

Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors

David A. Penry[†]

Graham Schelle[‡]

Daniel Fay[‡]

David I. August[†]

David Hodgdon[‡]

Dan Connors[‡]

Ryan Wells[†]

[†]Princeton University
Dept. of Computer Science
35 Olden St.
Princeton, NJ 08540

[‡]University of Colorado at Boulder
Dept. of Electrical and Computer Engineering
Engineering Center
Boulder, CO 80309

Abstract

Simulation is an important means of evaluating new microarchitectures. Current trends toward chip multi-processors (CMPs) try the ability of designers to develop efficient simulators. CMP simulation speed can be improved by exploiting parallelism in the CMP simulation model. This may be done by either running the simulation on multiple processors or by integrating multiple processors into the simulation to replace simulated processors. Doing so usually requires tedious manual parallelization or re-design to encapsulate processors.

Both problems can be avoided by generating the simulator from a concurrent, structural model of the CMP. Such a model not only resembles hardware, making it easy to understand and use, but also provides sufficient information to automatically parallelize the simulator without requiring manual model changes. Furthermore, individual components of the model such as processors may be replaced with equivalent hardware without requiring repartitioning.

This paper presents techniques to perform automated simulator parallelization and hardware integration for CMP structural models. We show that automated parallelization can achieve an 7.60 speedup for a 16-processor CMP model on a conventional 4-processor shared-memory multiprocessor. We demonstrate the power of hardware integration by integrating eight hardware PowerPC cores into a CMP model, achieving a speedup of up to 5.82.

1. Introduction

Microarchitects develop and use simulators to evaluate their ideas. Developing a simulator requires careful tradeoffs among speed, development time, and accuracy. The on-

going trend towards tying multiple processor cores on a single die into a chip multi-processor (CMP) makes determining these tradeoffs even more challenging. As the number of cores and the complexity of their interconnect increases, simulators become larger, more complex, and slower.

One means to accelerate CMP simulation is to exploit the parallelism inherent in the CMP. This can be done in two ways. First, the simulator can be parallelized to run on multiple host processors. Doing so is often difficult, requiring manual parallelization of simulator code not originally written to be parallelized. Second, real processors can be used to replace simulated processors in the CMP simulator. In this case, memory operations from the processors are simulated, while instructions are directly executed. This requires manual re-design of the simulator code to fully encapsulate the processors.

In this paper we show that the difficulties involved in manual parallelization and processor integration can be alleviated if the simulator is not written by a simulator designer, but instead is generated from a concurrent, structural model of the CMP, such as those used in [19] or [8]. Model structure and parallelism may be exploited to achieve automated parallelization and simple replacement of components of the model with actual processors. This ability to exploit parallelism comes in addition to the benefits of reduced development time and reduced modeling error previously shown [19, 16] for such models.

A microarchitectural model in a concurrent structural modeling framework consists of a collection of components. These components execute concurrently and are connected together by signals, which form the principal means of inter-component communication. The user instantiates, parameterizes, and connects components to create the model. The framework then combines the instantiation and connection information with the components' behavioral code to generate a simulator for the model. The con-

current structural modeling framework used and modified in this work is the Liberty Simulation Environment (LSE)[19]. LSE provides explicit support for structural modeling, an extensible library of components, and language features designed to make parameterization, instantiation, and library creation easier. One unusual feature of LSE is that it treats all components as “black boxes”: it attempts no analysis of how a component operates, allowing the possibility of the component using hardware to calculate its outputs.

A structural model is a natural candidate for parallelization, as the components in the model are already designed to execute concurrently. This execution must be carefully scheduled to achieve speed improvement. The scheduling problem is similar to that of instruction scheduling, but cache effects and the coordination of accesses to shared state are important additional factors. We describe modifications to parallel scheduling algorithms to deal with these factors. We evaluate automated parallelization on a family of chip multi-processor models and find that automated simulator parallelization achieves a modest speedup of 2.27 with 4 threads for a 4-way CMP model, but a much larger speedup of 7.60 with 4 threads for a 16-way CMP.

The integration of hardware components is made feasible because the structural nature of the system allows portions of the model to be replaced without affecting the remainder of the model. We describe the hardware and software support necessary to integrate an arbitrary piece of hardware into a model. In addition, hardware component integration increases accuracy, obviates the need to model available components, and enables an incremental architecture-to-implementation design flow. We integrate eight PowerPC 405 cores realized in Field-Programmable Gate Arrays (FPGAs) into a CMP model described in LSE, achieving a speedup of 5.82 with perfect caches and 1.31 with a detailed simulated memory hierarchy and interconnect.

Section 2 explains how automated simulator parallelization is carried out, while Section 3 evaluates the parallelization. Section 4 explains integration of hardware components, and Section 5 presents the results of this integration. Section 6 concludes.

2. Automated simulator parallelization

Simulator parallelization is one way to take advantage of parallelism in a CMP simulator. Parallelization, whether manual or automated, requires that the simulator be partitioned into tasks to be run concurrently. It then requires that these tasks be either scheduled or synchronized such that tasks wait for their inputs to become available and that tasks which could simultaneously access shared data do not run concurrently. Furthermore, the scheduling of tasks onto threads must be efficient, minimizing the time threads spend

waiting, if good performance is to be achieved. Meeting these requirements manually is a difficult problem when a simulator’s form obscures potential tasks and data sharing. In the following subsections we describe how automated parallel code generation for structural models deals with each of these requirements, obviating the need for manual parallelization.

2.1. Task formation

The general flow of an LSE simulator is shown in Figure 1(a). After an initialization step, control enters the “main loop”. For each time step, there are four substeps: PHASE_START, PHASE, PHASE_END, and between-cycle book-keeping. As time steps generally correspond to a single clock cycle of the simulated machine, the substeps can be described in terms of state machine behavior. During PHASE_START, components produce output signals dependent only upon state (i.e., Moore state machine outputs). During PHASE they produce outputs depending on inputs and possibly state (Mealy state machine outputs). During PHASE_END, they update their state.

Within each substep, the framework invokes methods for each substep from each of the component instances. The user does *not* have control of the order in which these invocations are made within the substep and must therefore write components which do not require any particular invocation order to work correctly. The method invocations made during the substeps are the tasks used in the automated parallelization.

