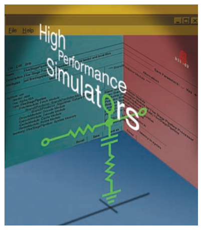
COVER FEATURE

Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors



Rsim is a publicly available architecture simulator for shared-memory systems built from processors that aggressively exploit instruction-level parallelism. Modeling ILP features in a multiprocessor is particularly important for applications that exhibit parallelism among read misses.

Christopher J. Hughes
University of Illinois at Urbana-Champaign

Vijay S. Pai
Rice University

Parthasarathy Ranganathan
Compaq Western Research Laboratory

Sarita V. Adve
University of Illinois at Urbana-Champaign

Given the complexity and associated cost of building modern computer systems, simulation is often the only practical way to test architectural ideas and assess system performance. Simulators provide the flexibility to modify and analyze the impact of various architectural parameters and components as well as enable more detailed statistics collection than real hardware. These benefits make simulation useful even for projects that will eventually implement hardware.

Prior to 1994, most academic shared-memory multiprocessor studies largely ignored the processor model, focusing instead on the memory system as the most important performance bottleneck. These studies assumed a simplistic processor model based on in-order issue, blocking reads, and no speculation. However, the early 1990s saw several announcements of commercial shared-memory systems using processors that aggressively exploited instruction-level parallelism (ILP) such as the MIPS R10000, Hewlett-Packard PA8000, and Intel Pentium Pro. These processors had the potential to reduce memory read stalls by overlapping read latency with other operations, possibly changing the nature of performance bottlenecks in the system.

Because no shared-memory ILP systems or simulators were available at that time, we designed Rsim—originally an acronym for Rice simulator for ILP multiprocessors—to study such systems. Two major questions guided our efforts:

- Does processor microarchitecture influence shared-memory performance and design to the extent that it justifies its detailed modeling and associated performance costs in a shared-memory simulator?
- With simple processor-based simulators already taking a long time to run, could we build such a detailed simulator efficiently enough to perform substantive architecture studies in reasonable time?

Our experience with Rsim demonstrates that modeling ILP features is important even in shared-memory multiprocessor systems. In particular, current simple processor-based approximations cannot model significant performance effects for applications exhibiting parallel read misses. Further, recent shared-memory designs—for example, aggressive implementations of sequential consistency[1]—directly use the aggressive ILP-enhancing features of modern processors that simple processor-based simulators do not model.

We have also demonstrated that significant multiprocessor studies can be performed with the current speed of ILP simulators. However, improving their speed is crucial for future workloads. Our

0018-9162/02/$17.00 © 2002 IEEE

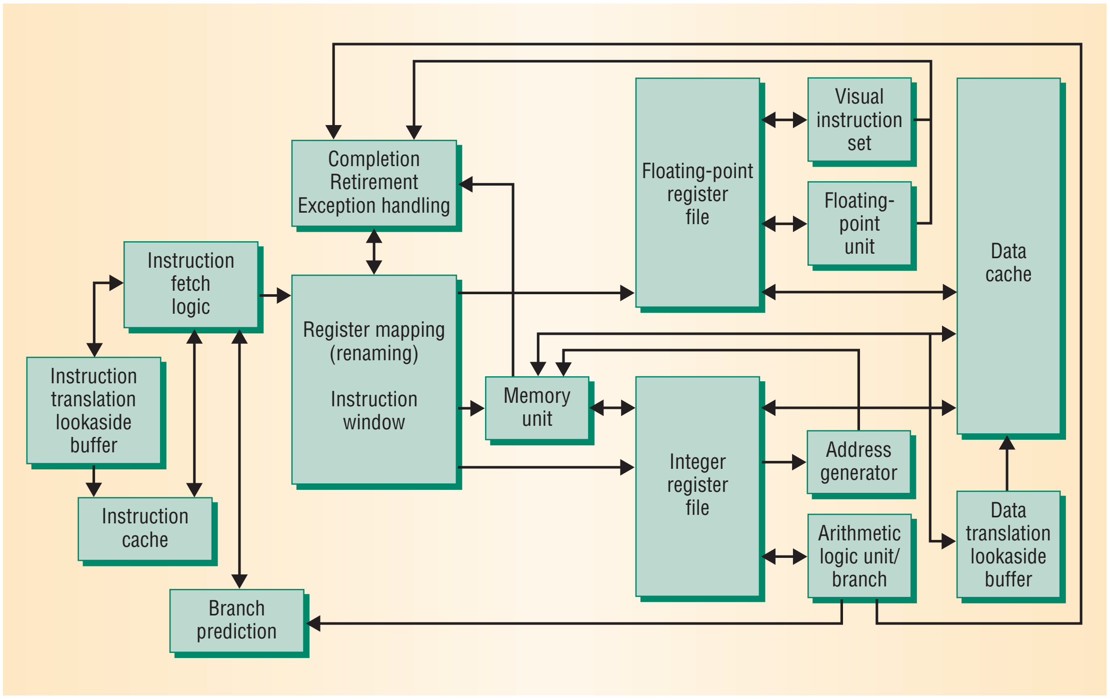


Figure 1. Rsim processor model. The most aggressive processor microarchitecture modeled by Rsim is closest to the MIPS R10000.

work with Rsim suggests several promising techniques in this fertile area of research.

