An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation

James Donald and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{jdonald, mrm}@princeton.edu

Abstract— **Multiple core designs have become commonplace in the processor market, and are hence a major focus in modern computer architecture research. Thus, for both product development and research, multiple core processor simulation environments are necessary. A well-known positive feedback property of computer design is that we use today's computers to design tomorrow's. Thus, with the emergence of chip multiprocessors, it is natural to re-examine simulation environments written to exploit parallelism.**

In this paper we present a programming methodology for directly converting existing uniprocessor simulators into parallelized multiple-core simulators. Our method not only takes significantly less development effort compared to some prior used programming techniques, but also possesses advantages by retaining a modular and comprehensible programming structure. We demonstrate our case with actual developed products after applying this method to two different simulators, one developed from IBM Turandot and the other from the SimpleScalar tool set. Our SimpleScalar-based framework achieves a parallel speedup of 2.2X on a dual-CPU dual-core (4-way) Opteron server.

I. Introduction

High-level simulation has been a primary means of design and planning for all modern uniprocessor and multiple-core processor designs. Although many multicore simulators exist today, questions remain regarding how to best develop these simulators to take advantage of existing platforms. Computer design is one of the few fields with a positive feedback property, of designing tomorrow's computers using today's. In order to best utilize multithreaded and multicore processors, these simulators should make use of parallelism.

Writing parallel programs is much more difficult and costly than sequential programming, and architectural simulators have not been thought of as an obvious exception [17]. Thus, many existing multicore simulators are written merely in a conventional sequential manner. Extending uniprocessor simulators to model CMPs had typically involved auditing all code to convert most variables into corresponding arrays, converting the inner simulator loop to iterate over these arrays or even arrays of arrays, and modifying procedures that act upon these variables in a line-by-line manner. For example, the following branch predictor declaration in Turandot:

```
char bp_cond_btable[BP_COND_SIZE];
```

would be expanded as

```
char bp_cond_btable[MAX_CORES][BP_COND_SIZE];
```

Correspondingly, all code that references this variable is required to use an additional index, and all code calling functions that reference this variable would have to pass in an additional index. This process would be repeated for hundreds of such variables across the simulator's entire code base.

Such schemes are time-consuming and obfuscate the overall simulator structure. To avoid inelegant array styles, using object-oriented or structure types could keep the program format somewhat modular. However, with variables spread across multiple source modules the code must first be wholly restructured to aggregate these variables. Thus, the source code of the final product will be dramatically modified regardless. Aside from being tedious and inelegant, this method does not provide any clear means to parallelize the simulator.

In this paper we propose an alternative technique that enables multicore support and parallelization with less programming effort. Our method creates a parallel multicore simulator with an execution design that mirrors the parallel nature of physical multicore processors. We accomplish this with relatively minor modifications to uniprocessor simulators, rather than the difficult two-stage process of first revamping a uniprocessor simulator to support multicore modeling then later restructuring for parallelism. Our design can be characterized as follows:

- Each simulated core is represented with a single POSIX thread containing its own main loop.
- Synchronization between these pthreads is done only when communication is necessary, reflecting properties of isolation in real multicore systems.
- The development process does not require an existing multicore simulator, but can and has been directly applied to extend uniprocessor simulators.
- The resulting simulators achieve parallel speedup on multiprocessor platforms, while retaining compatibility with uniprocessor environments.

Furthermore, we show this involves significantly less coding effort than conventional methods while retaining modularity and the majority of the codebase of the original simulator.

There has been much prior work on parallel simulation. As one recent example, Penry et al. have demonstrated other methods which utilize automated parallelization of a modeling framework [17]. Their method uses a behavioral model on a framework such as LSE [21] or SystemC [10]. Structural and behavioral models have many uses and have already been used for commercial processor development [7], but in reality this is done not as an absolute replacement for fast conventional high-level simulators. Rather, such simulators serve as one tool in a product development timeline as a complement to other simulators by trading speed for model accuracy. Fast simulators written in sequential languages have remained essential and are unlikely to disappear anytime soon.

We demonstrate our case with two usable simulators. The first is Parallel Turandot CMP (PTCMP) which extends from Turandot [14]. The second is based on the SimpleScalar framework [4]. Although Turandot is trace-driven while SimpleScalar is execution-driven, our methodology readily applies to both. We believe this should encourage similar development to other architectural simulators.

In addition to the benefits of modularity and ease of development, our method succeeds in boosting simulation performance, a major goal for any parallel application. We characterize the speedup of our two simulators on a real multicore system.