The ordering of component method invocations during PHASE_START and PHASE_END is arbitrary as no communication takes place during these substeps. During PHASE, however, signal values may be written and read concurrently, requiring an invocation order that obeys a formal model of computation defining the semantics of communication and concurrent execution. The model used by LSE is known as the Heterogeneous Synchronous Reactive (HSR) model[7]. While the details are beyond the scope of this paper, it is possible to create a static execution schedule of method invocations (possibly repeating some invocations) which guarantees that the collection of signal values reaches a fixed point in each cycle[7, 15]. The resulting schedule is a directed acyclic graph of invocations which may be executed in any valid reverse topological ordering.

2.2. Shared state discovery

There are four ways in which data can be legally shared between component instances in LSE:

Through signals All signals are part of the model structure and can be analyzed by the framework to determine

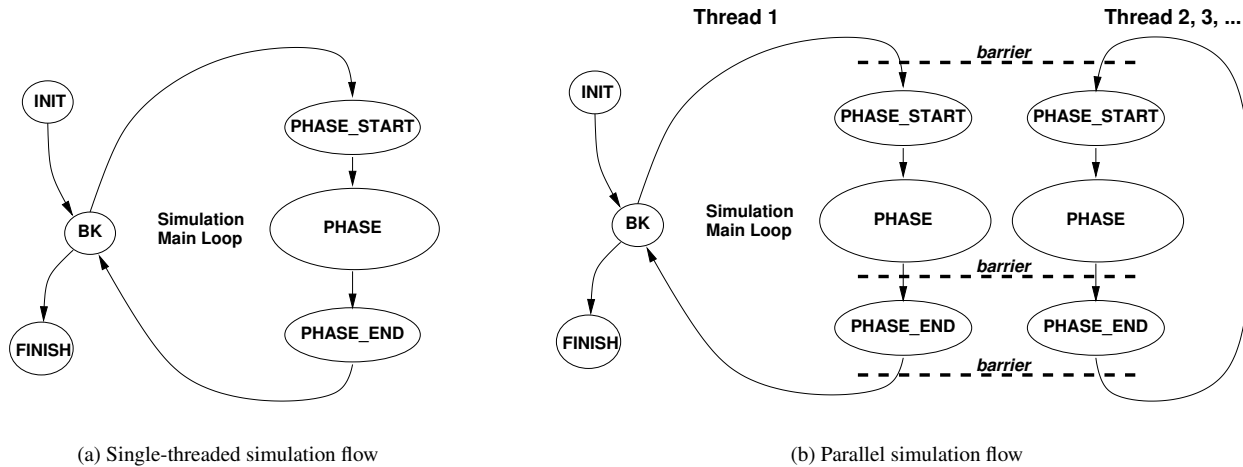


Figure 1. Simulation flows

data dependencies. Note that components are written with an *atomicity* assumption: no input signals examined by a component method may change while the method is running. This constraint is represented as an automatically generated *conflict* between potential producers and consumers of signals.

Through user-defined “runtime variables” Data dependencies through these variables within PHASE must satisfy the same producer-consumer relationship as some signal and thus need no additional analysis. Data dependencies within PHASE_END and PHASE_START are not allowed.

Through library calls Data dependencies through library calls must obey the same restrictions as data dependencies through runtime variables. Non-thread-safe library calls require that the framework avoid simultaneous calls to the library. As not all library calls are part of the structural description (due to the black-box nature of component code), some minimal user input is required to mark conflicts between the component methods making these calls.

Internally A component instance shares data internally with other invocations of its own methods. This constraint is represented as an automatically generated conflict between method invocations of the same instance.

2.3. Synchronization

The structure of the simulator once automated parallelization has been carried out is shown in Figure 1(b).

The PHASE_START and PHASE substeps are parallelized together, while PHASE_END is parallelized separately; barriers (represented by heavy dashed lines) are used to separate each parallelized section.

Cross-thread data dependencies are respected by automatically adding binary semaphores to indicate when data is available. The framework code waits on these semaphores before invoking consumer component methods and sets the semaphores after invoking producer component methods. Redundant semaphores are removed.

Mutual exclusion of component methods which have a conflict is accomplished by automatically acquiring and releasing locks around the conflicting invocations. Most potential locks are removed through a lock elision algorithm which looks at the cross-thread data dependencies to determine whether two invocations may truly run in parallel.

2.4. Scheduling

The input to the scheduling problem for PHASE_START and PHASE is the directed acyclic graph (DAG) of component instance method invocations produced as the single-thread schedule augmented with nodes for PHASE_START invocations and edges causing PHASE_START method invocations to precede other method invocations in the same component instance. For PHASE_END the input is a simple DAG of invocations with very few edges (there are actually two subsubsteps of PHASE_END which must proceed in order and the edges are used to enforce that).

Scheduling of a task DAG across multiple processors is a well-known problem for which many heuristics have been developed. We describe four strategies for performing this scheduling:

2.4.1 List schedulers

Our first strategy (called **LIST**) is a list scheduler similar to that of Yu[22]. A priority queue of tasks whose predecessor nodes have all been scheduled is maintained. The highest priority task is removed from the queue and scheduled in the earliest available time slot not violating data dependencies in any thread; the queue is then updated. This is repeated until the queue is empty. Priority is the length of the longest directed path originating at the task, where length is the sum of the (estimated) execution times of each task.

List scheduling takes into account load balancing and critical path length. However, it ignores two very important aspects of the scheduling problem for automated parallelization. First, it fails to take into account the potential for conflicts which prevent two tasks from executing concurrently. Second, cache effects on signal values and other data cause the execution times of tasks to depend heavily upon how other tasks accessing the same data (particularly tasks from the same component instance) have been scheduled.

Our second strategy (**LOCKAVOID**) reduces conflicts by actively avoiding locking during the thread selection step of the list scheduler. Tasks are scheduled onto the earliest time available on a thread which does not result in a lock being needed. If all threads would require a lock, the task is *skewed* in time to reduce the likelihood of lock contention; it is purposefully scheduled later than the earliest possible time if that time would be “too close” to a time when the lock is planned to be owned by another thread. Lower priority tasks can then be placed in the resulting schedule hole.

List scheduling has been extended in [22] to include communication costs, but this extension is not appropriate for scheduling around cache misses. The reason is that cache misses require the receiving thread of the communication to initiate the transfer and effectively block the receiving thread. Communication cost extensions to list scheduling assume that the sender initiates the transfer and that useful work may be done by the receiver during the communication time.

2.4.2 Clustering

An alternative to list scheduling is to make thread assignments by trying to *cluster* related tasks together in the same thread. Such an approach can be used to improve cache locality of signal values and other data. After clustering, a list scheduler is run with the thread assignments constrained to those chosen by the clustering strategy; lock skewing as previously described is used to reduce lock conflicts.