RSIM SYSTEM

Rsim consists of several interchangeable modules to model a range of architectures. At the time of its development, there were no publicly available ILP processor simulators and only limited documentation of commercial systems. We based the processor model primarily on a preprint version of the MIPS R10000 architecture manual, information from product announcements, and consultations with colleagues.

We based major portions of the memory and network subsystems on code from a previous-generation simulator, the Rice Parallel Processing Testbed (RPPT),[2] and followed industry experts' recommendations in setting their parameters. Because most of our development infrastructure, such as the compilers, was based on Sun Microsystems' processors, we used a subset of the Sparc V9 for Rsim's instruction set architecture.

The online reference manual (http://www.ece.rice.edu/~rsim/manual.ps) provides comprehensive documentation on Rsim version 1.0, which we made publicly available in 1997 free of cost for noncommercial use.

Processor microarchitecture

Rsim models processors that exploit varying amounts of ILP. Possible configurations range from single-instruction issue, in-order (static) instruction scheduling, and blocking memory operations to multiple-instruction issue, out-of-order (dynamic) instruction scheduling, and nonblocking memory operations. Other key features include

- register renaming,
- static and dynamic branch prediction,
- speculative memory disambiguation,
- software-controlled nonbinding prefetching,
- multimedia instruction set extensions,
- support for multiple memory-consistency models and various ILP-specific optimized implementations of these models, and
- simultaneous multithreading.

Most processor parameters are user configurable—for example, instruction issue width, instruction window size, and number of functional units.

Rsim's processor architecture, as shown in Figure 1, approximates the MIPS R10000. In particular, Rsim models the

- active list, which holds the currently active instructions and corresponds to the reorder buffer or instruction window of other processors;
- register map table, which holds the mapping from the logical to physical registers; and
- shadow mappers, which allow single-cycle state recovery on a mispredicted branch.

Rsim fetches, decodes, and retires—removes from the instruction window—instructions in program order; instructions can issue, execute, and complete out-of-order.

Rsim currently supports static branch prediction, dynamic branch prediction using either a two-bit history scheme or a two-bit agree predictor, and prediction of return instructions using a return address stack. Rsim allows multiple predicted branches at a time as long as each outstanding

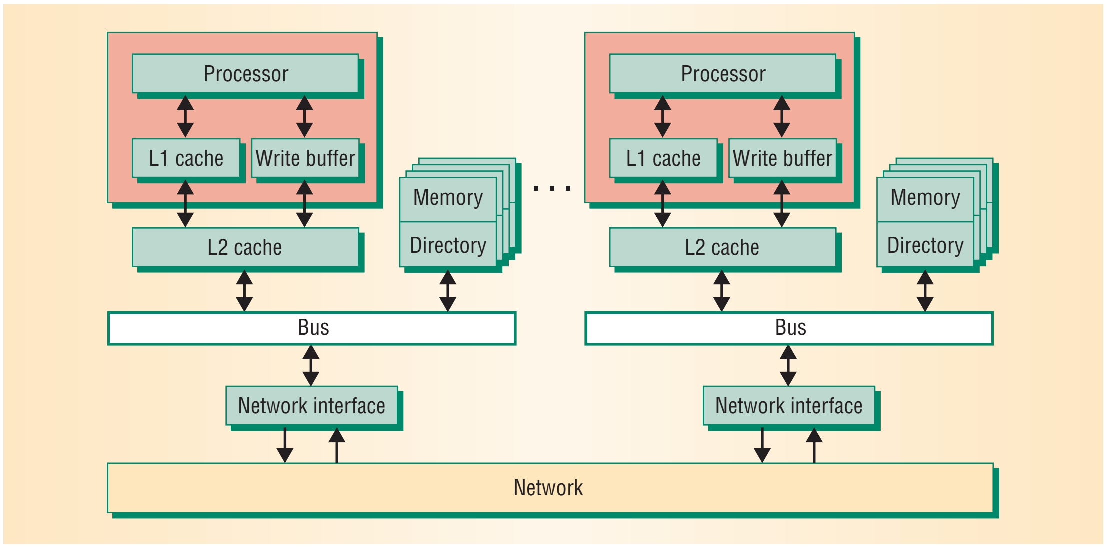


Figure 2. Rsim multiprocessor system. Each node consists of a processor and cache hierarchy along with part of the physical memory, its associated directory, and a network interface. A split-transaction bus connects the secondary cache, memory and directory module, and network interface.

branch has at least one shadow mapper. These branches can resolve out-of-order as well.

The Rsim processor also supports the visual instruction set (VIS) extensions to the Sparc V9 architecture targeted at accelerating media processing. The processor supports all instructions except blocked loads and stores and the array instruction. The VIS implementation is closely modeled after the UltraSparc-II and operates on the floating-point register file.

