ACM SIGMETRICS Performance Evaluation Review (PER)
Special Issue on Tools for Computer Architecture Research,
Volume 31, Number 4, pages 31-35, March 2004

SIMFLEX: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture

Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch,
Roland E. Wunderlich, Shelley Chen, Jangwoo Kim,
Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk
Computer Architecture Laboratory (CALCM)
Carnegie Mellon University, Pittsburgh, PA
http://www.ece.cmu.edu/~simflex

Abstract

The new focus on commercial workloads in simulation studies of server systems has caused a drastic increase in the complexity and decrease in the speed of simulation tools. The complexity of a large-scale full-system model makes development of a monolithic simulation tool a prohibitively difficult task. Furthermore, detailed full-system models simulate so slowly that experimental results must be based on simulations of only fractions of a second of execution of the modelled system.

This paper presents SIMFLEX, *a simulation framework which uses component-based design and rigorous statistical sampling to enable development of complex models and ensure representative measurement results with fast simulation turnaround. The novelty of* SIMFLEX *lies in its combination of a unique, compile-time approach to component interconnection and a methodology for obtaining accurate results from sampled simulations on a platform capable of evaluating unmodified commercial workloads.*

1. Introduction

Computer architects have long relied on software simulation to study the functionality and performance of proposed hardware designs. Despite phenomenal improvement in system performance over the last decades, the disproportionate growth in hardware complexity has steadily increased software model complexity and eroded simulation speed. Research studies of large-scale server systems have shifted focus from scientific applications [12] to commercial workloads [1]. This shift has forced simulation tool developers to expand the scope of their simulation tools to model system components beyond the processor and memory hierarchy, and support execution of unmodified operating systems with commercial workloads for which source code is unavailable [6][10]. The increasing complexity of both the system model and target workloads has elevated continued development of monolithic simulators [8][9][10] to a task of herculean proportions. Moreover, uniprocessor simulations of highly parallel systems are so slow that researchers must base conclusions on simulations of only fractions of a second of native execution time[11][13].

This paper introduces SIMFLEX, a component-based framework for creating timing models of uni- and multiprocessor server systems running commercial applications. SIMFLEX addresses the problem of exploding model complexity through a component-based approach to model construction, inspired by the Asim simulator[4]. SIMFLEX ensures that accurate and reliable performance results can be obtained quickly by integrating the SMARTS methodology [13] for representative simulation sampling with novel implementation techniques for eliminating the runtime overheads that arise from component-based software construction.

The following are the key features of SIMFLEX:

- **Full System Simulation**. SIMFLEX leverages the technology of the commercially-available *Simics* simulation tool [6] to provide functional execution of unmodified commercial operating systems and applications. SIMFLEX provides a framework for rapidly building timing models which augment the system emulation performed by *Simics.*
- **Compile-time component interconnection**. SIMFLEX takes a novel approach to the interconnection of simulator components designed to eliminate the runtime overhead of modular software design. SimFlex takes advantage of generic programming features of C++ to express component interconnection at compile-time. This enables the compiler to perform optimizations across component boundaries.
- **Simulation Sampling**. SIMFLEX applies the SMARTS methodology [13] for choosing and rapidly measuring a representative sample of each workload. SIMFLEX extends SMARTS to multiprocessor simulations, and provides support for the development of the code for warm-

ing model state that is essential to achieving unbiased measurement with SMARTS.

The remainder of this paper describes the design and impact of each of the key features of SIMFLEX.

2. Full-System Simulation

Until very recently, simulation tools for studying server architecture [7][8][9] have focused on the study of scientific applications, such as the SPLASH-2 benchmark suite [12]. In recent studies, the server architecture performance evaluation community has shifted focus to commercial applications, such as database management systems and web servers [1]. With this shift in focus, a new emphasis has been placed on full-system simulation. With scientific workloads, overall system performance is often governed by small kernels which stress CPU features such as floating point performance, or memory system bandwidth. Operating system code and peripheral devices have only a second-order, if any, effect on overall system performance. With commercial applications, however, operating system and I/O performance are first-order determinants of system performance, and must be included in the software model.

