Fast Functional Simulation with Parallel Embra

Robert E. Lantz
Computer Systems Laboratory
Stanford University
rlantz@cs.stanford.edu

Abstract

A shift towards chip multiprocessor (CMP) designs has rekindled interest in full-system simulation of multiprocessors. Recent work has attempted to address the problem of linear (or worse) slowdown incurred during sequential simulation of a multiprocessor. Methods such as exploitation of the speed/detail trade-off, division (decimation) in simulation time, and statistical simulation share the common requirement of a fast functional simulator, which executes a workload at high speed with functional correctness but relaxed requirements for detailed microarchitectural or timing simulation. Although functional simulators execute at increased speed relative to detailed simulators, they also incur significant slowdown when simulating multiprocessor workloads, a problem which motivates this work.

This paper introduces Parallel Embra, a fast functional simulator for shared-memory multiprocessors which is part of the Parallel SimOS complete machine simulator. Parallel Embra takes an aggressive approach to parallel simulation; while it runs at user level and does not make use of the MMU hardware, it combines binary translation with loose timing constraints and relies on the underlying shared memory system for event ordering, time synchronization, and memory synchronization. Although this approach results in non-deterministic execution, it does not compromise functional correctness. Workload tests using the SPLASH-2 shared-memory applications show that Parallel Embra's approach to fast functional simulation scales up to 64-way parallel simulation without appreciable overhead compared to sequential simulation, and supports complete machine simulation of up to 1024-processor systems with practical performance.

1. Introduction

The benefits of *complete machine simulation* – simulation of a computer system including processor, memory system, and devices in enough detail to run a production operating system and applications – are well-established. These benefits include support for analysis, development and debugging of operating systems and commercial workloads running on a variety of real or hypothetical hardware designs and configurations, as well as improved accuracy compared to application-level simulators. Cain [6] presents evidence that "simulators that ignore system effects, no matter how precise [the simulation models may be], are likely to be so inaccurate as to be useless, even for CPU intensive benchmarks like SPECINT 2000."

The compelling advantages of complete machine simulation have led to the development of a number of simulator implementations. Publicly available complete machine simulation environments include *SimOS* [18] and its successors (such as *SimOS-Alpha*, *SimOS-PPC* and *PHARMsim* [6]), *SimICS* [13] and derived environments (including *TFsim* [12] and *SimFlex* [22]), *Sparc-sulima* [7], *Mambo* [5], *M5* [4] and *PTLsim* [26].

The primary challenge for complete machine simulators is the slowdown incurred during multiprocessor simulation. The principal effect of linear slowdown derives from having to do *N* times as much work when simulating an *N*-way multiprocessor as is required to simulate a uniprocessor.

To mitigate this linear (or worse) slowdown, a variety of techniques have been developed, including: exploiting the speed/detail trade-off, division in simulated time, statistical simulation, and parallel simulation. The first three of these methods require a *fast functional simulator* which quickly produces behaviorally correct execution (including program output, memory contents, etc.) while possibly sacrificing detailed accuracy in simulated time. In spite of increased base simulation speed, functional simulators are equally affected by the problem of multiprocessor simulation slowdown, a problem which motivates this work.

This paper introduces *Parallel Embra*, a fast functional simulator which provides supports the first three simulation acceleration methods mentioned above – speed/detail trade-off, division in simulated time, and statistical simulation – by applying the fourth: parallel simulation. Parallel Embra builds upon *Embra* [23], a binary translation based CPU simulator from the SimOS system, and extends it to support thread-parallel execution on a shared-memory multiprocessor. In addition to the core CPU simulator, most other simulator modules may also execute in parallel.

Parallel Embra simulates shared-memory workloads in parallel by dividing up simulated hardware at the granularity of a host multiprocessor system. It takes an aggressive, and unusual, approach to parallel simulation, accessing hardware shared memory (not including the MMU) directly, and avoiding cycle-based synchronization seen in other simulators.

This work makes several contributions, including:

- Describing a straightforward and practical method for parallel, functional complete machine simulation
- Demonstrating that this method allows functional complete machine simulation to scale up to 64-way parallel simulation of machines of up to 1024 processors, a level beyond recently reported results for complete machine simulators
- Showing how Parallel Embra falls into an overall spectrum of speed vs. detail/flexibility of parallel complete machine simulation, complementing other approaches.

Parallel Embra supports several methods of full-system multiprocessor simulator acceleration, and provides evidence that complete machine simulation can be a highly scalable and attractive method for design and analysis of a wide range of multiprocessor systems, from multicore and manycore designs to multiprocessor clusters and large networks of multicore PCs.

2. The Need for Fast Functional Simulation

Three important methods of accelerating complete machine multiprocessor simulation include exploiting the speed/detail trade-off, division ("decimation") of simulation time, and statistical simulation. Each of these methods depends on the availability of a fast functional simulator, which either generates checkpoints or "fast forwards" a simulation before or after detailed simulation. Parallel Embra is designed to support these methods, and also provides additional benefits such as interactive usability, instrumentation capabilities, and extremely rough performance estimation.

Most complete machine simulators (including nearly every core simulator mentioned above) take advantage of the *speed/detail trade-off* intrinsic in simulation: i.e. that more detailed simulation methods tend to be slower, while faster methods tend to be less accurate. By providing multiple modes of simulation, along with the ability to switch between them statically or dynamically, simulation systems can expose the speed/detail trade-off and allow users to select the level of speed and detail most appropriate to the task at hand. Most importantly, less interesting portions of a workload, such as booting the operating system, loading an application, and initializing data, may be executed with a fast functional simulator. While Embra, for example, uses a simplified timing model (assuming fixed instruction and memory latencies), it achieves good performance – approximately 10x slower than hardware execution in the current implementation[1]. Parallel Embra's goal is to achieve multiprocessor simulation performance that comparable to that of Embra or other binary translation emulators on a uniprocessor.