Cache and memory system

Rsim supports a two-level cache hierarchy with separate first-level data and instruction caches and a unified second-level cache. The first-level data cache is multiported, pipelined, and either write-through or write-back. The first-level instruction cache is multiported, pipelined, and write-through. The second-level cache is pipelined and write-back. Systems with write-through first-level caches include a coalescing write buffer. Both data caches are lockup-free: They store the state of outstanding requests in miss status holding registers (MSHRs) and coalesce requests to the same line.

The main memory model is simple but allows interleaving and is accessed through a pipelined split-transaction bus. Rsim also supports instruction and data translation lookaside buffers (TLBs) with hardware miss handlers. Most cache and memory system parameters—including the number of L1 cache ports, the number of L1 or L2 MSHRs, cache sizes, and all latencies—are user configurable.

Multiprocessor system

Rsim simulates several variations on a base hardware, directory-based cache-coherent nonuniform memory access (CC-Numa) shared-memory multiprocessor. As Figure 2 shows, each of the base system's nodes consists of a processor and cache hierarchy along with part of the physical memory, its associated directory, and a network interface. A split-transaction bus connects the secondary cache, memory and directory module, and network interface. For remote communication, Rsim supports a wormhole-routed two-dimensional mesh network; other configurations are possible but not tested. To avoid deadlocks, the system includes separate request and reply networks. Again, most network parameters, such as network width, flit size, and flit delay, are user configurable.

Rsim employs a full-mapped invalidation-based directory cache-coherence protocol and can support either a modified, exclusive, shared, invalid (MESI) protocol or a modified, shared, invalid (MSI) protocol. Both protocols support cache-to-cache transfers for requests for lines another processor holds in modified state. Rsim supports three memory consistency models—sequential consistency, processor consistency, and release consistency—and optimizations specific to ILP processors for each model.

Applications interface

Rsim simulates applications compiled and linked for Sparc V9/Solaris using standard Sparc compilers and linkers at all optimization levels, with two exceptions. First, because Rsim models a 32-bit architecture, it does not support 64-bit integer and quad-precision floating-point operations. Second, Rsim's trap convention differs from that of Solaris, so it cannot use standard libraries and applications that rely on such traps. Instead, an Rsim applications library supports commonly used libraries and functions.

There are some unsupported traps and related functions, and we have only tested our library for C application programs. Further, all system calls are only emulated, not simulated. Rsim currently does not support dynamically linked libraries. For multiprocessor applications, the Rsim library includes support for synchronization with locks, flags, and barriers through parmacs macros.

Statistics

Rsim provides various execution statistics. For many metrics, these include the average value, the standard deviation, and a histogram showing the distribution of the metric's values. Rsim also provides scripts that interface with a plotting utility to graphically display statistics related to a run or set of runs.

Overall performance. Rsim displays the total execution cycles and instructions per cycle that the program achieves on the simulated system. It further categorizes the execution cycles into processor busy cycles and stalls due to various instruction classes including arithmetic logic units (ALUs), floating-point units (FPUs), data reads, data writes, exceptions, branches, synchronization, and up to nine user-defined classes. Rsim splits data read and write stalls according to the level of the memory hierarchy that resolved the memory operation: L1 cache, L2 cache, local memory, or remote memory.

Other processor statistics. Rsim provides statistics on the usage of various functional units in the processor, branch prediction behavior, and instruction window occupancy.

Cache, memory, and network statistics. Rsim classifies memory operations into hits and misses, and further classifies misses into cold, capacity, conflict, and coherence. It also collects the average latency of various classes of memory operations, MSHR occupancy, prefetch effectiveness, bus utilization, write-buffer utilization, network contention, traffic, and network switch buffer usage.

Implementation

Rsim interprets application executables rather than uses traces, enabling more accurate modeling of the effects of contention and synchronization in multiprocessor simulations as well as speculation in multiprocessor and uniprocessor simulations. For speed and portability, it converts the Sparc V9 instructions into an expanded, loosely encoded instruction set format and internally caches them.

Key Rsim subsystems include the

- out-of-order processor scheduling engine,
- processor memory unit,
- cache hierarchy,
- memory/directory module, and
- interconnection network.

Each of these acts as a largely independent block, interacting with the other units through a small number of predefined mechanisms.

Internally, Rsim is a discrete event-driven simulator based on RPPT's Yacsim (Yet Another C Simulator) library. The various events model the processor pipelines, cache and memory system, and network, including contention at all resources. Most Rsim subsystems are activated as separate events only to perform work; the only exceptions are the processor and cache events, which are activated each cycle.

We wrote Rsim in modular fashion using C++ and C for extensibility and portability. We have thus far tested it on Sun systems running Solaris (up to version 2.8), a Hewlett-Packard Convex Exemplar running HP-UX version 10, an SGI Power Challenge running IRIX 6.2, and x86 systems running Linux. Porting Rsim from an initial Sun version to other big-endian systems such as the HP PA-RISC and SGI systems was straightforward, but porting it to little-endian systems such as x86 and Alpha took more effort.