Our third strategy (**INSTCLUSTER**) groups all method invocations from the same component instance into one cluster. Clusters are then combined based upon the signal bandwidth between them, starting with the highest band-

width pair of clusters. Signal bandwidth between clusters is updated as they are combined. This combination continues until there are either as many clusters as there are threads or a minimum bandwidth value has been reached. The remaining clusters are then mapped onto threads in a load-balanced fashion. Note that this strategy takes into account probable cache effects and load balancing, but not the critical path.

Our final strategy (**IDSC**) is derived from Dominant Sequence Clustering (DSC)[21]. DSC attempts to cluster tasks on the critical path together for an infinitely parallel machine, observing that the critical path changes as decisions are made and threads become utilized. It takes into account communication costs, load balancing, and the critical path. It then assigns clusters to threads. While as noted for list scheduling, communication costs do not model cache effects well for the purpose of determining final schedules, they may be helpful for determining the estimated path lengths required in DSC. We therefore set the communication costs to be the the number of input signals referenced by the task. We also modify the assignment phase to be cache-aware by preferentially assigning clusters to threads which already contain invocations of components in the cluster.

2.5. Related work

Previous efforts to parallelize microarchitecture simulators for multi-processors have used coarse-grain processor-level task granularity with manual or hard-coded assignment of tasks to threads. The Wisconsin Wind Tunnel II[12] uses parallel discrete event simulation to simulate a multi-processor memory hierarchy. Chidester[3] created a parallel multiprocessor simulator by running a modified copy of SimpleScalar in each thread. Barr[1] has created parallel multiprocessor simulators in Asim, another structural modeling system, by defining new port types and changing the models to use these types. While the addition of framework support is similar in spirit to what we have done, it still requires manual thread assignment and model changes to use the new port types.

There is an extensive literature on parallel and distributed discrete event simulation. A good overview is given in [9]. The primary problem addressed is agreement about global time. Parallel HSR simulation is more closely related to multiprocessor scheduling, for which there is again an extensive literature; see [18] for additional references.

3. Evaluation of automated simulator parallelization

We evaluate the effectiveness of automated simulator parallelization for chip multi-processors by using the previously described scheduling strategies to generate parallel

simulators for a family of CMP models and measuring their speed.

3.1. Measurement methodology

Our CMP models have a tile-based architecture. Each processing tile contains a processor with its first-level data and instruction caches, a portion of the distributed second-level cache, and a memory controller. The tiles are connected via a standard wormhole routing network organized as a mesh. Five different sizes of CMP are used: 1, 2, 4, 8, and 16 processors. The processor core is an 1-wide in-order core with branch prediction, chosen to be similar to the PowerPC 405 core used in Section 5.

We generate one, two, three, and four-threaded simulators using each of the scheduling strategies. Two to four threads is the number of threads available in today’s inexpensive workgroup servers, in particular those becoming available which themselves use CMPs. Task length is predicted from the number of potential input and output signals of the method invocation using a regression-based model derived from measurements upon the 2-core CMP model. In addition, a manually-guided clustering of invocations to threads (called **GUIDED**) is used for comparison; the guided clustering attempts to map complete tiles to threads, though it splits tiles in the 2-way and 1-way CMP models. This clustering should enjoy good cache behavior in addition to being intuitive and similar to previous work. We emphasize that this guided clustering is *not* a manual parallelization of the simulator; it merely overrides the thread assignments of automated clustering.

Each simulator is run on the *FFT* and *Radix* benchmarks from the Splash-2 benchmark suite[20], linked with a mini-OS which provides processor initialization and simple I/O capabilities. The benchmarks were compiled using gcc 3.3 with compiler flags `-g -O2 -msoft-float`. Input sizes and parameters were chosen to use all simulated processors and to provide several minutes of running time. Results are presented for both benchmarks; they are so similar that numbers quoted in the discussion are all from *Radix*.

All experiments were run on a 4-processor server with 6 GB of physical memory running Red Hat Linux Fedora Core 3. The processors used are 2-GHz AMD Opteron processors, which each have 64KB L1 data, 64KB L1 instruction, and 1MB L2 private caches. Memory is partitioned across processor nodes. The *oprofile*[14] tool is used to profile simulator execution. This tool records the program counter (PC) every N times an event such as a clock cycle or a cache miss occurs, where N can be set by the user. A post-processing step then maps the PC back to instructions, source lines, and functions.

3.2. Results

We begin by looking at the performance scalability of the non-parallelized simulator with increasing CMP model size. Figure 2 shows time per simulated cycle vs. number of simulated processors, normalized to one simulated processor. The speed of the 1-way CMP is 36,130 simulated cycles/sec. As the number of simulated processors grows beyond four, there are large slowdowns. These slowdowns are due to a sharp increase in the number of L2 cache misses as the working set of the simulator increases. The 4-way CMP model incurs only 300 data L2 cache misses per simulated cycle, but the 8-way CMP model incurs 2500 data misses and the 16-way CMP model nearly 5000 data misses per simulated cycle. Such cache behavior has a significant effect upon the effectiveness of automated parallelization.

Figure 3 shows the speedups achieved by automated parallelization. Each panel shows the speedup achieved using each scheduling strategy measured relative to a single-threaded simulator as the model size changes. Note that the baseline single-threaded simulator is different for each model size. Different panels correspond to different benchmarks and numbers of threads.

The most obvious trend visible from Figure 3 is that speedup vs. a single-threaded simulator increases as the size of the model increases. These results can be explained using two factors. First, cache effects should penalize smaller models; when the model is small enough that its working set fits within the L2 cache of a single processor, parallelization will increase cache misses relative to a single-threaded simulator. On the other hand, for larger models whose working sets did not fit within the L2 cache of a single processor but do fit within the combined L2 caches of multiple processors, parallelization can decrease cache misses and boost performance. Second, as the model increases in size, scheduling strategies should have more opportunities to schedule around locks and achieve higher parallelism.

We can separate these two factors by breaking down overall speedup into cache-caused and parallelism-caused speedup. We compute the cache-caused speedup by comparing the number of CPU seconds spent in component method invocations (as reported by *oprofile*) in the parallelized and single-threaded simulators. The remaining speedup we attribute to parallel execution (though it actually includes and is reduced by all extra overhead due to parallelization). The two speedup components multiply to form the overall speedup. Figure 4 shows the speedup breakdown for *Radix* for two-threaded and four-threaded simulators.