The approach of dividing simulation time, known as *Decimation in Simulation Time (DiST)* [11] parallelizes a sequential simulation by dividing its time axis. This is achieved by taking periodic checkpoints and then using multiple detailed simulator instances to simulate slightly overlapping time intervals in parallel until some degree of statistical convergence is achieved. This requires an initial simulation run to generate the checkpoints; a fast functional simulator such as Parallel Embra may be used to quickly generate the checkpoints, which provide starting points for parallel detailed simulation of each execution segment.

Statistical simulation samples a statistically significant series of execution time regions in detail and then generalizes the results using a statistical argument. For example, the *SimFlex* [22] complete machine simulation environment applies the *SMARTS* [25] methodology for selecting a subset of execution time to execute using a detailed simulator, while a fast functional simulator is required to handle execution between detailed simulation intervals. SimFlex also combines statistical simulation with the decimation in simulation time approach, requiring multiple checkpoints generated by a fast functional simulator.

Besides supporting several methods for multiprocessor simulator acceleration, fast functional simulators provide a variety of additional benefits, such as:

- supporting interactive/on-line use (e.g. typing, editing, connecting to a real network)
- providing a flexible platform for OS development
- allowing OS and software development for hardware that is not available
- allowing detailed instrumentation of and introspection into OS and user applications in a way that may not be supported by the hardware and software environment
- measuring event counts and other non-time-critical statistics
- estimating order-of-magnitude performance trends

Several of these additional benefits have become clearer in practice through use of Parallel Embra for OS development and in self-hosting the simulator, i.e. running it recursively to analyze and debug itself; particularly useful supplemental benefits, beside the core use of workload positioning for detailed simulation, have included instrumentation capabilities and event counting, such as counting TLB misses and other traps.

3. Design and Implementation

Parallel Embra's architecture and code (Figure 1) are based on Embra, an execution-driven, binary translation CPU simulator that dynamically generates code to simulate a single-cycle instruction and memory system. It supports address translation, exceptions and other events, access to simulated hardware, and basic statistical measurement, as

[1] This is somewhat larger than the 4-9x slowdown reported in the original Embra paper; while this work uses different test applications, some overhead was added when the translator was modified for retargetable code generation, and when 64-bit support was added, making use of a direct address translation array impractical. Parallel infrastructure itself added negligible overhead to serial simulation.

well as dynamic instrumentation via a scripting subsystem. Translated basic blocks are stored a Translation Cache (TC), interrupts and other events are queued in an event queue, and any features which are not implemented directly in translated code are handled through external callouts. Callouts are expensive compared to translated code, as context state must be saved and restored, and an entire C routine (or many routines) can be invoked to provide the callout behavior.

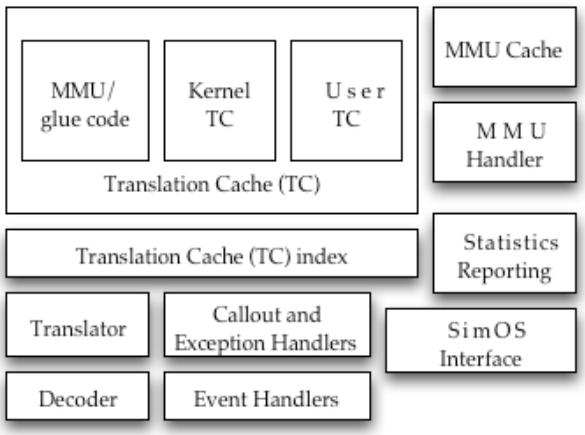

Figure 1: Parallel Embra Architecture. *Nearly every module of Parallel Embra – including callouts into the rest of the complete machine simulator – can operate in parallel, with the exception of statistics logging (which is serialized and written to a single file) and non-reentrant subsystems, including device models and (currently) the Tcl scripting subsystem. Components such as the Translation Cache and Index are replicated for local memory access, and to avoid contention.*

Primary issues for parallel simulators include workload division and resource multiplexing, and correct synchronization of parallel access to simulator data structures. Parallel Embra's solutions to these problems are described in sections 3.1 and 3.2, respectively. Unlike other functional simulators, Parallel Embra relaxes constraints for cycle synchronization across virtual processors, an approach described in section 3.3.

Like sequential Embra, Parallel Embra accesses shared memory directly after address translation, but in parallel rather than sequentially. While this is avoids the overhead of maintaining serialized memory access, it introduces non-determinism, as well as correctness issues addressed in section 3.4.

3.1 Parallel Organization

Parallel Embra divides a simulated machine into pieces which the simulator executes in parallel, at the granularity of the underlying hardware. For example, a 16-processor simulation with 2 GB of memory could be divided across 4 real hardware nodes by assigning 4 simulated processors and 512 MB of simulated memory to each node. Practically, this is achieved by creating 4 simulator threads, each of which allocates and places 512MB of simulated memory (as well as associated simulator data structures for that virtual processor) on the hardware node on which it is running. Careful attention to memory placement reduces slowdowns that might otherwise occur due to NUMA effects. After a thread allocates its memory, it begins simulating its assigned processors in round-robin fashion, with a dynamically configurable time slice.

This parallel organization, where the simulator resource allocation mirrors the simulated hardware organization, enhances TLB, cache and memory affinity, since a virtual processor will be rescheduled on the same physical processor, and since cache and local memory references in the simulated hardware are more likely to hit in cache or local memory on the simulator host. Additionally, simulator data structures, such as register arrays, address translation or other caches, etc., are more likely to be found in cache or local memory.

3.2 Simulator Synchronization

Parallel Embra's internal concurrency design resembles that of multiprocessor operating systems: fine-grained locking is used to allow concurrent access to any simulator subsystems which can be accessed concurrently. In the original Embra design, parallel execution was limited to translated code, and a single global lock serialized callouts into simulator code to handle traps, device access, etc.; this seriously limited parallelism, particularly in trap handling and access to the interrupt/clock/memory controller chip, as well as concurrent I/O and device access across multiple threads. In Parallel Embra, each of these subsystems may be accessed in parallel. Two other important modules of Parallel Embra, the translation cache and event queue, are replicated to allow parallel access without contention.