The clearly defined subsystems and their interfaces enable significant extensions to Rsim through modification of a limited portion of the code. Most changes to a subsystem affect only that subsystem, easing debugging.

Rsim provides execution statistics including the average, standard deviation, and distribution of a metric's values.

IMPORTANCE OF MODELING ILP FEATURES

Many shared-memory multiprocessor simulation studies use simple processor models such as in-order issue, single-instruction issue per cycle, blocking-demand read misses, and no speculative execution. To model ILP's benefits, researchers typically speed up the simulated simple processor's clock rate and primary cache access time by a *clock multiplier* factor *N*, which can range from 1 to the issue width of the ILP processor modeled. We call such simulators simple-*N*x and have shown that such simple approximations are currently inadequate.

Clock multiplier

N is a measure of the average computational parallelism that a real processor can extract from an application. It also controls the average rate at which the processor sends requests to the memory system. The appropriate value of *N* depends on both the application and the system, and currently no known technique can determine *N* a priori. Educated guesses made on a case-by-case basis would require validation on a real machine, which may not exist, or on a detailed simulator, which defeats the purpose of using simple-*N*x simulators.

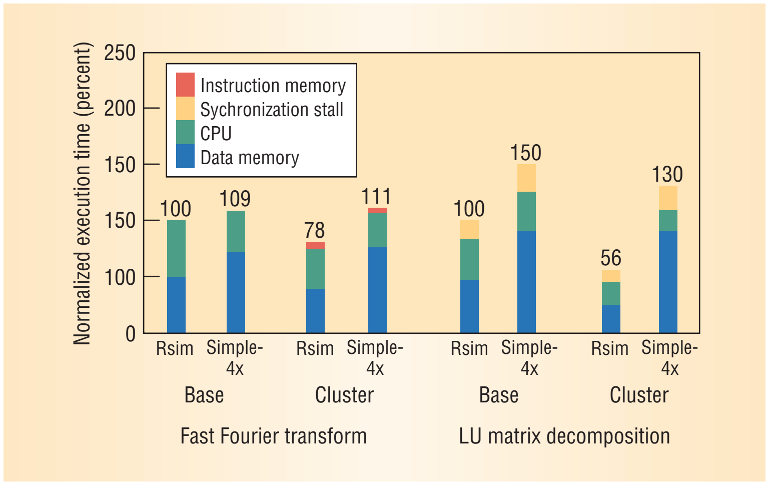


Figure 3. Execution times predicted by Rsim and simple-4x for an application with and without clustering, normalized to the Rsim time without clustering.

Read miss parallelism

The inability to model multiple outstanding read misses, or *read miss parallelism,* distinguishes simple and detailed simulators for most studies. Simple-*N*x simulators can model the benefits of overlapped computation with an appropriate *N*, but in applications that exploit hardware support for parallel read misses they can potentially produce large and unpredictable errors.[3] Our previous work showed that as processors become more aggressive, modeling read miss parallelism becomes increasingly important.[4]

To determine the extent of read miss parallelism, we experimented with several applications from the Stanford Parallel Applications for Shared Memory suite (Splash-2); http://www-flash.stanford.edu/apps/SPLASH/). Surprisingly, many original applications exhibited limited read miss parallelism with realistic instruction window sizes.[3] Because these applications were written to exploit spatial locality, contiguous sequences of instructions present together in a hardware instruction window worked on elements of a small number of cache lines. This limited the number of distinct overlapping cache misses, underutilizing the hardware resources for parallel read misses.

Read miss clustering. To increase read miss parallelism without sacrificing spatial locality, we proposed a compiler optimization, *read miss clustering*, that provides significant performance benefits by reordering instructions to pack independent read misses together within the same instruction window. Here, we report performance predictions from Rsim and simple-*N*x for two applications, fast Fourier transform (FFT) and LU matrix decomposition, with and without read miss clustering. The results are for a multiprocessor configuration similar to that in our prior work.[5] Results for a uniprocessor are qualitatively similar. Because simple-*N*x generally overestimates execution time, to maximize its advantage we chose the largest reasonable value for *N*: the issue width of the modeled processor—4 in our case. To let simple-4x best approximate out-of-order instruction issue effects, we set both simulators' functional unit latencies equal to one cycle.

Execution time predictions. Figure 3 presents the execution time predicted by Rsim and simple-4x on the application's original (base) and clustered versions, normalized to the Rsim prediction for the base version. Following previous work,[3] we categorized the execution times into four components:

- data memory stall time,
- CPU time,
- synchronization stall time, and
- instruction memory stall time.

For the base FFT, simple-4x matches Rsim well in predicting total execution time, but it underpredicts CPU time because the 4x clock multiplier is too aggressive. It overpredicts memory time because it does not model FFT's read miss parallelism. The under- and overpredictions balance out, making the overall time close to that of Rsim.