Figures 4(a) and 4(c) show that cache effects are as predicted; smaller models are hurt by cache effects while the largest model is helped by the larger total effective cache size; the tradeoff point appears at around 8 simulated cores

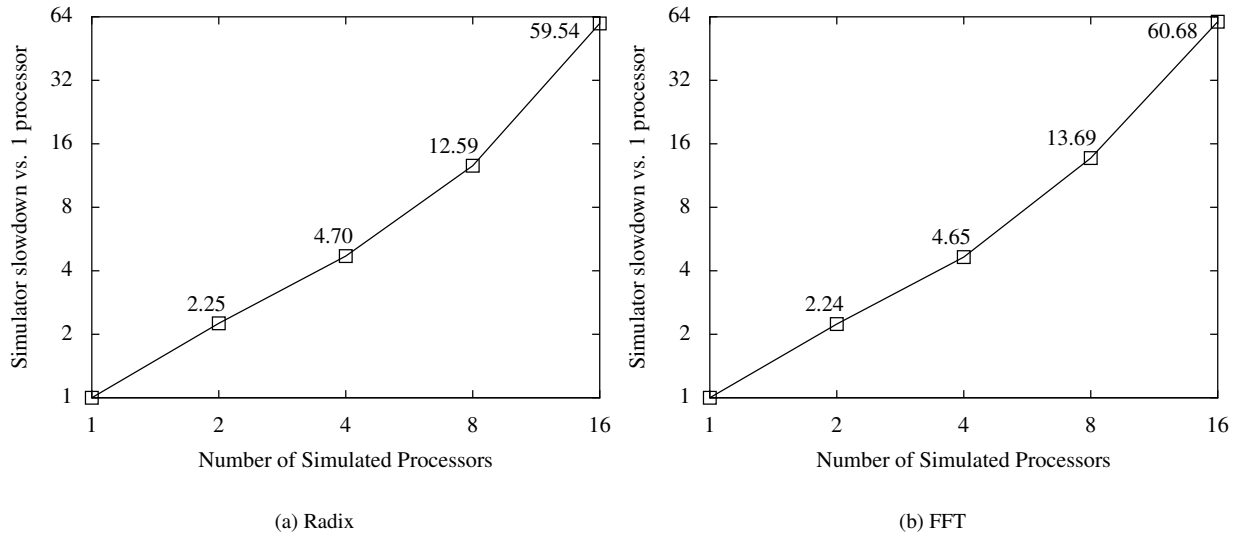


Figure 2. Scalability of single-threaded simulator performance

for 4 threads. This speedup boost due to the larger effective cache size is what leads to superlinear overall speedup. As might be expected, cache effects increase in intensity as the number of threads increases. Both figures also show that clustering strategies are better than non-clustering strategies at handling cache effects; **INSTCLUSTER** is the most effective strategy overall, though **IDSC** is nearly as effective once model size grows beyond four simulated cores. Beyond two simulated cores, both clustering strategies are competitive with manually-guided clustering.

Figures 4(b) and 4(d) indicate how well the strategies find and exploit parallelism. As model size increases, speedup generally increases for most strategies as scheduling opportunities increase. For two threads, there is little difference between the strategies, other than that **LIST** is worse than the rest. However when four threads are used, **IDSC** shows itself to be particularly good at exploiting parallelism, achieving greater than 80% efficiency for four threads. **LOCKAVOID** and **INSTCLUSTER** obtain somewhat lower results. Note also the poor performance of **GUIDED** for smaller models; this is indicative of the difficulty of doing good load-balancing by hand when a good clustering is not obvious to the user.

3.3. Summary of results

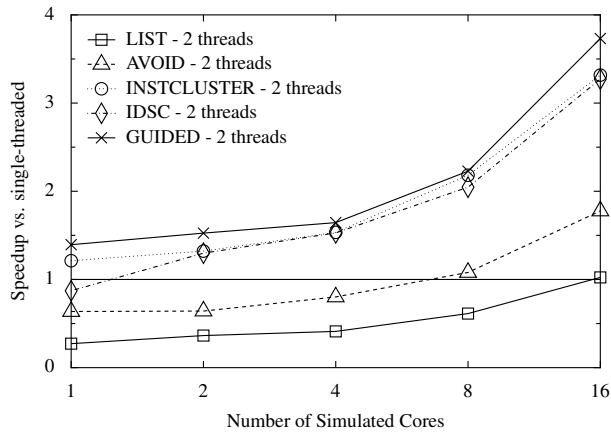
Standard list scheduling (**LIST**) is not effective for task scheduling in the simulator because it ignores both cache and locking effects. Locking effects improve when lock avoidance mechanisms are used, but the schedule is still vulnerable to poor cache locality. Locality im-

proves through use of clustering techniques, but there is no clearly preferred clustering strategy. The simpler instance-based strategy (**INSTCLUSTER**) achieves better locality but poorer parallelism than the more complex strategy (**IDSC**) taking into account the critical path. Overall, **INSTCLUSTER** is better when the model is small and there are few threads, but **IDSC** is better when the model is large or there are many threads.

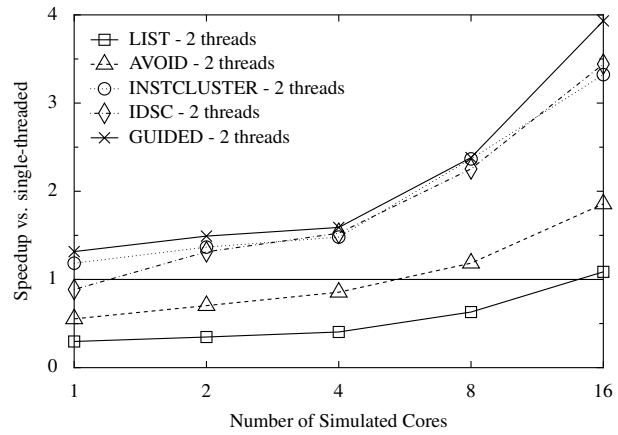
Figure 5 shows the scalability of simulator performance (defined as in Figure 2) as the model size increases for the instance-based strategy scheduler, **INSTCLUSTER**, for different numbers of threads. Ideally, a 4-threaded simulation of a four-way CMP would take as long as a 1-threaded simulation of a 1-way CMP. This is unfortunately not the case; as discussed before, small models do not achieve large parallel efficiency due to the extra L2 data misses implied by parallelization. However, large models do see a large benefit from parallelization due to the increase in effective L2 cache size for data. This benefit is so large that it becomes the primary benefit of parallelization, increasing the range of model sizes for which simulation speed scales approximately linearly with model size.