Maintaining consistency of translated code is a key problem for binary-translation emulators, and is exacerbated by parallelism. To maintain TC coherency, Parallel Embra detects writes to code pages (which are not mapped writable in the TLB hash) and invalidates the (replicated) TC. Although a partial TC flush mechanism was implemented, it did not improve performance compared to parallel, global TC flushes for the test workloads, and was removed to reduce complexity.

Global synchronization – for example halting threads for a checkpoint or for a TC flush after a code write – is handled by raising a barrier flag, which causes threads to wait at a barrier at the end of their next timeslice.

For cases where locking proved too costly, Parallel Embra uses non-blocking synchronization. Non-blocking synchronization of the global time value can occur either via atomic 64-bit writes to its value (in 64-bit mode), or atomic 32-bit writes to a generation counter (in 32-bit mode) to guarantee stable reads. Parallel Embra also supports address translation hash invalidation via an atomic pointer write; this allows one thread to dynamically instrument another thread (by invalidating a memory

translation, causing a trap on the next access), and could also be used to support parallel cache simulation.

3.3 Virtual CPU Synchronization

Parallel Embra takes an aggressive approach to cycle synchronization across simulated CPUs. Parallel Embra includes a barrier-based cycle synchronization feature, but tests revealed that regardless of the synchronization interval – anywhere from one basic block to millions of cycles – any cycle synchronization reduced performance, typically by a factor of two or more. This problem occurs due to unavoidable variations in simulation speed across simulator threads. In the case where a single thread simulates a single virtual processor, load imbalance across threads is unavoidable with the Parallel Embra design, as it cannot subdivide execution within a single virtual processor. As a result, for maximum performance Parallel Embra ignores cycle synchronization across processors, relying on the memory system for event ordering.

To provide a notion of global time, required by the operating system, Parallel Embra uses the cycle counter of processor 0. As noted above, this counter is guaranteed to be updated and read monotonically.

Simulated locking and synchronization events, such as the MIPS ll/sc (load locked/store conditional) instructions, are handled using equivalent hardware instructions, once again taking advantage of the memory system.

As noted above, all other synchronization within the simulator also happens using either blocking or non-blocking synchronization using the underlying shared memory hardware.

3.4 Correctness Argument

While it is clear that Parallel Embra's loosely-synchronized design compromises determinism, it is not immediately obvious that it does not compromise functional correctness as well. We argue that although Parallel Embra may expose timing dependencies, such as timing-dependent algorithm effects and race conditions, it does not compromise overall memory consistency. This is important, since it is required to ensure correct results from workloads (including the OS kernel itself as well as parallel applications) that synchronize themselves using locks, barriers, and other memory system based methods.

To test the functional correctness of the Parallel Embra implementation itself, application output (including numerical results) was compared with the output from serial simulation as well as hardware execution.

3.4.1 Timing dependencies

By allowing direct access to shared memory, Parallel Embra creates a dependency between simulated time and simulation time. This can result in timing-dependent algorithm effects, such as a different overall ordering of computation than would be seen on actual hardware. For example, since events are delivered according to a local cycle timer, they may occur at different global times on the simulator than would be observed on hardware. This effect was observed in the Irix timeout interrupt handler, where timeout interrupts could be delivered (and reposted) before their deadline in global time. Additionally, race conditions that do not cause incorrect results in a hardware environment can be exposed in the simulated environment. Fortunately such effects were not observed in the test workloads, since both the Irix operating system and the SPLASH-2 benchmarks attempt to use explicit synchronization, such as locks and barriers, to avoid race conditions. Correct synchronization behavior relies on the preservation of memory consistency, discussed in the next section.

3.4.2 Memory consistency

To explain why memory consistency is preserved, it can be shown that memory events cannot be seen to occur in a different order relative to global time across simulated processors:

1. The underlying shared memory system guarantees sequential consistency, so hardware memory events will not be reordered across processors.
2. Simulated memory accesses are implemented as accesses to the underlying shared memory system.
3. Synchronization events are also ordered in the underlying shared memory system.
4. Other simulator events (e.g. interrupts, device accesses) are serialized via locking in the memory system.
5. Clock updates and reads are serialized in the memory system.

Given the above five conditions, it is generally not possible for one processor to see an ordering of event A followed by event B, and another processor to see an ordering of event B followed by event A. In short, all simulator events, including simulated memory accesses, device accesses, and clock accesses, are implemented by hardware memory events which are sequentially consistent.

As in real hardware, there is a possibility of inconsistent address translations during a page table or TLB update. However, locking the simulated TLB and invalidating the TLB hash entry guarantees that the next access to an invalidated page cannot proceed until the update has completed.

3.4.3 Implementation testing

To verify the functional correctness of the actual Parallel Embra *implementation*, parallel simulator results (including numeric results as well as other data such as the Raytrace application output shown in Figure 2) were compared with the results of both serial simulation and hardware execution. Consistent results provide further confidence as to the behavioral correctness of the approach as well as its implementation.

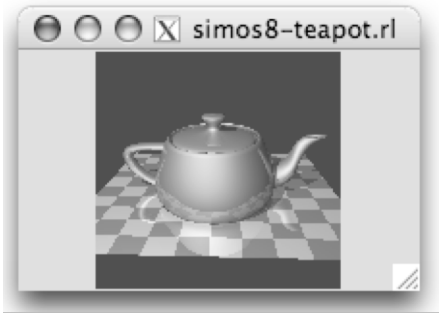

Figure 2: Parallel Embra output for Raytrace application. *The correctness of the Parallel Embra implementation was checked by comparing the output, numerical and otherwise, of applications simulated in parallel against the output from sequential simulation and hardware execution.*

4. Performance Evaluation

This section presents performance results for simulating several benchmarks from the *SPLASH-2* [24] and SPLASH [19] suites of parallel applications, as well as a parallel make benchmark, pmake, based on the compile phase of the Modified Andrew Benchmark (distributed with SimOS). The results are from up to 64-way parallel configurations of Parallel Embra, running on the Stanford FLASH machine [10], a 64-way multiprocessor based on the SGI Origin platform, as well as a stock SGI Origin 2000 (195 MHz R10000, 16 processors). The Irix 6.5 operating system is executed by both real and simulated hardware environments.