For the clustered FFT, simple-4x overpredicts execution time by 42 percent relative to Rsim because it cannot model the (higher) read miss parallelism. The clustering transformation also reduces CPU time through scalar replacement,[5] resulting in a closer match between the CPU times predicted by the two simulators. Thus, simple-4x's underpredicted CPU time no longer balances its memory time overprediction, resulting in a large overprediction of total time relative to Rsim.

For LU, simple-4x overpredicts execution time relative to Rsim by 50 percent for the base version and 132 percent for the clustered version—again, because both LU versions contain read miss parallelism, which simple-4x does not model.

Interpreting results. In addition to causing significant differences in predicted absolute execution times, simple-4x's inability to model read miss parallelism can lead to different conclusions about the effectiveness of memory-system-related optimizations. Figure 4 illustrates the reduction in execution time or speedup from read miss clustering, as predicted by Rsim and simple-4x and as seen on a Convex Exemplar's symmetric multiprocessor (SMP) hypernode. Rsim reported a speedup of 22 percent for FFT and 44 percent for LU, with a significant part from the data memory component. These results make a strong case for clustering optimization. In contrast, simple-4x reported a *slowdown* for FFT and only a 13 percent speedup—all from the CPU component from scalar replacement—for LU. These simple-4x results fail to make a strong enough case to implement clustering optimization in a compiler.

Although there are significant differences between the Exemplar and the Rsim model, we measured the benefits of clustering on the real machine to ensure that our results are not simply an artifact of our simulation models. Figure 4 shows that the Exemplar also experiences significant benefits from clustering. The benefits are less than with Rsim because its CC-Numa system has a higher bandwidth than the Exemplar's SMP hypernode. Nevertheless, the results illustrate simulators' ability—or lack thereof—to capture phenomena that are important on real hardware.

Prefetching

Prefetching also can improve memory parallelism in a program. Simple-*N*x simulators can model prefetches and miss parallelism because they do not return values to the processor or block instruction retirement. Simple-*N*x simulators can potentially predict execution times well for application versions that use prefetches to exploit all available memory parallelism.

Our recent work examining the interaction between clustering and prefetching[6] found that clustering alone often outperformed prefetching alone, but combining the two techniques achieved the best performance. Comparing simple-4x, Rsim, and the Exemplar for FFT and LU with and without both prefetching and clustering, we again found that simple-*N*x is close to Rsim in some, but not all, cases. Interestingly, prefetching degraded performance on all LU versions on the Exemplar, underscoring the difficulty of relying on this technique to justify using simple-*N*x simulators.

Other ILP features

Some multiprocessor architecture research inherently requires modeling aggressive ILP features. For example, researchers have proposed using speculation and out-of-order issue to remove the performance gap between strict and relaxed memory consistency models.[1] It is not clear how to use simple-*N*x simulators to determine if such techniques are successful because simple-*N*x does not model the features that the techniques exploit. Using real machines for this kind of study is also difficult—we are not aware of any that are similar in all respects other than their consistency model implementations.

As processors exploit ILP more aggressively, it is also possible that other effects such as the impact of memory references generated from mispredicted paths will increase in importance.

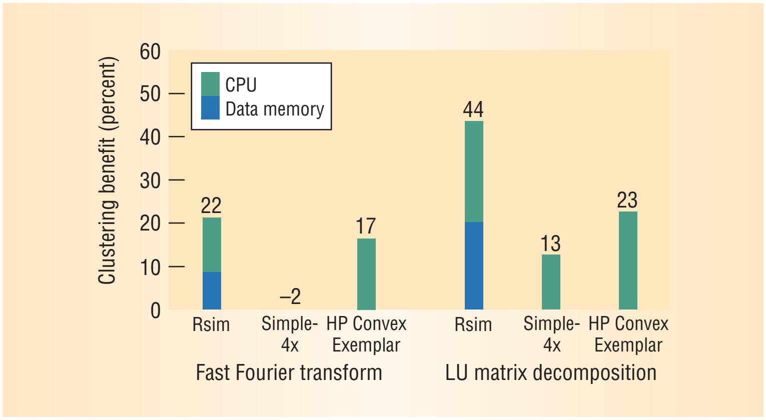


Figure 4. The reduction in execution time with read miss clustering predicted by Rsim and simple-4x, as seen on a Convex Exemplar's symmetric multiprocessor hypernode.

Reconciling results: The Flash study

A recent study by Jeffrey S. Gibson and colleagues[7] that validated various simulators against the Flexible Architecture for Shared Memory (Stanford Flash) machine (http://www.flash.stanford.edu/), which uses MIPS R10000 processors, concluded that a detailed simulator with capabilities similar to Rsim is no more accurate than simple-*N*x simulators with an appropriate *N* value.

The Flash study used four original Splash-2 codes enhanced with prefetching, including FFT and LU. Most original Splash-2 codes have little read miss parallelism, and prefetching can capture some of what is available. However, our work shows that modeling detailed ILP features becomes increasingly important with read miss parallelism—for example, with read miss clustering. Further, because applications are expected to perform best when transformed to increase read miss parallelism and run on modern ILP processors, considering these forms of the applications is important. Simple-*N*x cannot model the benefits of read miss parallelism in such applications.