SIMFLEX is built on top of the *Simics* simulation environment to provide functional emulation of a uni- or multiprocessor system and associated peripheral devices [6]. *Simics* models the complete instruction set architecture and peripherals of a target system in sufficient detail to boot an unmodified operating system and run commercial applications. When run alone, *Simics* assumes a simple timing model where all instructions and memory accesses take a uniform amount of time. SIMFLEX adds timing to *Simics*: *Simics* provides a stream of fetched instructions to SIMFLEX, and SIMFLEX models system timing and controls the advance of time in *Simics*.

Simics can emulate a wide variety of systems and instruction set architectures (x86, SPARC, etc.). ISA-specific parts of SIMFLEX are isolated in a single component, making it easy to retarget SIMFLEX to provide a timing model for any ISA supported by *Simics*. To date, SIMFLEX has been used to model both x86- and SPARC-based uni- and multiprocessor systems.

3. Component-based Design

As the scope of a detailed software model increases, so does the complexity of the software itself. Simply understanding the model requires a considerable investment of time and the learning curve that must be climbed before starting research with monolithic simulation tools is steep. The keys to successful design of a large-scale timing model are *abstraction* and *composability*. When working with a complex tool, a researcher must not be forced to understand the intricate details of each part of the model at all times. Rather, model detail must be hidden by layers of abstraction which simplify details irrelevant to the problem at hand. A complex model should be composed of abstracted pieces whose general function can be understood at a glance.

SIMFLEX is designed as a framework for connecting model components. The conceptual design and terminology of SIMFLEX follows that of Asim, a component-based simulation tool developed at Intel[4]. Each SimFlex component models a part of the system. Generally, these components correspond directly to parts of the hardware being modelled, for example, a level of cache hierarchy, or a cache-coherence protocol engine. Other components are pure software constructs, for example, the “feeder” component which fetches instructions from *Simics*, or components which collect traces of memory transactions for offline analysis. A SIMFLEX simulator is a collection of components connected together in a hierarchical fashion as specified in what is called a *wiring* description. SIMFLEX is unique in that these wiring descriptions are C++ code which, when fed to the compiler, produce a custom simulator binary reflecting the desired wiring.

3.1. Compile-time Interconnection

The concepts of abstraction, modularity and composability are not new innovations of SIMFLEX. Indeed, these are the very foundations of successful software engineering, and should be employed by any software development effort of the size and scope of a detailed full-system timing model. However, many traditional software development approaches to modularization, such as object-oriented programming, incur a performance overhead when used to compose many components. In Asim, components are interconnected by named wires. When one component wishes to send data to another, the component writes to the wire. This data is routed to the receiving component by looking the wires name up in a global hash table. For large hardware models, which can have hundreds or thousands of signals, these hash table lookups are a noticeable fraction of total simulation time, as much as 20%[5].

In SIMFLEX, components are interconnected at compile time, rather than at run time. SIMFLEX takes advantage of C++’s *template* generic programming facilities to describe components. Each component is written with its connections to other components, called *ports*, left as unspecified C++ template parameters. The description of these ports specifies the nature and direction of data and control flow between components. Components can exchange arbitrarily complex data types, for example, a description of a memory transaction or a type representing a CPU instruction with associated functions for retrieving

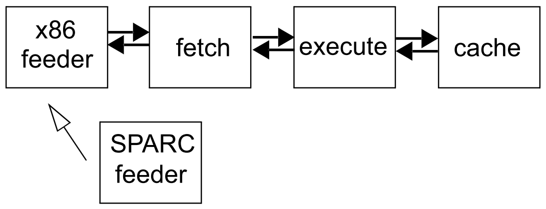

Figure 1. A Simple SimFlex Simulator.
Recompiling with the SPARC feeder binds the fetch function call to code in the SPARC feeder, and transparently changes the instruction data type exchanged by all components.

the inputs and outputs of the instruction. To create a simulator from a collection of components, the researcher writes a wiring description in highly stylized C++ code which concisely lists the components used in the simulator and how their ports are interconnected. When fed to the compiler, this wiring file results in the instantiation of component templates with the specified connections.