While the FLASH machine's specifications are decidedly vintage (such as 225 MHz MIPS R10000 processors and 14 GB usable real memory), this environment provided the compelling advantage of being able to run Parallel Embra as well as the simulated workload directly on the machine (and operating system) it was simulating, allowing measurement of simulator overhead (slowdown) irrespective of variation in underlying hardware.[2] The FLASH and Origin are NUMA architectures, typical of a cache coherent shared-memory multiprocessor supporting more than 8 processors or cores.

Considering self-relative slowdown, the results indicate that Parallel Embra can simulate a multiprocessor with overhead comparable to sequential Embra simulating a uniprocessor. Self-relative speedup results show notable performance gains for up to 64-way parallel simulation, although overall speedup was generally limited by serial initialization phases of the applications. Large system tests demonstrate practical functional simulation of large systems of up to 1024 processors – more than an order of magnitude larger than what could be simulated practically with sequential Embra.

4.1 Hardware-relative Slowdown

Simulator *slowdown*, defined as the ratio of simulator execution time to hardware execution time, measures how much longer a workload takes to run on the simulator, compared to execution time on actual hardware. To eliminate effects of faster simulation hardware (e.g. simulating a slow target architecture using a much faster machine), we measure *hardware self-relative slowdown,* i.e. the slowdown resulting from simulating the hardware (in our case the FLASH multiprocessor) that the simulator is running on. Hardware self-relative slowdown provides a measure of the overhead of simulation itself, and can be compared between serial and parallel simulation modes. Figure 3 summarizes self-relative slowdown for Parallel Embra running the test application suite.

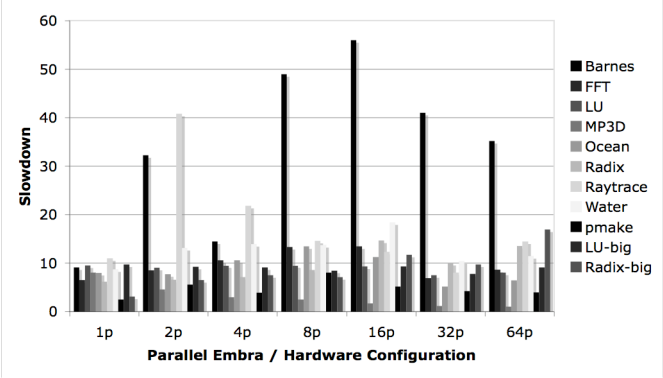

Figure 3: Hardware Self-relative Slowdown. *Since Parallel Embra runs on the system it simulates, we were able to measure hardware self-relative slowdown for our benchmarks for 1 to 64 processors. LU-big and Radix-big are the LU and Radix programs run with larger data sets.*

With the exception of Barnes and Raytrace-2, hardware self-relative slowdown does not increase significantly from one to 64 processors, which is to say that Parallel Embra runs parallel simulations with comparable slowdown to serial simulations, around 10x, given enough parallelism in the underlying hardware.

To understand the results from Figure 3 – including anomalous results – we examined the behavior in closer detail using a virtual profiling and visualization subsystem that we developed for Parallel Embra. The profile results revealed that overall performance may be reduced by three main workload effects: system imbalance effects, excessive contention, and startup/serial initialization effects.

4.1.1 System Imbalance Effects

Profiling revealed that the Barnes benchmark showed particularly poor self-relative slowdown and reduced speedup even in its computation phase, the result of *system imbalance effects* – that is, a change in the relative

[2] SimOS supports MIPS, Alpha, PowerPC and x86 CPU simulators, but the MIPS simulators included the widest variety, from binary translation to microarchitecture, and ran on the largest locally available multiprocessor. The Parallel Embra approach is equally applicable to other SimOS CPU simulators such as *Delta* for the Alpha architecture. Parallel Embra is also capable of booting MIPS Linux, though no port existed for the FLASH hardware.

execution speed of threads as we move from hardware execution to simulation with Parallel Embra.

Running on Parallel Embra, computation for the Barnes benchmark is poorly load-balanced, and much time is spent in TLB miss handlers. The Barnes benchmark appears to show an extreme case of system imbalance effects, where excessive TLB misses on certain threads slow down simulated execution much more than hardware execution, resulting in load imbalances on the simulator that are not present when executing natively on the hardware. Since Parallel Embra cannot parallelize beyond the intrinsic thread parallelism of the application, this is an unavoidable limitation of our overall approach.

4.1.2 Contention Effects

While system imbalance effects can reduce performance under parallel simulation, excessive contention effects in a benchmark can sometimes make simulator performance look better than it actually is. Profile analysis of the MP3D benchmark revealed excessive contention and descheduling, explaining its remarkably good self-relative slowdown, but revealing that it is actually achieving poor parallel performance.[3]

4.1.3 Serial Phase Effects

Even applications with good performance in simulation do not necessarily benefit as much as they could from parallel simulation. For example, LU is the star performer of the benchmark suite: in contrast to the previous two applications, its profile showed no load imbalance and minimal contention and descheduling effects. Nonetheless, LU suffered from the common problem of serialization in its startup phase, due to sequential thread creation calls and serial initialization in the master thread. Parallel performance in such phases is wasted since most CPUs are (rapidly) executing the idle loop. All of the benchmarks exhibit significant serial startup and initialization effects that reduce overall performance. (Effects on speedup, as per Amdahl's law, are illustrated in the next section.)