4. Integrating processors in simulation

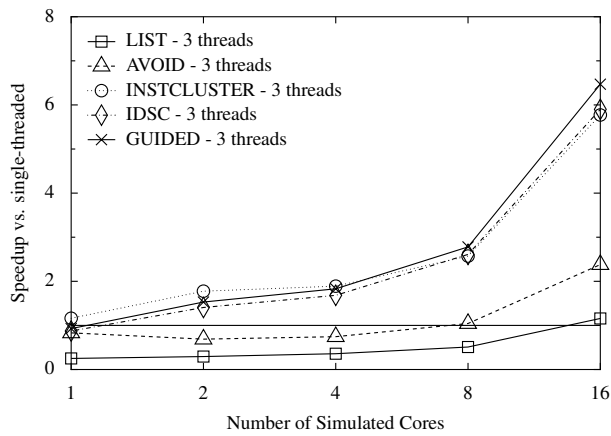
While parallelism in a CMP model can be exploited by using multiple processors to parallelize the simulator, these processors can also be integrated as components of the simulation itself. To do so requires that the simulator be explicitly developed in such a way as to encapsulate the hardware, providing input signals to and receiving output signals from



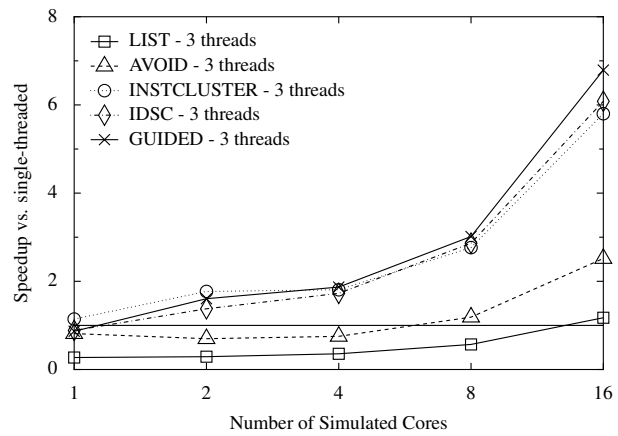
(a) Radix 2-threaded



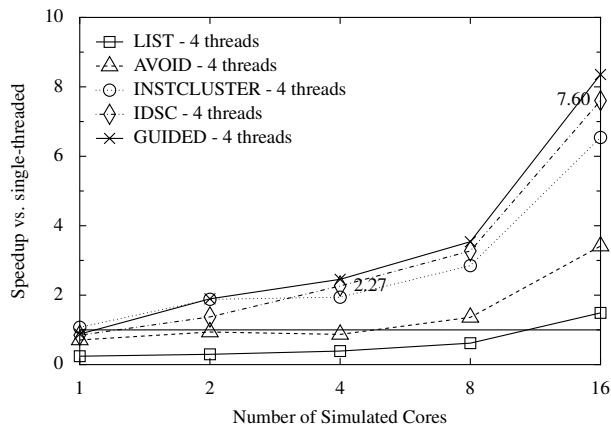
(b) FFT 2-threaded



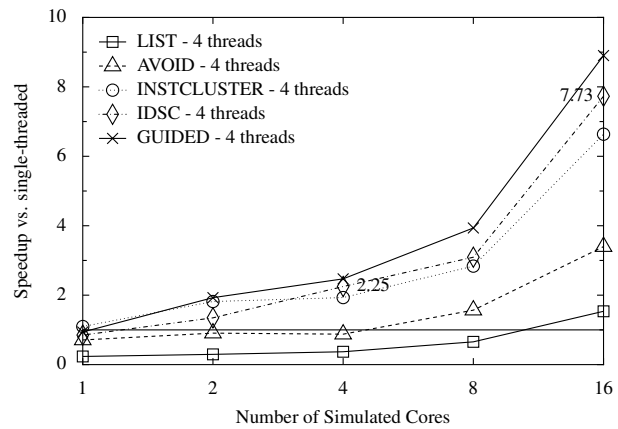
(c) Radix 3-threaded



(d) FFT 3-threaded

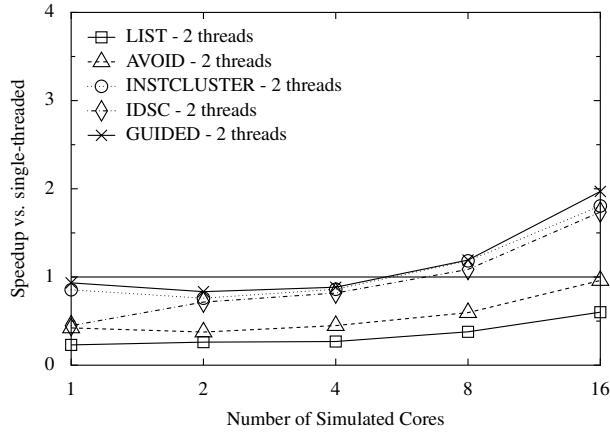


(e) Radix 4-threaded

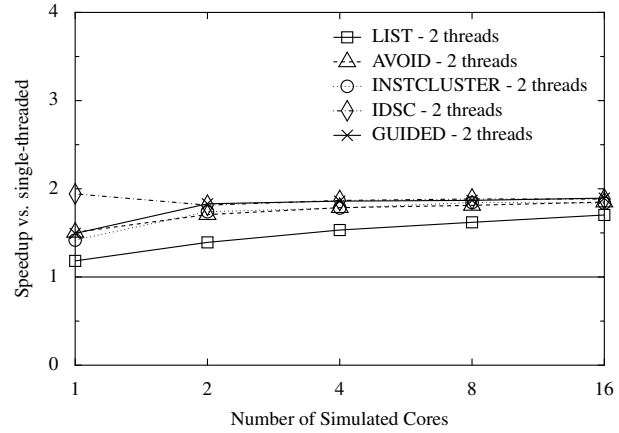


(f) FFT 4-threaded

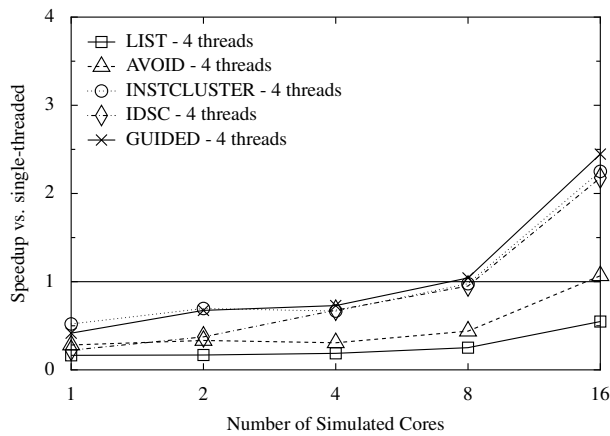
Figure 3. Achieved speedup for multi-threaded simulators by scheduling strategy