Consistent with our work, the Flash study found that the clock multiplier is a key factor for simple-*N*x simulators and that determining an appropriate *N* value a priori is difficult. The study used three different values for *N* and showed significant differences in predicted execution times for each value. For example, too large a value for FFT resulted in significant underpredictions of multiprocessor speedups. The final conclusion about the adequacy of simple-*N*x simulators was based on results obtained using the best of the three values studied for *N*.

Finally, the Flash study did not consider research on optimizations that directly make use of a processor's aggressive ILP features—for example, using speculation for strict consistency models.

CHALLENGES

Besides the overall effort required to develop such a large software system, we faced two key challenges in designing and using Rsim: a relatively slow simulation speed and lack of real hardware for validation.

Table 1. Rsim performance in normal and Rabbit mode, with and without cache simulation.

Application	Description	Instructions (millions)	Execution speed (application instructions per second)			Slowdown versus native		
			Normal mode	Rabbit mode		Normal mode	Rabbit mode	
				Cache simulated	No cache simulated		Cache simulated	No cache simulated
Minimum spanning tree	Pointer-intensive	206	27K	33K	3,070K	4,300	3,400	37
LU matrix decomposition	Scientific	484	24K	70K	11,100K	7,100	2,400	15
MPEG-2 encode	Multimedia	1,070	28K	2,900K	10,000K	15,300	150	43

Simulation speed

Rsim's detail comes at the cost of simulation speed, a limitation shared by most detailed simulators. Although we could design the simulator efficiently enough to perform substantive architecture studies, it is about an order of magnitude slower than simple-processor-based multiprocessor simulators.[4]

To overcome part of the speed penalty, we enhanced Rsim with Rabbit, a fast functional simulator. We modeled Rabbit after Embra, a fast binary-translation-based simulator that SimOS uses to fast forward workloads to interesting points and to checkpoint state.[8] In Rabbit mode, Rsim accelerates initialization and other portions of the code when collecting timing statistics is unimportant. It uses binary translation to let the host processor run sections of the code natively, invoking simple cache simulation as a user-specified option.

Table 1 gives some performance results for Rsim in normal and Rabbit mode on a Sun Ultra 5 with a 400-MHz UltraSparc II for applications representing pointer-intensive, scientific, and multimedia codes. All runs simulated a 1-GHz, out-of-order uniprocessor with an issue width of 4, and 64-Kbyte L1, and 1-Mbyte L2 caches. In normal mode, Rsim's speed is relatively constant across different classes of applications, averaging about 26,000 instructions simulated per second. The slowdown over native execution is more varied—4,300 times to 15,300 times, with an average of 8,900 times—and depends on how fast the host machine executes the applications in native mode; it is lower for more memory-intensive applications.

Rabbit mode without cache simulation is on average 310 times faster than normal mode, with only a 32 times slowdown from native execution; with cache simulation, Rabbit mode is 40 times faster. Rabbit's speed varies depending on the types of instructions in the simulated application. With cache simulation, the speed varies with the number of cache misses as Rsim enters normal mode to handle such misses.

Open questions remain about how to use Rabbit effectively for sampling on general workloads, particularly for multiprocessor systems. Also, its current implementation is not as highly optimized as Embra.[8] Likewise there are questions about how to integrate other complementary speed-enhancing approaches such as direct-execution for uniprocessors[9] and multiprocessors[4] and statistical simulation for uniprocessors.[10] Overall, improving ILP simulation speed without losing accuracy remains an important object of research.

Validation

Another challenge at the time we started developing Rsim was the lack of real hardware to guide the design. Consequently, Rsim does not model any one real system and lacks validation against existing hardware. As the "Performing Valid Studies with Unvalidated Simulators" sidebar indicates, this is a pervasive problem in computer architecture research as the complexity of designing real hardware leads to an increasing reliance on simulation.

We made some deliberate modeling abstractions in Rsim's design to reduce programming complexity or time, or simply because we did not have detailed information. Newer internal versions have more realistic models, but some abstractions remain—for example, the memory model and parts of the processor pipeline. Current high-frequency ILP processors now face constraints not seen in the then state-of-the-art MIPS R10000, and the Rsim processor does not model the implementation decisions due to those constraints.

Validating Rsim on the basis of absolute execution time was difficult when aggressive ILP uniprocessors and multiprocessors became com-

Performing Valid Studies with Unvalidated Simulators

Two recent studies concluded that execution times predicted from simulation deviate significantly from real hardware results and that tuning the simulators to match the hardware requires considerable effort.

Jeffrey S. Gibson and colleagues used their results to underscore the importance of building real hardware as a final validation of architectural ideas,[1] but expecting all research groups to build hardware is unfeasible given the increasing complexity of current systems. Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler[2] advocate that simulation studies use community-supported sets of parameters and compare results against a reference hardware platform or validated simulator. They also recommend that experimental studies be reproducible and demonstrate that optimization benefits are *stable*—demonstrable on a range of architectural parameters.