4.1.4 Summary

Self-relative slowdown results show that Parallel Embra effectively mitigates the multiplicative slowdown caused by multiprocessor simulation, allowing simulation with comparable (10x) slowdown to that of simulating a uniprocessor. Examining three of the test applications in detail revealed important workload effects for parallel simulation: system imbalance effects, which can lead to load imbalances in the simulator that do not exist in hardware; excessive contention (and poor benchmark scalability), which can make a simulator's performance look better than it actually is; and serialization effects, which limit the overall benefit from parallel execution. The next section examines parallel performance in more detail, irrespective of serial initialization effects.

[3] Although it was not included in the SPLASH suite, and we did not test it on Parallel Embra, a subsequent version of the MP3D application, *PSIM4* [14], developed by the original author, achieved excellent performance on both message-passing and shared-memory machines.

4.2 Self-relative Speedup

Although simulation time is the most meaningful measurement of simulator performance, parallel program developers frequently report self-relative *speedup*, or the ratio of serial to parallel execution time for the same program and workload, as a measure of parallel performance or efficiency. In addition to overall speedup including startup and initialization phases, Figures 4 and 5 show speedup for parallel computation regions, to more accurately reflect the parallel performance of the simulator independent of serial startup effects.

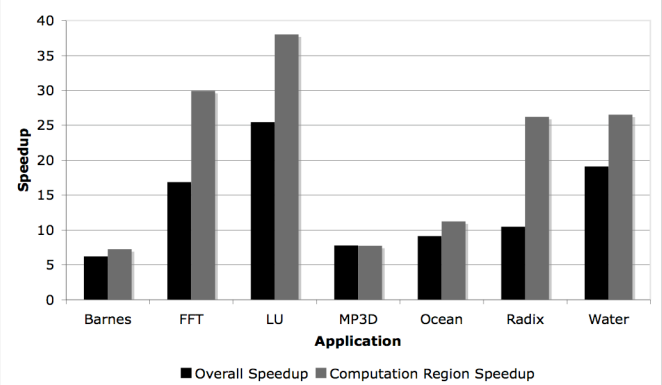

Figure 4: Speedup: Overall vs. Computation (32p)

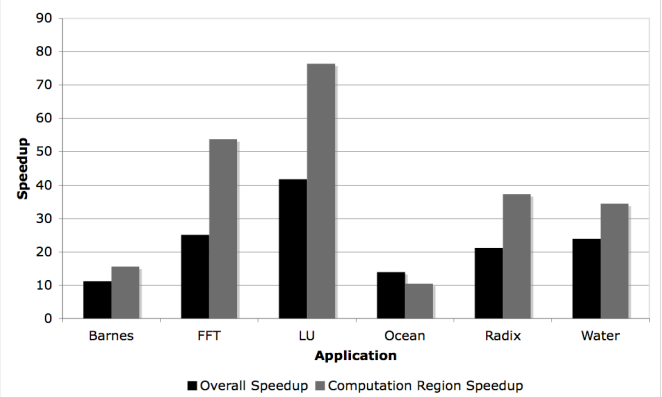

Figure 5: Speedup: Overall vs. Computation (64p). *Parallel Embra shows improved self-relative speedup from 32 to 64 processors for every application except Ocean. Speedup is limited in some cases, particularly for Barnes and Ocean, by system imbalance effects. Additionally, MP3D and Ocean are limited by frequent synchronization.*

As shown in Figures 4 and 5. FFT, LU, Radix and Water achieve significant parallel speedup in their computation regions (including superlinear speedup for LU), but Barnes, MP3D and Ocean suffer from poorer parallel performance. In the case of Barnes and MP3D, speedup was reduced by system imbalance effects, while MP3D and Ocean are limited by frequent synchronization, which can be more costly at higher degrees of parallelism.

Figure 6: 1024-processor System Boot. *Exploring the limits of resource overcommitment (1024 virtual CPUs on 8 real CPUs) for interactive use, our Tcl/Tk-based console tool displays a simulated cluster of 16 64-processor machines running 8-way parallel. Varying programs and activities appear in different panes of the virtual console tool.*

4.3 Large System Results

To evaluate its scalability limits, we configured Parallel Embra to simulate a 1024-processor machine. Since our port of the Irix operating system (to both real and simulated FLASH hardware) did not reliably support more than 128 processors, the simulated machine was split into 128- or 64-processor partitions, connected to a global shared memory. For test applications in this simulated configuration, Parallel Embra ran on 32 and 64 processors. A highly overcommitted configuration of 1024 simulated processors on 8 real processors was also found to be capable of booting the Irix OS and executing Unix commands interactively, if slowly.

4.3.1 1024-processor Tests

Figure 7 shows results from a 1024-processor (1024p) simulation, divided into 8 128-processor partitions, running the Radix and LU benchmarks on 32- and 64-way parallel simulations, respectively, on the FLASH machine.[4]

Although it was not feasible to get a hardware or serially simulated 1024p result on the test machine (for the latter, besides the excessive simulation time, the FLASH hardware would most likely have crashed, hung, or been rebooted), Figure 7 compares real time/simulated time slowdown for 32- and 64-way parallel 1024p simulations with estimated slowdown for serial 1024p simulations, the latter conservatively modeled as taking 8x as long as a serial 128-processor simulation with the same application and work per processor. Overall simulation time is reduced by a factor of 13-21. For a one minute (simulated time) benchmark, this would reduce simulation time from more than a week to approximately 8 hours.