Manuscript submitted: 27 June 2006. Manuscript accepted: 27 July 2006. Final manuscript received: 2 Aug. 2006.

Posted to IEEE & CSDL on 8/3/2006
10.1109/L-CA.2006.14

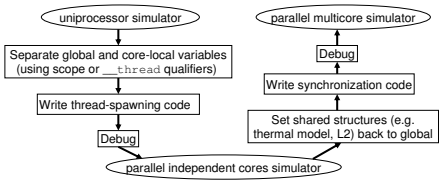

Fig. 1. Steps to develop a parallel multicore simulator. The parallel independent cores simulator without synchronization is a stable intermediate for debugging purposes.

Section II gives an overview of our technique and constructs for simulator parallelization. Section III describes the two implementations utilizing our techniques while Section IV characterizes their simulation performance. Section V discusses related work and Section VI concludes.

II. Overview of Technique

A. Parallel Structure

While writing parallel programs is generally difficult, specifying a parallel structure for a multicore architectural simulator turns out to be more natural compared to other applications. Parallel simulation is a vast field employing countless techniques, but instead of an adequate combination of methods for all problems, various techniques are best applied only on an application-specific basis [8]. Fortunately, the problem of parallelizing multicore architecture simulation is reasonably controlled in terms of symmetry and isolation. In this paper we show it can achieve significant parallel speedup without employing overly complex parallel programming techniques.

Our approach uses POSIX threads (pthreads) in which a pthread is made responsible for each modeled core. The original uniprocessor simulator effectively clones itself into duplicate cores by calling `pthread_create`. Each core then runs its own simulation in parallel with other cores. This way, the simulator code retains the same structure as the original uniprocessor simulator. Developers focusing only on the processor core can remain mostly unaffected by the encapsulating parallel multicore structure.

Chidester and George demonstrate a distributed simulator developed by parallelizing SimpleScalar with a message-passing interface (MPI) [5]. They examine several issues for optimizing communication costs in a distributed environment. Our work instead parallelizes simulators to run on a shared memory host. This avoids many of the complex issues of parallelizing for distributed hosts and allows the programmer to use most of the original simulator codebase while still providing significant parallelism.

Variables may fall in the category of being core-specific or shared. For example, since our processor model shares the L2 cache, this variable should not be cloned for each core. Core-local variables are inherently cloned by our pthread design. Specifically, local variables in the pthread starter function are cloned upon thread creation. On the other hand, variables that need not be cloned are declared in the parent function or global storage. Lastly, in order to deal with core-specific variables that are defined as global in the original uniprocessor simulator, we use the technique described in Section II-C.

B. Shared Resources and Synchronization

A simple method for synchronizing resources is to use barriers. For example, to calculate temperatures dynamically the HotSpot [11] thermal modeling tool requires power data at fixed intervals. This is implemented using a conventional barrier that is invoked at coarse-grain intervals (in our case 10,000 cycles).

Even without temperature modeling, shared resources pose a fundamental need for synchronization. Our method mirrors sharing and isolation in physical chip multiprocessors. Ensuring ordered accesses to the shared L2 cache is only required when cores request so. This is also a barrier, but is invoked not at fixed intervals but rather only in the event of accessing a shared resource. In our simulators, when an L1 cache miss occurs we impose the following check before accessing the shared L2 cache structures:

```
for i = 0 to num_cores-1
     if i != core_id && cycle[i] < cycle[core_id]
         lock, wait, restart loop when woken
     if i < core_id && cycle[i] == cycle[core_id]
         lock, wait, restart loop when woken
```

The wake signal is sent anytime another core's cycle count is incremented. Ultimately, the above condition is that a core cannot access a shared resource if any other cores have not yet passed that timestamp. Thus, synchronization is done on a per-cycle basis, although invoked only when shared resources are accessed.

On a distributed environment such as a cluster utilizing message passing, synchronization at the cycle level would be prohibitively expensive. Chidester and George work around this in their MPI implementation by introducing a time-slip quantum to improve communication performance [5]. On a shared memory platform, however, we find that cycle-level synchronization is adequate, providing speedups as demonstrated in Section IV, and well worth its simulation accuracy and simplicity.

C. Thread-Local Storage

A characteristic of SimpleScalar, and many other simulators written in sequential languages, is that much of the simulator state is shared across global variables. Since many of these variables should instead have multiple copies to reflect the states of multiple cores, this poses an initial problem. Fortunately, we are able to exploit a language construct known as thread-local storage (TLS) [20].

