parallel simulator is found to be in the orders of magnitude better than any previously reported speed-up technique.

6. FUTURE WORK

The theoretical speed-up bound sets the upper limit on speed-up that can be achieved using the parallel simulator. According to the bound, the simulation speed of the parallel simulator cannot exceed the simulation speed in the functional simulation mode. A method to break this speed barrier is required. One possible method is to avoid functional simulation by executing the workload natively on the *agent* doing the simulation. However, an efficient way to map the state of the native processor to the state of the simulated processor is required. Avoiding functional simulation would result in speed-up that increases almost linearly with the number of *agents* taking part in the simulation.

7. ACKNOWLEDGMENTS

This work was supported in part by the NSF/NMSU-ADVANCE and by the IBM Faculty Award Program.

8. REFERENCES

- [1] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. *Technical Report 1342, Computer Sciences Department, University of Wisconsin*, June 1997.
- [2] Standard performance evaluation corporation (spec). <http://www.spec.org>.
- [3] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.
- [4] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.
- [5] Jeanine Cook, Richard L. Oliver, and Eric E. Johnson. Examining performance differences in workload execution phases. *4th IEEE International Workshop on Workload Characterization*, December 2001.
- [6] T.M. Conte. Systematic computer architecture prototyping. *PhD Thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois*, 1992.
- [7] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1996.
- [8] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters, Volume 1*, June 2002.
- [9] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Designing computer architecture research workloads. *Computer Vol 36, Num 2*, February 2003.
- [10] Tim Sherwood, Erez Perelman, and Brad Calder. Block distribution analysis to find periodic behavior and simulation points in applications. *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.
- [11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [12] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, June 1994.
- [13] Sylvain Girbal, Gilles Mouchard, Albert Cohen, and Olivier Temam. Dist: a simple, reliable and scalable method to significantly reduce processor architecture simulation time. *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, June 2003.
- [14] John W. Haskins, Jr., and Kevin Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. *In Proceedings of the 2001 International Conference on Computer Design*, September 2001.
- [15] John W. Haskins, Jr., and Kevin Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. *In Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.