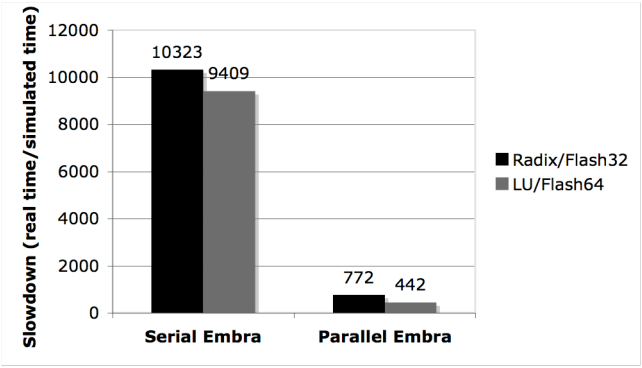

Figure 7: Slowdown for 1024-processor simulations. *This chart compares estimated slowdown for a 1024-processor serial simulation to slowdown for Parallel Embra running the Radix and LU applications, 32- and 64-way parallel, respectively.*

4.3.2 Scalability Limits

Simulating a 1024-processor system on an 8-processor system represents the upper limit of scalability for Parallel Embra, at which point the effects of linear slowdown, cache and TLB exhaustion become excessive. While extreme overcommitment is possible, simulating a machine of 16 to 64 times larger than the underlying real hardware yields more usable performance.

Running 8-way parallel on a stock SGI Origin 2000, this simulation (Figure 6, top of page) booted up in approximately 90 minutes, and although the simulated machines were slow to respond, they demonstrate the possibility of interactive use of a large machine simulation.

Simulators require enough storage to encapsulate the state of a simulated machine, and this can further limit scalability. For the 1024p tests, the largest state component by far was the target machine's memory. The test hardware's 14 GB (16 for the Irix boot test) of real memory, and about 32 GB of virtual memory limited the 1024-processor simulated machine to 32 MB of memory

[4] This configuration tended to put the experimental FLASH hardware into an inconsistent state, limiting the ability to perform large system experiments. While checkpoints can provide some resiliency, large checkpoints turned out to be unwieldy (e.g. exceeding local storage and requiring excessive time to save and restore many gigabytes of data.)

per processor, a much smaller amount of memory than would be included in a real machine, and a workload (in this case, applications with fairly low memory requirements) which could fit on the scaled-down machine.

4.4 Performance Discussion

Self-relative slowdown results show that Parallel Embra can simulate a parallel workload with comparable overhead to serial Embra simulating a serial workload, and that parallel overhead does not increase greatly for up to 64-way parallel simulation, although parallel performance can be limited by system imbalance effects, where variance in relative operation cost create unavoidable (for N simulated processors on M real processors, where $N \leq M$) load imbalances under parallel simulation. Although results were not included here, experience indicates that multiplexing multiple simulated processors on a single simulator thread can result in better load balance.

Other test configurations supported up to 128 virtual processors per single real processor, but our experiences indicate that 16:1 to 64:1 are more practical ratios of virtual to real processor overcommitment for large system simulation. Practical limits such as the amount of memory (real and virtual) available to Parallel Embra also place an upper bound on the size of the system which may be simulated, and the degree of memory overcommitment (a factor of two in the 1024-processor tests) will also affect performance.

5. Related Work

Parallel Embra addresses the slowdown inherent in complete machine simulation of multiprocessors by providing a fast, parallel functional simulator that supports several acceleration techniques: speed vs. detail trade-off, decimation in simulation time, and statistical simulation. This section compares Parallel Embra with related work in parallel complete machine simulation, as well as alternate approaches such as direct execution, workload reduction, and hardware-based simulation.

5.1 Parallel Simulation

There is a large body of work describing parallel system simulators, and a much smaller body of work describing parallel complete machine simulators – i.e. parallel simulators capable of booting an operating system. A milestone parallel simulator is the *Wisconsin Wind Tunnel* [17] a simulator which used the ECC bits on the CM-5 to support simulation of cache-coherent shared-memory architectures, and which provided repeatability using fast global synchronization (provided by the CM-5 hardware) and a time quantum smaller than the interconnect delay. *WWT-2* [15] extended this technique to simulation on modern multiprocessors and clusters. Although shared memory and cache were simulated by both WWT systems, operating system effects were not. Many other parallel simulators have been developed, most commonly application-level simulators that support a message-passing API. (Examples include *Compass* [3] a parallel, application-based simulator for MPI applications, and *BigSim* [27], a parallel application-level simulator for very large message-passing machines such as IBM's Blue Gene.)

The original Embra work included an experimental parallel mode which used process-based parallelism, barrier synchronization and non-deterministic execution. Parallel execution was limited to execution of translated code, serializing any traps or callouts across the entire simulation, and limiting performance. In contrast, the current Parallel Embra implementation uses thread-based parallelism and allows nearly all simulator operations to proceed in parallel.

Besides Parallel Embra, two recent efforts to parallelize complete machine software simulators were undertaken with Mambo [20] and Sparc-sulima[5] [16]. The Mambo work is closest to Parallel Embra, in that it aims to produce a fast functional simulator by extending a binary-translation based emulation mode ("turbo" mode); published results include a speedup of up to 3.8 for a 4-way parallel simulation. Published work does not discuss whether parallel Mambo preserves determinism by synchronizing event timing across threads, or whether turbo mode accesses shared memory directly. If the latter, it is close to the simulation core of Parallel Embra, although it takes a somewhat different approach of switching simulation context after every instruction, with instructions being fed to a thread via a time-sorted priority queue. In contrast, Embra and Parallel Embra translate entire basic blocks, which results in better performance at the expense of a larger minimum context switch granularity.

Sparc-sulima implements two parallel simulation modes: a deterministic "active backplane" mode which accelerates cycle-accurate simulation by synchronizing cycles across processors using the window method from the Wisconsin Wind Tunnel, and a non-deterministic "passive backplane" which supports cache simulation with approximate cycle synchronization that occurs via barriers after a fixed time quantum. Similar to Parallel Embra's TLB locking (and a partially implemented parallel cache mode), Sparc-sulima locks cache lines, using locks which are not synchronized in virtual time. Both approaches yielded an approximate 3x speedup for a 4-way parallel simulation.