In gcc, this feature is invoked using the `__thread` keyword before the declaration of any global or static local variable to instruct that it will be automatically cloned into a local copy upon thread creation. This provides a single keyword to manually distinguish variables that represent properties of individual cores (local) versus variables that should be shared among all cores. Thread-local storage is supported on many development platforms such as gcc, Microsoft Visual C++, Borland C/C++ Builder, and Intel C/C++ compiler [20]. TLS is not, however, a central tenet of our methodology, but rather a useful implementation trick specific to multithreading in languages with global variables such as C and C++.

D. Putting It Together

Using all techniques described in this section, a stepwise development process is shown in Figure 1. This design flow is simpler than the general problem of converting any array-based sequential program to use multiple threads. Specifically, this involves cloning a program rather than partitioning, and a stable intermediate can be debugged before implementing synchronization.

Although debugging multithreaded programs is notoriously difficult, our method partitions development by separating core-specific and multicore interface variables. Bugs in the multicore model are limited to routines acting on shared resources guarded by the synchronization condition, while the process of debugging core-only features can be simplified by

running only one core. These methods also give the developer the option to extend the parallel simulator to model other sharing options besides the L2 cache.

III. Implementations

A. Turandot (PTCMP)

Our first parallel multicore simulator is Parallel Turandot CMP (PTCMP), based on Turandot [14]. PTCMP has already been used in [6]. For power and thermal modeling, it is integrated with Powertimer [2] and HotSpot version 2.0 [11].

The control flow of PTCMP is very similar to that of parallel programs that run a fixed number of pthreads. One pthread for each core is generated at startup, and all threads are joined at the end.

The original Turandot's characteristic program structure involves storing almost all core variables on a single stack. This allows us to conveniently take advantage of variable cloning through the use of pthreads. As described in Section II-C, TLS is necessary only if variables to be cloned reside in global storage. Because each local variable in the function spawned by calls to `pthread_create` is already effectively cloned per core, our latest version of PTCMP does not require any thread-local variables hence it can be built using a compiler without TLS support.

PTCMP also supports simultaneous multithreading, parallel benchmarks (used in [6]), heterogeneous cores, and frequency scaling all within its parallel framework. In terms of functional correctness, PTCMP was written to exactly match the timing and power statistics of Turandot CMP [12]. This is possible because our method does not limit modeling accuracy worse so than in sequential simulators.

The parallel benchmark support requires modeling both synchronization and MESI cache coherency. These are handled using the same method as for the shared L2 cache. Although fairly large development tasks, our simple synchronization method ensured these to be not any more difficult to develop and debug than in the case of a sequential simulator.

Turandot is a purely trace-driven simulator. In the following subsection, we show that our methodology can be readily applied to an execution-driven simulator.

B. SimpleScalar

We developed a parallel version of SimpleScalar using lessons learned from the development of PTCMP. Because Turandot is PowerPC-based, we chose the PowerPC ISA version of SimpleScalar [19] in order to remain consistent in our comparisons and for compatibility with our toolchain. To best match the synchronizing penalties used in PTCMP, we required a thermal model, which inherently requires a power model. Thus, we began by extending `sim-outorder` with Wattch [3] and then HotSpot 2.0 [11].

A main difficulty with SimpleScalar that had not been encountered with PTCMP was the abundance of global variables. Unlike PTCMP, SimpleScalar shares its variables across several independently compiled source files. All global and local static variables, including those in subcomponent libraries, had to be manually located and set to use TLS if appropriate. Although we cannot simulate multiple cores on hosts that do not support TLS, such as Cygwin, we are at least able to retain portability for single-core simulation support. This is done simply by using a global build option `-D__thread=` to disable TLS.

The SimpleScalar PowerPC package, containing our parallel `sim-outorder` and its associated tools, is freely available to download for noncommercial purposes using the link provided in Section VI. It has been tested for compatibility under Fedora Core 4 and gcc 4.0.

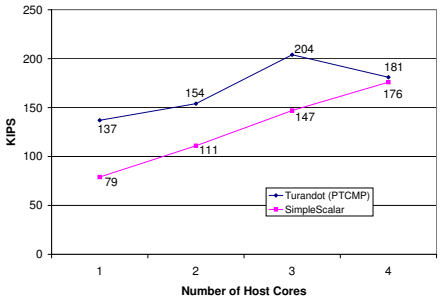

Fig. 2. Simulator performance when varying the number of available cores on a 4-way Opteron server.