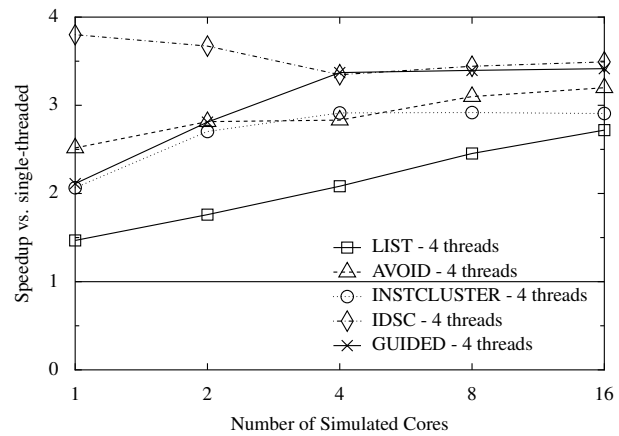
(a) Cache-caused speedup (2 threads)



(b) Parallelism-caused speedup (2 threads)

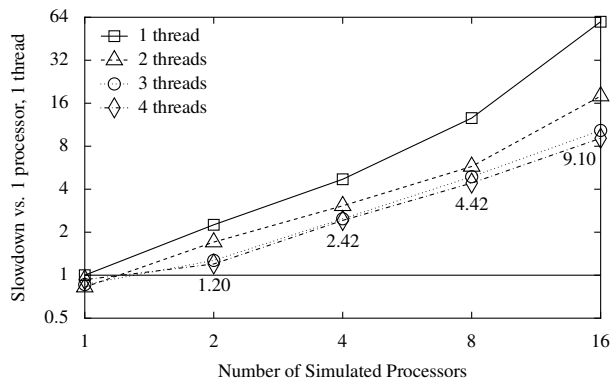


(c) Cache-caused speedup (4 threads)

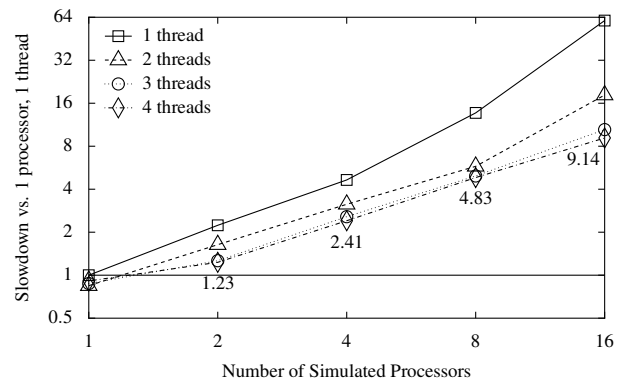


(d) Parallelism-caused speedup (4 threads)

Figure 4. Breakdown of speedup for 2-threaded and 4-threaded simulators on Radix



(a) Radix



(b) FFT

Figure 5. Performance scalability of INSTCLUSTER

the hardware. Concurrent structural simulation frameworks make this process much easier, as this is the normal modeling paradigm: components communicate with each other through signals. As a result, once the hardware component has a corresponding LSE adapter component, integrating the component into the simulator is simply a matter of swapping the software component with the hardware one. No other changes are required to the rest of the simulation model.

Integration of processors (or arbitrary hardware components) can offer additional benefits beyond increases in simulation speed. Two major motivating factors behind the development of structural modeling have been reuse and reduction of error. Using actual hardware can be seen as the ultimate expression of these goals: the component need not be modeled at all and is already completely validated. In addition, simulated components can be incrementally implemented and become hardware components, creating an incremental architecture-to-implementation design flow.

4.1. Integration Requirements

Three requirements must be met when integrating hardware into a structural simulation framework. First, the simulator must control and communicate with the hardware. Second, the hardware must be made to appear as if it obeys the simulator's model of computation. Finally, hardware signals must be translated and mapped to simulator signals.

Control and Communication The simulator must be able to control all of the input signals of the hardware component and to sample all of its output signals. The simulator must also be able to control the hardware clock. We accomplish this by placing hardware input registers, hardware output registers, and a hardware register controlling a clock driver on the interfaces of the hardware component instantiated in a Field-Programmable Gate Array (FPGA), as shown in Figure 6. These registers can be written to and read from via a Linux device driver written for the board upon which the FPGA resides. Device driver methods to write and read these registers are called from an LSE adapter component written specifically for the hardware being integrated.

Model of computation The Heterogeneous Synchronous Reactive (HSR) model of computation which LSE uses requires that components carefully control when signals are set. An output signal must not be given a value before all inputs needed to compute it are known. An output signal should also be given a value as soon as all of the inputs needed to compute it are known. The essential invariant which an LSE adapter component around a hardware component must maintain, therefore, is to not set a signal value

too early (before it would be set by an HSR model of the component) nor too late.

It is useful to think of the outputs of the hardware to be integrated as being either Moore or Mealy state machine outputs. At the beginning of the simulation timestep, Moore machine outputs should be available from the hardware, and LSE requires that they be driven out of the LSE adapter during the `PHASE_START` method of the simulation time step. As inputs become available during the timestep, LSE calls the `PHASE` method of the adapter. The adapter provides its inputs to the hardware and then samples the hardware's outputs. It then drives in LSE those outputs which are known to have resolved to a known value; as the hardware will *not* indicate this, knowledge of the zero-cycle paths through the hardware must be put into the adapter. Finally, during the `PHASE_END` method, the adapter provides its inputs to the hardware and causes the hardware clock to tick.

Translation and mapping LSE creates three signals for each connection made to a component. These signals – called data, enable, and ack – are used to provide default flow-control behavior across components. Furthermore, data has an implicit “data valid” signal carried with it. Most hardware components will not use precisely the same scheme. To resolve this, the adapter component shown in Figure 6 translates the hardware signals into LSE signals signifying the hardware's flow-control semantics, and vice-versa. It may also translate datatypes; for example, transforming the raw bits off of a command bus into some enumerated type that lists the commands in human-friendly form.