Parallel Embra implements a different speed/detail trade-off than parallel modes of Mambo or Sparc-sulima, one intended to achieve greater speed at the expense of simulation detail. Unlike Mambo, it uses a higher minimum simulation granularity. Like Sparc-sulima, Parallel Embra introduces non-determinism; however its fastest mode of execution avoids cycle synchronization entirely, and also does not simulate caches.

[5] While Sparc-sulima was originally intended to support full OS booting and simulation, porting difficulties motivated a switch to an OS emulation layer to model system effects [7].

5.2 Direct Execution

Parallel Embra accesses memory directly, but translates simulated virtual addresses without use of the hardware MMU. In contrast, a direct execution approach makes use of the hardware MMU to accelerate address translation.

The original SimOS implementations supported direct execution, but this was abandoned due to the inconvenience of having to modify the underlying operating system to support simulator MMU usage, combined with the good performance of binary translation emulation. This situation has changed somewhat with the wide adoption and availability of virtual machine monitors: PTLsim, for example supports a direct execution mode based on the Xen hypervisor. A PTLsim layer within a Xen domain provides multiprocessor emulation, and a guest kernel directly executes a test workload. Special instructions allow the test workload to switch into a cycle-accurate simulation mode. Parallel direct execution resembles Parallel Embra in that it produces non-deterministic results, but it bears a much closer resemblance to hardware execution, particularly in the non-multiplexed case.

In contrast to the direct execution approach, binary-translation based fast functional simulation provides increased flexibility such as detailed instrumentation, custom statistical measurement, the ability to modify instruction behavior, and the ability to run on a different host architecture; thus it represents a different and complementary point in the speed vs. detail/flexibility trade-off spectrum.

5.3 Workload Reduction

Parallelism addresses simulation speed, but host memory size can limit the scalability of full system simulation, as noted in section 4.3.2. Efforts such as the *Wisconsin Commercial Workload Suite* (WCWS) [1] aim to produce representative samples of computation which can predict the behavior of larger machines but which have a smaller memory footprint, allowing them to fit in a smaller simulator's memory, as well as reduced computational demands. Such efforts are orthogonal and beneficial to any complete machine simulator. However, given enough memory, complete machine simulators such as Parallel Embra can run full-scale, unmodified commercial workloads.

5.4 Hardware-based Simulation

As FPGAs increase in power and drop in cost, they are increasingly being used to develop flexible cycle-accurate architectural simulators. Examples include *FAST* [8], *RAMP* [21], *ProtoFlex* [9] and many others. Such simulators provide compelling advantages of high performance and believability [2].

Nonetheless, software simulators still retain some critical advantages which are likely to preserve them for the foreseeable future, notably negligible manufacturing and distribution cost. Software simulators may also be easier to implement, extend, and configure than hardware-based simulators, and they can be developed and deployed on cheap and readily available PC hardware. Moreover, hardware approaches may still have to deal with analogous slowdown effects seen in software simulators: when simulating larger systems than the actual hardware, they must address similar problems of multiplexing and linear slowdown.

6. Conclusion

This work has presented a parallel, functional simulator, Parallel Embra, which supports complete machine simulation of shared-memory multiprocessors. Parallel Embra's fast functional simulation capabilities support several methods for improving the performance of multiprocessor simulation, including exploiting the speed/detail trade-off, decimation in simulation time, and statistical simulation. As an on-line, execution-driven simulator, Parallel Embra also provides useful capabilities such as interactive simulation and support for multiprocessor OS and software development.

Parallel Embra takes a different and complementary approach to that taken by recent parallel complete machine simulation efforts: it relies on the underlying shared memory system for synchronization, and runs without cycle synchronization across virtual processors (although a global monotonic time base is provided, as is a mechanism for barrier synchronization). As a result it executes workloads non-deterministically but preserves overall correctness, since memory events cannot be reordered across virtual processors.

This approach scales up to 64-way parallel execution without significant increases in simulation overhead. At the extreme level, Parallel Embra is able to accelerate simulations of up to 1024 processors. Parallel Embra shows high scalability compared to recently reported results in parallelizing complete machine simulators. This may be due in part to Parallel Embra's reduction in detail and synchronization requirements relative to those simulators (e.g. not simulating caches like Sparc-sulima, and having a higher minimum granularity than Mambo).

While Parallel Embra achieves good performance overall, workload tests reveal several important limitations of its approach; loss of determinism may be the most significant, but less obvious limitations include exposure of timing dependencies and race conditions, potentially unavoidable system imbalance effects caused by variation in execution speed across virtual processors, and contention effects due to true parallel execution. Additionally, the test applications exhibited significant serial startup regions which cannot be accelerated by Parallel Embra's sequential execution of individual threads.

Considering an overall spectrum of speed vs. detail/flexibility for execution-driven parallel complete machine simulation (Figure 8), Parallel Embra falls between direct parallel execution (such as in PTLsim on

Xen) and non-deterministic parallel cache simulation (as in Sparc-sulima).

↑ ***Faster Simulation Speed***

Parallel Direct Execution (e.g. PTLsim)
Parallel with Paging (e.g. Parallel Embra)
Parallel with Caches (e.g. Sparc-sulima passive)
Parallel Deterministic (e.g. Sparc-sulima active)
Parallel Microarchitecture (future work)

↓***Greater Simulation Detail/Flexibility***

Figure 8. Speed vs. Detail/Flexibility spectrum for parallel complete machine simulation. *Each of these methods may be further varied in speed and detail by changing pieces of the model, for example ignoring cache simulation in a deterministic model, or simulating a detailed interconnect model in a non-deterministic simulator***.**

Along with other recent developments, experiences with Parallel Embra suggest that complete machine simulation may scale well to medium-scale multiprocessor hosts, as well as large target systems of 1024 processors or more, systems which primarily have been modeled with application-level simulators. Such large, many-processor systems include multiprocessors, multicore and manycore architectures, multiprocessor clusters, and networks of multicore PCs. We look forward to complete machine simulators that span the entire breadth of the speed/detail trade-off, including flexible parallel modes as well as compatibility and integration with virtual machine and hardware-assisted simulation environments.