IV. Speedup Results

We test an 8-program workload formed from the following SPEC CPU2000 benchmarks: `applu`, `bzip2`, `gap`, `gzip`, `mesa`, `mgrid`, `swim`, and `wupwise`. Although PTCMP supports parallel benchmarks, our parallel SimpleScalar does not, so we have opted to examine a multiprogrammed workload. To ensure matching instruction stream inputs to both simulators, we actually used SimpleScalar to generate the tt6e-format traces for Turandot. The workload is run such that at least 50 million instructions have completed on each core, totaling about 800 million instructions due to the mix of programs with different IPC. We simulate this on a dual-processor Opteron server where each processor has two 1.8 GHz cores, for a total of four virtual processors. The performance results of both simulators measured in thousands of instructions per second (KIPS) are shown in Figure 2. Because our design assigns one pthread per simulated core, these simulations each spawn 8 pthreads.

We do not compare in detail the absolute speed of our simulators to the few prior works on parallel multicore simulation because we are using different models and testing on different host platforms. For example, both our simulators are significantly faster than cycle-accurate multicore simulation with LSE [17], reasonable because LSE provides more detailed microarchitectural modeling [21].

PTCMP is the faster simulator in all cases because of Turandot's extensive use of predecoded information [14] and because the overhead of functional modeling is avoided in a trace-driven framework. However, because this simulator is trace-driven it becomes I/O-bound when simulating a moderately large number of input programs. Thus, we are able to achieve at most 1.5X speedup and performance decreases beyond three nodes because of congestion in the HyperTransport channels.

On the other hand, our parallel SimpleScalar is execution-driven. Thus, for this we see larger speedups linear with respect to the number of nodes. The parallel speedup achieved when utilizing all four cores is 2.2X.

Although the results in Figure 2 are only for a single 4-way system, we have observed parallel speedup on HyperThreaded™ processors and expect greater speedups on future many-core platforms. Our multithreaded simulators also may run on uniprocessor hosts at baseline performance. Thus, when the parallelism is not satisfactory, high-throughput computing can be achieved by running several multicore simulation jobs each using only one core. This way, if individual simulation turn-around time is not the highest priority, researchers can still benefit from quickly developing a robust

multicore simulator. Like most sequential simulators, the baseline single-node performance when using our method slows down only linearly with respect to the number of simulated cores or number of instructions executed. Furthermore, even on uniprocessor hosts, parallel simulators may still see minor speedup versus sequential simulators because multithreaded applications can better overlap computation with I/O.

V. Related Work

An alternative simulation method that served partly as the motivation for PTCMP was that of Zauber [13]. Although Zauber is a significant improvement in simulation speed compared to its predecessor Turandot CMP [12], it utilizes an approximate model for shared resources requiring a second simulation pass. This removes cycle-accuracy in single-pass simulation, which is necessary for experiments in inter-core dynamic adaptive policies.

Much past work has explored the broad problem of parallel discrete event simulation (PDES) [8], [16]. Various studies have examined synchronization protocols, conservative vs optimistic synchronization, hybrid techniques, load balancing, and many other issues. However, the consensus is that there is no one silver bullet and various techniques are best implemented on an application-specific basis. To consider some state-of-the-art techniques, we could apply optimistic synchronization to increase our simulators' parallel performance. However, our goal has not been to maximize parallelism at any cost. Rather, we have sought to achieve parallelism using relatively simple modifications to existing simulators.

Structural design methodologies can be modeled with PDES and are able to exploit parallelism with the help of an automating framework. This not only includes [17], but also RTL-based and gate-level simulation. Existing simulators for hardware description languages such as Verilog and VHDL have been written to exploit parallelism through various PDES techniques. However, these stages of processor design exist for the purpose of detailed simulation, and are inherently orders of magnitude slower than high-level simulators.

Among parallel architecture simulators, the Wisconsin Wind Tunnel (WWT) [18] and WWT II [15] use PDES and direct execution to simulate shared-memory target systems on a a variety of parallel hosts. Tango Lite is another direct execution simulator, originally written for uniprocessors but later parallelized for distributed systems [9]. However, direct execution precludes simulating architectures with an ISA different from the host or testing microarchitectural parameters such as issue width and pipeline depth. Chidester and George implement a parallel version of SimpleScalar using MPI [5]. Although they address many issues relevant for efficient distributed processing, our work shows similar speedups can be achieved with a simpler implementation on a shared memory host. Barr et al. develop a parallel implementation of Asim [7] to show how simulators built from the ground up with an object-oriented multicore model may allow additional cores to be simply instantiated then parallelized [1]. Their infrastructure requires a barrier synchronization method like the one we have presented, but no task of additional separation between core and shared variables. Thus, such practical parallelization on already-modular simulators can be thought of as a subset of our technique.