The strength of this approach is that each connection between two components results in direct function calls between the components at run time. Figure 1 shows an example of a simplified SimFlex simulator with four components, with two alternative versions of the feeder component for x86 and SPARC. When the fetch component is wired to the x86 feeder component, it calls a function on the feeder component which returns the fetched instruction. By making a simple change to the wiring and recompiling, the fetch component will instead call a similar function on the SPARC feeder. This change also transparently changes the instruction data type exchanged between all components to represent SPARC instructions. The template facility of C++ allows the same component code to interact with instructions from both feeders, despite the large differences between the data types used to describe instructions in each feeder.

Interconnection of components via function calls could also be accomplished at run time by clever use of C function pointers or C++ virtual functions. However, an advantage of the C++ template approach is that, since the function call destination is known at compile time, the compiler can optimize the call. Analysis of compiled SimFlex simulators reveals that nearly all function calls across ports are inlined by the compiler, often several levels deep and across multiple components. This kind of compiler optimization is not possible if components are interconnected at run time.

3.2. A Library of Reusable Components

Component-based model design enables development of a library of reusable model components to represent hardware structures common across many different architectures. Moreover, multiple versions of a component can model the hardware structure at various levels of detail. This allows researchers to trade off accuracy for simulation speed for each component in the system. For example, we have developed a simple memory component that applies a constant latency to each memory access, and a more complex and slower component that models DRAM bank conflicts. Each experiment can choose to employ the fast, simple model or more accurate and slow model.

The initial focus of research with SimFlex has been on adding new hardware components to distributed shared memory nodes to predict future memory requests and initiate coherence transactions in advance of demand requests by processing nodes. In order to support this research, we require highly flexible and detailed models of cache coherence protocols and the hardware implementations of these protocols. Thus, we have developed a detailed simulation of a microcoded coherence engine, based on the design of the coherence engines of the Piranha prototype from Compaq [2], and cache models which support a rich bus protocol sufficient for directory-based and snoopy coherence protocols. New coherence mechanisms can be specified in the abstract and intuitive microcode language that the coherence engine employs, without the need to modify the simulator code. Since the focus of this research is on memory system behavior, we have a correspondingly complex and slow model of memory system components. However, we use a simplified in-order CPU model to save development and debugging time, and accelerate simulation speed.

4. Fast and Accurate Measurement

The disadvantage of detailed software modeling of a hardware system is the enormous slowdown of simulation relative to the modelled hardware. Detailed uniprocessor simulators, such as SimpleScalar [3] are 4000 times slower than the modelled hardware. Multiprocessor simulators, such as Rsim [8], are even slower, and suffer the penalty of simulating parallel hardware nodes in series on a single host. These low simulation speeds render it impossible to simulate complete commercial workloads from beginning to end.

To mitigate prohibitively slow simulations, researchers often use abbreviated instruction execution streams of benchmarks as representative workloads in design studies. More than half of the recent papers in top-tier computer architecture conferences presented performance claims extrapolated from abbreviated runs. Researchers predominantly skip the initial 250 million to two billion instructions and then measure a single section of 100 million to one billion instructions. However, this technique rarely captures representative behavior.