7. Acknowledgments

Thanks to Mendel Rosenblum for advising the author during this project, for comments on the paper, and for information about the early development of SimOS (section 5.2.) Alan Swithenbank was instrumental in keeping the Stanford FLASH multiprocessor alive long enough to run tests. Parallel SimOS and Parallel Embra build upon the work of the many developers of the SimOS simulator.

8. References

[1] Alameldeen, A. R., Martin, M. M., Mauer, C. J., Moore, K. E., Xu, M., Hill, M. D., Wood, D. A., and Sorin, D. J. 2003. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer* 36, 2 (Feb. 2003), 50-57.

[2] Asanovic, K. Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report. UCB/EECS-2006-183. University of California at Berkeley, Berkeley, CA, 2006.

[3] Bagrodia, R., Deeljman, E., Docy, S., and Phan, T. 1999. Performance Prediction of Large Parallel Applications Using Parallel Simulations. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, GA, May 04 - 06, 1999). PPoPP '99. 151-162.

[4] Binkert, N.L., Derslinki, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K., 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*. 26,4 (July-Aug. 2006), 52-60.

[5] Bohrer, P., Peterson, J., Elnozahy, M., Rajamony, R., Gheith, A., Rockhold, R., Lefurgy, C., Shafi, H., Nakra, T., Simpson, R., Speight, E., Sudeep, K., Van Hensbergen, E., and Zhang, L. 2004. Mambo: A Full System Simulator for the PowerPC Architecture. *SIGMETRICS Perf. Eval. Rev.* 31, 4 (Mar. 2004), 8-12.

[6] Cain, H., Lepak, K. Schwartz, B and Lipasti, M. 2002. Precise and Accurate Processor Simulation. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads* (Cambridge, MA, 2002). 13-22.

[7] Clarke, B. 2004. Solemn: Solaris Emulation Mode for Sparc Sulima. In *Proceedings of the 37th Annual Symposium on Simulation* (April 18 - 22, 2004). Annual Simulation Symposium. 64-71.

[8] Chiou, D., Sunwoo, D., Kim, J., Patil, N. A., Reinhart, W., Johnson, D. E., Keefe, J., and Angepat, H. 2007. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture* (December 01 - 05, 2007). MICRO-37. 249-261.

[9] Chung, E. S., Nurvitadhi, E., Hoe, J. C., Falsafi, B., and Mai, K. 2008. A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs. In *Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 24 - 26, 2008). FPGA '08. 77-86.

[10] Gibson, J., Kunz, R., Ofelt, D., Horowitz, M., Hennessy, J., and Heinrich, M. 2000. FLASH vs. (Simulated) FLASH: closing the simulation loop. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 49-58.

[11] Girbal, S., Mouchard, G., Cohen, A., and Temam, O. 2003. DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time. In *Proceedings of the 2003 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, June 11 - 14, 2003). 1-12.

[12] Mauer, C. J., Hill, M. D., and Wood, D. A. 2002. Full-System Timing-First Simulation. In *Proceedings of the 2002 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems* (Marina Del Rey, California, June 15 - 19, 2002). SIGMETRICS '02. 108-116.

[13] Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B. 2002. Simics: A Full System Simulation Platform. *IEEE Computer* 35,2 (Feb. 2002), 50-58.

[14] McDonald, J.D. 1991. Particle simulation in a multiprocessor environment. In *Proceedings of the 26th Thermophysics Conference* (Honolulu, HI, 1991). 1366-1369.

[15] Mukherjee, S. S., Reinhardt, S. K., Falsafi, B., Litzkow, M., Hill, M. D., Wood, D. A., Huss-Lederman, S., and Larus, J. R. 2000. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency* 8, 4 (Oct. 2000), 12-20.

[16] Over, A., Clarke, B., Strazdins, P. 2007. A Comparison of Two Approaches to Parallel Simulation of Multiprocessors. in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software, ISPASS 2007* (April, 2007). 12-22.

[17] Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. 1993. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.* 21, 1 (Jun. 1993), 48-60.

[18] Rosenblum, M., Herrod, S.A., Witchel, E., Gupta, A. 1995. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel & Distributed Technology: Systems & Applications* 39, 4 (Winter 1995) , 34-43.

[19] Singh, J.P., Weber, W.-D., Gupta, A. 1992. SPLASH: Stanford Parallel Applications For Shared Memory. *ACM SIGARCH Computer Architecture News* 20, 1 (March 1992), 5-44.

[20] Wang, K., Zhang, Y., Wang, H., and Shen, X. 2008. Parallelization of IBM Mambo System Simulator in Functional Modes. *SIGOPS Oper. Syst. Rev.* 42, 1 (Jan. 2008), 71-76.

[21] Wawrzynek, J., Oskin, M., Kozyrakis, C., Chiou, D., Patterson, D., Lui, S., Hoe, J. C., Asanovic, K. 2006. *RAMP: A Research Accelerator for Multiple Processors.* Technical Report. UCB/EECS-2006-158. University of California at Berkeley, Berkeley, CA, 2006.

[22] Wenisch, T.F., Wunderlich, R.E., Ferdman, M., Ailamaki, A., Falsafi, B., Hoe, J.C. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (July-Aug. 2006), 18-31.

[23] Witchel, E. and Rosenblum, M. 1996. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems* (Philadelphia, PA, 1996). 68-79.

[24] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual international Symposium on Computer Architecture* (S. Margherita Ligure, Italy, June 22 - 24, 1995). 24-36.

[25] Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C. 2006. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. Model. Comput. Simul.* 16, 3 (Jul. 2006), 197-224.

[26] Yourst, MT. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2007* (San Jose, CA, 2007). 23-34.

[27] Zheng, G., Kakulapati, G., Kalé, L. 2004. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (Santa Fe, NM, 2004). 78-87.