4.2. Related Work

Hardware emulation has been used for microprocessor design and verification in many efforts in the past. Ray and Ho[17] describe using an FPGA prototyping board to verify a synthesizable processor design. Within industry, microprocessors such as the AMD K5[10] and the Sun UltraSPARC I[11] have been verified through hardware emulation using specialized emulation systems constructed from FPGAs. These efforts have required that the entire design be emulated, not just portions of it, and that the entire design be described in a hardware-synthesizable language.

Nakamura, et al.[13] present a method for integrating C/C++ simulators with hardware implemented in FPGAs. This is similar in some ways to our work, but the simulators used are written in an ad-hoc fashion and must intersperse driver calls throughout the simulation without the benefit of a structured simulation cycle. Davis, et al.[5] present a prototyping system for multiprocessors built out of FPGA boards with processors. This system does not integrate with simulators, but instead is intended to be used to prototype

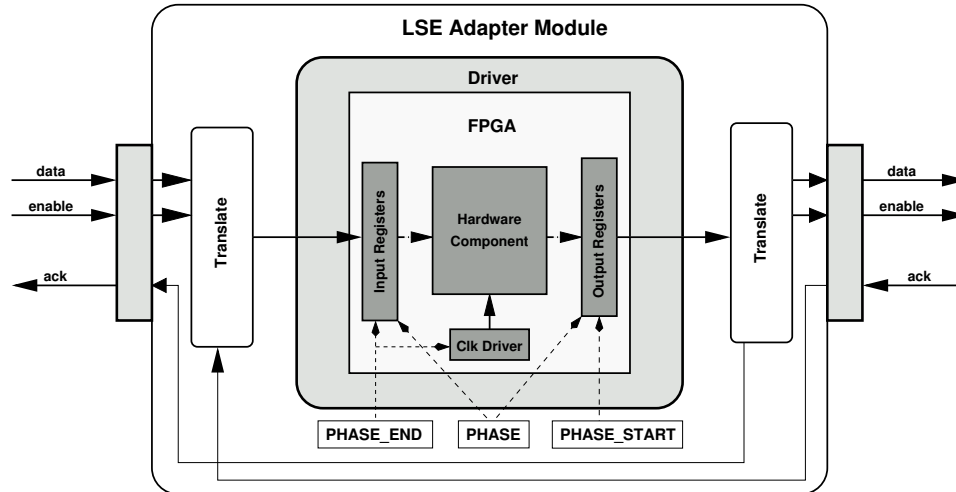


Figure 6. A Hardware Component in LSE

the entire multiprocessor. Chiou[4] partitions simulation between functional and timing simulation and offload the simulation onto FPGAs, but does not present results.

DirectRSIM[6] uses direct execution to model the functionality of processors in a multi-processor system, thus “integrating” them in a sense. The communication mechanism with the processors is not cycle-accurate; the binary being run by the processors is instrumented to call the timing simulator at each memory operation and pass it the execution path taken since the last operation. The timing simulator then performs a timing-only simulation of the processor to determine the time at which the memory operation would actually have been issued.

Finally, integration of different models of computation was one of the goals of the Ptolemy[2] project, out of which the HSR model originated.

5. Evaluation of hardware integration

In this section we demonstrate the feasibility of integrating processors into a concurrent structural simulation model by integrating hardware PPC405 cores into 1-, 2-, 4-, and 8-way chip multi-processor models. This integration is done by simply replacing the PowerPC software LSE component with a PPC405 LSE adapter component. Having swapped the components, we then compare the new simulator’s performance to the software-only models.

5.1. Experimental infrastructure

Up to four AMIRIX AP130 PCI boards are used, each board containing one Xilinx Virtex-II Pro XC2VP30 FPGA containing two PPC405 cores implemented as hard macros. 54% of the logic resources (slices) on these mid-size FPGAs

are used. The boards share the host system’s single PCI bus, which is 32 bits wide and runs at 33MHz.

The simulations use the minimal number of boards for a given size of CMP model: i.e. the number of cores divided by two. Because the external bus of the PPC405 has no zero-cycle input-to-output paths, a PHASE method is not needed in the adapter. Instead, it simply reads the core’s outputs at PHASE_START and drives inputs to the core and toggles the clock at PHASE_END. The adapter translates the core’s bus signals to and from cache transactions understood by the simulated cache models. It also handles some special cases of the cache protocol, such as responding to invalidation requests, that the core does not expect to see.

The hardware integration experiments use the same benchmarks and inputs as those used in Section 3. Three families of CMP are considered: the first, **GRID**, is the complex, tiled architecture used in Section 3. The second, **BUS**, has a bus-based architecture and a monolithic L2 cache. The final family, **PERF**, has perfect L1 data and instruction caches and serves to illustrate speedup when nearly all of the simulation has been moved to hardware. The host system, which is also used for software-only speed measurements, has a Pentium 4 630 (3.0 GHz “Prescott” with a 2MB L2 cache) system running the 32-bit version of Red Hat Fedora Core 3.

5.2. Results

Table 1 shows the speed measured in simulated cycles per second of the purely software-based simulators and the hardware-integrated simulators for both *Radix* and *FFT*. Also shown for *Radix* is a profile of the percentage of time spent in the adapter component and the board driver in the hardware-integrated models.

| Model | Radix | | | | | FFT | | |
|------------|-------------------------------|--------|---------|---------------------------|---------|-------------------------------|--------|---------|
| | Speed (cycles/sec) SW-only | HW | Speedup | % time spent in driver | adapter | Speed (cycles/sec) SW-only | HW | Speedup |
| 1-way PERF | 37314 | 160978 | 4.31 | 68.7 | 10.8 | 37461 | 162920 | 4.35 |
| 2-way PERF | 17277 | 71316 | 4.13 | 56.5 | 17.8 | 17312 | 72329 | 4.18 |
| 4-way PERF | 7496 | 38363 | 5.12 | 46.7 | 22.7 | 7481 | 39049 | 5.22 |
| 8-way PERF | 3136 | 17930 | 5.72 | 40.8 | 27.3 | 3130 | 18229 | 5.82 |
| 1-way BUS | 24720 | 41860 | 1.69 | 12.7 | 7.0 | 24858 | 41974 | 1.69 |
| 2-way BUS | 11544 | 21494 | 1.86 | 12.7 | 7.7 | 11731 | 21664 | 1.85 |
| 4-way BUS | 5074 | 9759 | 1.92 | 11.5 | 9.9 | 5154 | 9873 | 1.92 |
| 8-way BUS | 2205 | 4278 | 1.90 | 9.4 | 11.6 | 2146 | 4359 | 2.03 |
| 1-way GRID | 22210 | 31692 | 1.43 | 9.4 | 5.3 | 22348 | 31898 | 1.43 |
| 2-way GRID | 9551 | 12341 | 1.29 | 7.5 | 5.3 | 9584 | 12408 | 1.29 |
| 4-way GRID | 3951 | 4761 | 1.21 | 5.7 | 6.1 | 3983 | 4767 | 1.20 |
| 8-way GRID | 1649 | 2092 | 1.27 | 4.7 | 6.3 | 1604 | 2101 | 1.31 |