This is excellent guidance when real hardware is too expensive to build. However, we believe that these recommendations set an impractically high and sometimes insufficient standard for much research, and that simulators such as Rsim can continue to provide valuable insights into architectural phenomena and relative trends.

Building validated simulators

For studies based on significantly new architecture paradigms, validating simulators is difficult because no base hardware exists. For research based on local changes to existing architectures, building base architecture simulators and validating them against a reference machine is conceptually feasible, but much system information is proprietary. For example, Desikan and colleagues could not validate their simulator's memory system because it lacked documentation. Although they achieved remarkable accuracy validating the processor model with microbenchmarks, significant and widely varying errors occurred in the execution times predicted for the full applications—up to 43 percent, with a mean of 18 percent and coefficient of variance of 73 percent.

Selecting reference hardware

Even if building an open source simulator that validates well against reference hardware were possible, choosing such hardware would be controversial even for research on localized optimizations for three reasons:

- Technology evolves very quickly. By the time the reference hardware appropriate for the research horizon of a specific project becomes available and is validated, a new architecture may become the desired reference point.
- There is often no clear consensus on the ideal reference architecture. For example, there is currently no consensus on the architecture of either the future high-performance or embedded system.
- Most machines have one or more performance-inhibiting features that in hindsight are considered incorrect decisions; conversely, every machine has relatively aggressive performance-enhancing features that are not implemented in other contemporary systems. A broad scientific evaluation requires considering the impact of the proposed research independent of machine-specific constraints.

Choosing the right workload

Regardless of the simulator's accuracy, experimental results also depend on the workload choice. Workloads evolve as architectures change, but standard benchmark suites typically used in architectural studies evolve much more slowly. Having only an accurate simulator is not enough—we must interpret results from such a simulator on nonrepresentative workloads with as much care as the results of an unvalidated simulator.

Emphasizing analysis

Following previous studies, we support efforts to validate simulators and make them as close to real hardware as possible. Even with a validated simulator, however, understanding the proposed ideas' impact beyond the specific machine and workload studied is important. Architects will therefore inevitably continue to use a multitude of unvalidated simulators and previous-generation workloads to evaluate their ideas.

For such studies to remain relevant, researchers should emphasize analysis to determine the architectural and workload characteristics under which a proposed technique is expected to provide both a performance benefit and loss. The insights obtained from such analyses will likely have more relevance than absolute performance improvement numbers. Practitioners can use both analytic insights and simulation data to determine whether a proposed technique is worth further exploration in their specific environment.

References

1. J. Gibson et al., "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 00), ACM Press, New York, 2000, pp. 49-58.
2. R. Desikan, D. Burger, and S.W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM Press, New York, 2001, pp. 266-277.

mercially available. We configured our model to match an R10000 and UltraSparc-II as much as possible and used microbenchmarks to compare the execution times that Rsim predicted to real times on each of these machines. In many cases, Rsim closely matched one of the processors but not the other, and in some cases it did not match either processor. This was not unexpected because the MIPS R10000 has a different instruction set, the UltraSparc-II has a different core pipeline, and both processors have different memory hierarchies than Rsim.

Our results conclusively indicated that modeling aggressive features of ILP processors is important even for multiprocessor studies.

Nevertheless, we have validated *relative* performance benefits predicted by our work with Rsim. Although the Convex Exemplar and Rsim have different architectures, when comparing the benefits of read miss clustering and prefetching, the simulator predicted the same overall phenomena and trends that we observed on the real machine.[5,6]

Further, to increase confidence in our results, we have focused on the insights obtained from our results rather than the absolute numbers themselves. We have also correlated insights obtained with detailed behavior of the hardware and specific application patterns to help identify and fix errors in our simulator that may have occurred due to inadvertent bugs, poor abstraction of the real system, or omissions of real system constraints.

EXPERIENCE

We have used Rsim extensively in multiprocessor and uniprocessor research on scientific, database, and multimedia workloads.

We initially theorized that aggressive ILP processors' ability to overlap read miss latency would make synchronization time relatively more important, but our first experiments quickly showed this to be incorrect. ILP features were effective in speeding up the CPU component but not in addressing data memory stall times, which continue to be a dominant performance bottleneck.[3]

However, our results conclusively indicated that modeling aggressive features of ILP processors is important even for multiprocessor studies. Previous-generation simple-processor simulators often led to erroneous and inconsistent results. We therefore initiated a broader research study on the impact of ILP processors on shared-memory systems, concentrating on performance, programmability, and evaluation techniques. This work led to several performance-enhancing and evaluation techniques targeted at ILP-based systems.

Detailed ILP simulators are clearly necessary for multiprocessor studies in which aggressive ILP features play a direct role, such as evaluating speculation-based memory consistency implementations.[1] For other studies, modeling the processor to the extent that it generates memory references at the correct rate—determined by the amount of ILP in the computation and the amount of memory parallelism—is important. Currently, no known techniques can correctly model these effects in a simple-processor-based simulator without knowledge of application behavior determined from detailed simulation or real hardware. As processors exploit increasing amounts of ILP, these effects will become increasingly relevant.[4] Other effects such as impact of memory references generated from mispredicted paths may also become more important.