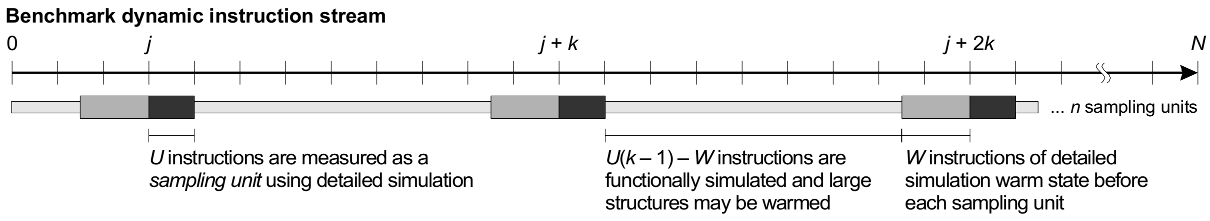
Benchmark dynamic instruction stream
0
j
j + k
j + 2k
N
... n sampling units
U instructions are measured as a sampling unit using detailed simulation
U(k – 1) – W instructions are functionally simulated and large structures may be warmed
W instructions of detailed simulation warm state before each sampling unit

Figure 2. Systematic sampling in SMARTS.

4.1. The SMARTS approach

In [13], we proposed the *Sampling Microarchitecture Simulation* (*SMARTS*) framework which applies statistical sampling theory to address prohibitively low simulation speeds and the inaccuracy of using non-representative samples. Unlike prior approaches to simulation sampling, SMARTS prescribes an exact and constructive procedure for selecting a minimal subset from a benchmark's instruction execution stream to obtain performance estimates with a desired confidence interval. SMARTS uses a measure of variability (coefficient of variation) to determine the optimal sample that captures a program's inherent variation. An optimal sample generally consists of a large number of small sampling units. Unbiased measurement of sampling units as small as 1000 instructions is possible by applying careful *functional warming*—maintaining large microarchitectural state, such as branch predictors and the cache hierarchy—during fast-forwarding between sampling units.

The SMARTS procedure details how to apply systematic sampling to choose an optimally small sample to estimate performance metrics, such as cycles per instruction (CPI), with a desired degree of confidence. In SMARTS, a sampling unit is defined as U consecutive instructions in a benchmark's dynamic instruction stream such that the population size N is the length of the stream divided by U. Changing the size of each sampling unit affects the required sample size, n, since estimating CPI at a given confidence is directly proportional to the square of the population's coefficient of variation, $n \propto V_{CPI}^2$. The coefficient of variation V_{CPI} decreases due to averaging effects as U is increased, resulting in fewer sampling units required to achieve an acceptable estimate. In [13], we demonstrate that choosing a sample with $U = 1000$ results in a minimal number of detail-simulated instructions to achieve estimates with a desired confidence. Typical benchmark applications will require a sample size of approximately $n = 10{,}000$ units to achieve 99.7% confidence of $\pm 3\%$ error.

The challenge in applying SMARTS in a practical simulator is in developing techniques to quickly skip past the portion of each workload between the many sampling units, and then computing the correct microarchitectural state prior to detailed measurement of each sampling unit. *Simics* simulation, with SIMFLEX disabled, provides a very fast functional simulation mode, but leaves microarchitectural state (e.g., cache hierarchy, branch predictors and target buffers, or pipeline state) unchanged. Stale microarchitectural state introduces a large bias in the measurement of individual sampling units and, consequently, the final estimate. We have observed stale-state induced bias as high as 50% for sampling units of 10,000 instructions.

We address this stale-state bias with a two-tier strategy for warming model state. For model state which has a long history, such as caches, we update the cache state during functional simulation, an approach we call *functional warming*. By continuously warming microarchitectural state with very long history, we can analytically determine a bound on the *detailed warm-up* required to initialize the remaining state. Figure 2 graphically illustrates how SMARTS alternates between functional warming of $[U(k-1)-W]$ instructions, detailed warm-up of W instructions (without measurement), and detailed simulation and measurement of U instructions. [13] details the procedure for determining a sampling rate k to achieve a desired level of confidence in the resulting estimate. We believe functional warming with brief detailed warm-up is the most cost-effective approach to achieve accurate CPI estimation with simulation sampling.