Table 1. Hardware-integrated vs. software-only simulators

Large speedups are seen for the **PERF** models. Integration of hardware has accelerated simulation by moving most of the computation into efficient, highly-parallel hardware. Up to 69% of simulation time is spent within the driver; of this time, nearly all of it is spent waiting for PCI read transactions to complete. Notice also that the proportion of time spent in the adapter relative to the driver increases as the number of cores increases. We believe this is because the frequency with which memory transactions occurs increases as the number of cores increases due to the increased amount of time spent in synchronization constructs such as locks in the target benchmarks. As a result, the adapter must spend proportionally more time translating signals, as the driver’s time to provide the signals to and from the hardware each cycle remains relatively constant.

We stress that the simulation model of the PPC405 core is a very simple parameterized generic core model which does not model the PPC405 particularly accurately. A more detailed and accurate core model would likely be much slower, causing the speed benefit of hardware integration to be even higher. This effect would be even more pronounced for more complex cores. Thus these speedup results should be considered to be very conservative.

The **BUS** and **GRID** models show less speedup. This is a consequence of Amdahl’s Law. As the memory hierarchy becomes more complex and takes up a much larger proportion of simulation time, the potential speedup due to reducing the time spent in the processors is reduced. The large reduction in speedup seen implies that implementation and integration of portions of the memory hierarchy (for example, first-level caches) in hardware could be worthwhile.

6. Conclusion

We have shown that the use of structural concurrent models for CMPs allows simulators to exploit structure and parallelism in the CMP model. This can be achieved either through automated parallelization of the simulator or by integrating processors into the simulation itself. Structural models make it possible to use either technique without requiring the user to manually parallelize or re-design the simulator.

Automated parallelization is possible because the structural, concurrent nature of the model provides a rich assortment of tasks which can then be automatically scheduled. We show that shared data access and cache behavior are important to consider while scheduling and demonstrate an automatic scheduler which can achieve moderate (2.27) speedup on a 4-way CMP model and large (7.60) speedup on a 16-way CMP model. Future work will focus on further improving the data cache locality of the parallelized simulator.

We have also demonstrated that processors can be integrated into a structural CMP model without requiring re-design of the model, achieving a speedup of 5.82 for an 8-way CMP model with perfect caches and 1.31 for an 8-way CMP model with a complex tiled memory architecture. Besides this speed advantage, hardware component integration offers advantages of accuracy, reduced model development time, and incremental architecture-to-implementation design flow possibilities. Future work will improve board communication overhead with the driver and explore the performance issues involved in simulating multiple microarchitectural components on multiple FPGAs and FPGA boards as well as the effects of various hardware parameters and configurations on performance.

7. Acknowledgements

This work has been supported by the National Science Foundation (CAREER CCF-0133712, NGS-0305617). Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation. We would also like to acknowledge Krista Marks and Glenn Steiner of Xilinx Corporation for the donation of FPGA prototyping equipment and software.

References

- [1] K. C. Barr, R. Matas-Navarro, C. Weaver, T. Juan, and J. Emer. Simulating a chip multiprocessor with a symmetric multiprocessor. In *Boston Area Architecture Workshop*, January 2005.
- [2] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation*, 4:155–182, 1994.
- [3] M. Chidister and A. George. Parallel simulation of chip-multiprocessor architectures. *ACM Transactions on Modeling and Computer Simulation*, 12(3):176–200, July 2002.
- [4] D. Chiou. FAST: FPGA-based acceleration of simulator timing models. In *In Workshop on Architecture Research Using FPGA Platforms*, February 2005.
- [5] J. D. Davis, L. Hammond, and K. Olukotun. A flexible architecture for simulation and testing (FAST) multiprocessors systems. In *In Workshop on Architecture Research Using FPGA Platforms*, February 2005.
- [6] M. Durbhakula, V. S. Pai, and S. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, January 1999.
- [7] S. A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, University of California, Berkeley, 1997.
- [8] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 0018-9162:68–76, February 2002.
- [9] A. Ferscha. Parallel and distributed simulation of discrete event system. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 1003–1041. McGraw-Hill, 1996.
- [10] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, M. Wang, and J. Nishimura. Hardware emulation for functional verification of K5. In *Proceedings of the 33rd Design Automation Conference (DAC)*, pages 315–318, 1996.
- [11] J. Gateley, M. Blatt, D. Chen, S. Cooke, P. Desai, M. Doreswamy, M. Elgood, G. Feierbach, T. Goldsbury, D. Greenley, R. Joshi, M. Khosraviani, R. Kwong, M. Motwani, C. Narasimhaiah, S. J. N. Jr., T. Ozeki, G. Peterson, C. Salzman, N. Shayesteh, J. Whitman, and P. Wong. UltraSPARC-I emulation. In *Proceedings of the 32nd Annual Design Automation Conference (DAC)*, pages 13–18, 1995.
- [12] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast and portable architecture simulator. In *Workshop on Performance Analysis and its Impact on Design (PAID)*, June 1997.
- [13] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In *Proceedings of the 41st Annual Design Automation Conference (DAC)*, pages 299–304, 2004.
- [14] OProfile. Web site: <http://oprofile.sourceforge.net/>.
- [15] D. Penry and D. I. August. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference (DAC)*, June 2003.
- [16] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation (MOBS)*, June 2005.
- [17] J. Ray and J. C. Hoe. High-level modeling and FPGA prototyping of microprocessors. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 100–107, 2003.
- [18] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*, chapter 5. Marcel Dekker, Inc., New York, NY, 2000.
- [19] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.
- [20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.
- [21] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.
- [22] W. H. Yu. *LU Decomposition on a Multiprocessing System with Communication Delay*. PhD thesis, University of California at Berkeley, Berkeley, CA, 1984.