As microprocessor systems become more complex, we believe that the availability of shared infrastructure source code will become increasingly crucial. Since we made Rsim version 1.0 publicly available, several groups worldwide have used it for research and education. While distributing and supporting Rsim has required a significant time commitment, the experience has been very positive. We have used feedback from other research groups to identify bugs in the simulator and further refine its model.

We plan to release a new version shortly that will include instruction caches, TLBs, multimedia extensions, simultaneous multithreading, Rabbit fast simulation mode, and ports to Linux platforms. A current limitation is that Rsim does not support full system simulation—specifically the effects of system calls, I/O, and virtual memory. Simulators such as SimOS (http://simos.stanford.edu/) and Simics (http://www.simics.com) have the ability to run an unmodified operating system, making them easier to use on workloads that use OS features extensively. Future editions will address this limitation as well as improve performance and ease of use. ■

Acknowledgments

The Rsim project began while the authors were at Rice University. It has been funded in part by an IBM Partnership Award; Intel Corp.; the National Science Foundation under grant nos. CCR-9410457, CCR-9502500, CDA-9502791, CDA-9617383, CCR-0096126, and EIA-0103645; Rice University; the Texas Advanced Technology Program under grant nos. 003604016 and 003604-025; and the University of Illinois. Christopher J. Hughes has been supported by a Texas Instruments Fellowship and a Richard T. Cheng Fellowship, Vijay S. Pai by a Fannie and John Hertz Foundation Fellowship, Parthasarathy Ranganathan by a Lodieska Stockbridge Vaughan Fellowship, and Sarita V. Adve by an Alfred P. Sloan Research Fellowship. We thank Hazim Abdel-Shafi, Murthy Durbhakula, Dennis Geels, Jonathan Hall, Tracy Harton, and Rohit Jain for assistance in various stages of Rsim's development.

References

1. K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. Int'l Conf. Parallel Processing* (ICPP 91), vol. I, CRC Press, Boca Raton, Fla., 1991, pp. 355-364.
2. R.G. Covington et al., "The Efficient Simulation of Parallel Computer Systems," *Int'l J. Computer Simulation*, Jan. 1991, pp. 31-58.
3. V.S. Pai, P. Ranganathan, and S.V. Adve, "The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology," *Proc. 3rd IEEE Symp. High-Performance Computer Architecture* (HPCA 97), IEEE CS Press, Los Alamitos, Calif., 1997, pp. 72-83.
4. M. Durbhakula, V.S. Pai, and S.V. Adve, "Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors," *Proc. 5th Int'l Symp. High-Performance Computer Architecture* (HPCA 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 23-32.
5. V.S. Pai and S.V. Adve, "Code Transformations to Improve Memory Parallelism," *Proc. 32nd Ann. Int'l Symp. Microarchitecture*, (MICRO 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 147-155.
6. V.S. Pai and S.V. Adve, "Comparing and Combining Read Miss Clustering and Software Prefetching," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 292-303.
7. J. Gibson et al., "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 00), ACM Press, New York, 2000, pp. 49-58.
8. M. Rosenblum et al., "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. Modeling and Computer Simulation,* vol. 7, no. 1, 1997, pp. 78-103.
9. E. Schnarr and J. Larus, "Fast Out-of-Order Processor Simulation Using Memoization," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 98), ACM Press, New York, 1998, pp. 283-294.
10. M. Oskin, F.T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM Press, New York, 2000, pp. 71-82.

Christopher J. Hughes *is a graduate student in the Computer Science Department at the University of Illinois at Urbana-Champaign. He is currently researching adaptive architectures for multimedia applications. Hughes received an MS in computer science from the University of Illinois at Urbana-Champaign. He is a member of the ACM and the IEEE. Contact him at cjhughes@cs.uiuc.edu.*

Vijay S. Pai *is an assistant professor in the Department of Electrical and Computer Engineering as well as the Department of Computer Science at Rice University. His research interests include computer architecture, high-performance networking, and performance evaluation. Pai received a PhD in electrical and computer engineering from Rice University. Contact him at vijaypai@rice.edu.*

Parthasarathy Ranganathan *is a researcher at Compaq Western Research Laboratory in Palo Alto, Calif. His research interests include low-power system design, computer architecture, and performance evaluation. Ranganathan received a PhD in electrical and computer engineering from Rice University. He is a member of the ACM and the IEEE. Contact him at Partha.Ranganathan@compaq.com.*

Sarita V. Adve *is an associate professor in the Computer Science Department at the University of Illinois at Urbana-Champaign. She directed the Rsim project while at Rice University. Her research interests are in computer architecture and performance evaluation methods. Adve received a PhD in computer science from the University of Wisconsin-Madison. She is a member of the ACM and the IEEE. Contact her at sadve@cs.uiuc.edu.*