VI. Conclusions

We have proposed and demonstrated a methodology for parallel multicore simulator construction. With successful implementations on 2 different infrastructures, we believe this shows our techniques can readily be applied to other frameworks.

On a 4-node system we found speedups of 1.5X and 2.2X for PTCMP and SimpleScalar, respectively. This gain is only secondary to the benefits of code structure modularity and quick development from uniprocessor simulators without tedious recoding.

The parallel multicore edition of SimpleScalar PowerPC is available for download at http://parapet.ee.princeton.edu/ssppc/. It may be used as a foundation for studies in performance, power, and temperature of multicore PowerPC architectures, or may serve as a concrete reference point to understand and apply this methodology to other simulators.

VII. Acknowledgements

We thank David Brooks, Noel Eisley, Ben Lee, David Penry, Bob Safranek, Karu Sankaralingam, and the anonymous reviewers. We also thank IBM for the Turandot license agreement. This work is supported in part by grants from NSF, Intel, SRC, and the C2S2/GSRC joint microarchitecture thrust.

References

[1] K. Barr *et al.*, "Simulating a Chip Multiprocessor with a Symmetric Multiprocessor," in *Proc. of the Boston Area Architecture Workshop*, Jan. 2005.
[2] D. Brooks *et al.*, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro*, vol. 20, no. 6, pp. 26–44, Nov/Dec. 2000.
[3] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *Proc. of the 27th Intl. Symp. on Computer Architecture*, June 2000.
[4] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," Technical Report 1308, Computer Sciences Department, University of Wisconsin, July 1996.
[5] M. Chidester and A. George, "Parallel Simulation of Chip-Multiprocessor Architectures," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 3, pp. 176–200, July 2002.
[6] J. Donald and M. Martonosi, "Power Efficiency for Variation-Tolerant Multicore Processors," in *Proc. of the Intl. Symp. on Low Power Electronics and Design*, Oct. 2006.
[7] J. Emer *et al.*, "Asim: A Performance Model Framework," *IEEE Computer*, vol. 35, no. 2, pp. 68–76, Feb. 2002.
[8] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.
[9] S. Goldschmidt, "Simulation and Multiprocessors: Accuracy and Performance," Ph.D. dissertation, Stanford University, June 1993.
[10] T. Grotker *et al.*, *System Design with SystemC*. Kluwer Academic Publishers, 2002.
[11] W. Huang *et al.*, "Compact Thermal Modeling for Temperature-Aware Design," in *Proc. of the 41st Design Automation Conf.*, June 2004.
[12] Y. Li *et al.*, "Performance, Energy, and Thermal Considerations for SMT and CMP Architectures," in *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2005.
[13] ——, "CMP Design Space Exploration Subject to Physical Constraints," in *Proc. of the 12th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2006.
[14] M. Moudgill, J.-D. Wellman, and J. H. Moreno, "Environment for PowerPC Microarchitecture Exploration," *IEEE Micro*, vol. 19, no. 3, pp. 15–25, May/June 1999.
[15] S. Mukherjee *et al.*, "Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator," in *Proc. of the Workshop on Performance Analysis and Its Impact on Design*, June 1997.
[16] D. Nicol and R. Fujimoto, "Parallel Simulation Today," *Annals of Operations Research*, no. 53, pp. 249–285, 1994.
[17] D. Penry *et al.*, "Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multiprocessors," in *Proc. of the 12th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2006.
[18] S. K. Reinhardt *et al.*, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," in *Proc. of the SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1993.
[19] K. Sankaralingam *et al.*, "SimpleScalar Simulation of the PowerPC Instruction Set Architecture," Tech Report TR2000-04, University of Texas, 2001.
[20] "Thread-local Storage," http://en.wikipedia.org/wiki/Thread_Local_Storage, 2006.
[21] M. Vachharajani *et al.*, "Microarchitectural Exploration with Liberty," in *Proc. of the 35th Intl. Symp. on Microarchitecture*, Nov. 2002.