4.2. Multiprocessor SMARTS

Simulating multiprocessor server systems with SIMFLEX presents new challenges to simulation sampling. Selecting systematic samples from a single-processor program execution stream is a well-defined and straightforward procedure. A multiprocessor program execution, on the other hand, is comprised of multiple instructions streams with asynchrony and non-determinism among them. A key challenge for SIMFLEX lies in acquiring a sample measurement that is free of distortion from these effects. In addition, because the emulation phases of SMARTS do not capture timing information, another challenge is in approximating the relative progress of the different processors.

The most important performance metric for multiprocessor systems is total program run time, which, in turn, can be used to determine other metrics. To apply sampling to this problem, we focus on multiprocessor program

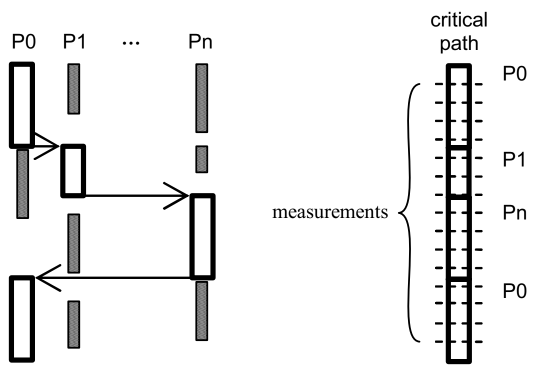

Figure 3. Sampling the critical path.

execution along the *critical path.* In typical parallel execution, at any given moment, there are a small number of processors that are responsible for generating critical results for other processors. These critical path processors set the pace for the entire system via interlocking synchronization primitives. To estimate the total run time of a multiprocessor program, we only need to sample the program executing on the critical path processors during the cycle-accurate measurement phase. Figure 3(left) depicts an example execution of a parallel program and its critical path. Figure 3(right) depicts how cycle-accurate measurement along the critical path reduces to a (uniprocessor-like) interleaving of measurements across the processors. The behavior of the off-critical-path processors does not contribute to the determination of overall runtime. Synchronization along the critical path, and between critical and non-critical paths, ensures that the relative progress on each path seen during fast-forwarding is representative of a native execution. This ensures that measurements of the critical path processors are not perturbed by incorrect relative progress on the non-critical path. We are now working to evaluate the effectiveness of the critical-path-driven sampling approach with the SIMFLEX infrastructure.

5. Conclusion

The key to successful simulation studies of large-scale multiprocessor systems running commercial workloads is to have a solid foundation for rapid comprehension and development of timing models, and a rigorous methodology for obtaining representative measurement results quickly and reliably. SIMFLEX leverages the latest commercial technology, *Simics*, to execute complex workloads with minimal development effort; employs a low-overhead component-based design to accelerate model comprehension and development; and applies a rigorous statistical methodology for ensuring accurate measurement results with minimal simulation turnaround time.

Acknowledgements

This work was funded in part by grants and equipment from IBM and Intel corporations, the DARPA PAC/C contract F336150214004-AF, and an NSF CAREER award.

References

[1] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[2] L. Barroso, K. Gharachororloo, R. McNamara, S. Qadeer A. Nowatzyk, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture base on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000. Primary Piranha Reference.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[4] J. Emer, P. Ahuja, E. Borch, A. Klauser, Chi-Keung Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, February 2002.

[5] Joel Emer. Personal communication., March 2003.

[6] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg amd Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[7] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 27–36, June 2002.

[8] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Third Workshop on Computer Architecture Education*, February 1997.

[9] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. May 1993.

[10] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchell, and Anoop Gupta. Complete computer simulation: The simos approach. *IEEE Parallel and Distributed Technology*, 1995.

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

[12] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[13] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.