PERFORMANCE ANALYSIS AND VALIDATION OF THE PICOJAVA PROCESSOR

THIS DEVELOPMENT TEAM USED SIMULATOR CHECKPOINTS TO SPEED RTL MODEL SIMULATION, A SIMPLE AND EFFECTIVE METHOD THAT ACCURATELY ANALYZES PERFORMANCE AND VALIDATES DESIGN.

Sudheendra Hangal and Mike O'Connor
Sun Microsystems, Inc.

In the process of designing high-performance microprocessors, architects and designers build processor models at varying levels of abstraction for a range of purposes. Examples of such uses are simulating and verifying the processor, predicting processor performance, and developing operating systems and applications for the processor before silicon is available. Such models can range in complexity from simple analytical performance models in spreadsheet format to the detailed design expressed in a hardware description language (HDL). Other models that fall in between on the range of complexity include instruction-accurate simulators, trace-driven performance simulators,[1,2] and cycle-accurate simulators.[3] There is usually a trade-off between runtime performance and accuracy while using these models.

Developing a large number of models for a processor is an expensive proposition. Beyond the engineering resources to build the initial models, developers must verify each of the models for correctness against the design specification; they must also maintain and update the models as the design evolves. Clearly, a strategy that limits the number of models yet meets the requirements of early software developers, verification engineers, and performance analysts is desirable.

During the course of the picoJava processor[4] design, we developed a simple and effective methodology that meets the goals of both accurate performance analysis and design validation. Although we applied this technique to the picoJava processor, the technique can be applied to other processor designs.

A typical design and validation methodology

To design the core, picoJava designers described the chip's functionality and implementation in Verilog, a popular HDL for logic design, and coded it in register-transfer level (RTL) form. RTL code specifies the detailed implementation of the chip at the level of such logic blocks as registers, comparators, and multiplexers. The designers then simulated and debugged their blocks on a Verilog simulator running on desktop workstations.

In the early stages of the picoJava project, we also built an instruction-accurate simulator for the picoJava processor. It accurately modeled the architectural features of the processor, such as instructions, registers, interrupts, traps, and caches, but did not model pipeline detail or timing issues.

Modeling the picoJava architecture at a higher level of abstraction in software allowed us to develop this simulator in a short time. Our operating system and tools groups began their development and debugging on the instruction-accurate model of the processor

0272-1732/99/$10.00 © 1999 IEEE

long before completion of the picoJava design. Verification engineers also used the instruction-accurate model to develop and debug more than 2,000 diagnostic tests that exercised every part of the design, including tests for unusual corner cases.

The instruction-accurate model also served as a "golden" model for verifying the RTL functionality. Our validation environment included a cosimulation mode, in which we started the instruction-accurate and RTL simulators together. The environment then stepped both simulators through program instructions, one at a time. At the end of every instruction, it compared the architectural states of the processor between both models and, if they were not identical, flagged an error. When the program stopped, the environment compared other states, such as memory contents.

We had two independent models of the same specification, implemented at different levels of abstraction using different methodologies and languages, and developed by different teams, which meant that both models would rarely have the same bug. The cosimulation mode also allowed us to validate the RTL model using programs generated by a pseudorandom test generator. Pseudorandom test case generation is a very popular method for validating processor functionality, since it can exercise the processor design on unusual instruction sequences not often encountered in manually written test programs.[5,6]

Cycle-accurate simulation

Before silicon that was based on the picoJava-II core became available, we wanted to evaluate the performance of Java and C code on the core. However, we could not use an instruction-accurate simulator for performance analysis because it did not model the timing of every operation.

In the past, architects have used cycle-accurate simulators for this purpose and sometimes for validation as well.[3] Cycle-accurate simulators typically run into tens of thousands of lines of code and are much more complex to develop and debug than instruction-accurate simulators. They often require several person-years of work. Cycle-accurate models also have to be developed and verified independently for each new implementation of the architecture, while instruction-accurate simulators typically undergo only minor change between implementations of the architecture.

Cycle-accurate simulators generally run at much lower speed than instruction-accurate simulators because of the level of detail at which they simulate the processor. Therefore, it is impractical to run entire benchmarks on the cycle-accurate model. Performance analyses often resort to sampling the benchmark trace and use these samples to compute processor performance.[7] An important advantage of cycle-accurate simulators is that they permit "what-if" analysis because the performance behavior of the chip can be modified fairly quickly.

> Clearly, a strategy that limits the number of models yet meets the requirements of early software developers, verification engineers, and performance analysts is desirable.

The following steps describe the traditional way of analyzing performance with a cycle-accurate simulator:

1. Run the complete benchmark on a fast instruction-accurate model.
2. Generate traces from the instruction-accurate model as it is running, starting at regular or random points during program execution.
3. Run each trace on the cycle-accurate model, which simulates the processor as it runs the program trace, clock for clock. Since an instruction trace is available from the instruction-accurate model, cycle-accurate simulators often simulate only the execution time of the program but not the functionality of the instructions. Since they are independent, individual traces can be run in parallel on the cycle-accurate model.
4. Compute the average cycles per instruction (CPI) over all the traces, then use this CPI and the total instruction count to compute total program time.

RTL simulation

One approach to performance analysis is to run benchmark programs using the RTL model. However, conventional RTL simulators are very slow since the design is simulated at a great level of detail. Moreover, the speed

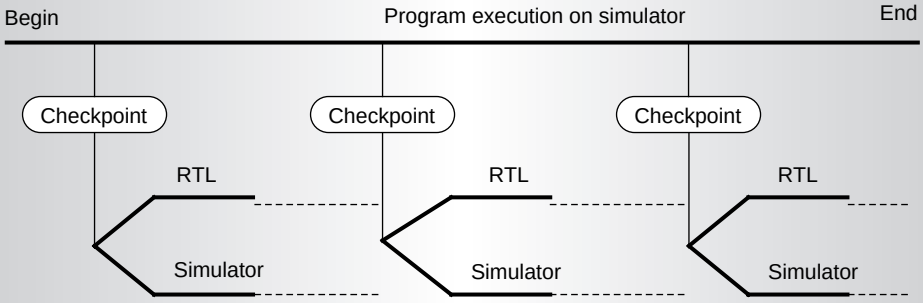

Figure 1. Restarting checkpoints from a long-running program on both the RTL and the simulator.

Simulation trade-offs

Table A illustrates the trade-offs in using different processor models and simulation techniques during processor design. Instruction-accurate simulators are used throughout the project as reference models, for software development and for cosimulation. Cycle-accurate simulators help architects predict performance, and carry out what-if analysis while the design is in progress. Designers develop and debug all their RTL using software simulators. During the final stages of the project, when the design is relatively stable, gate-level models of the chip may be run on simulation accelerators or emulators to debug long-running programs such as operating systems. Simulation speeds listed in Table A are approximate, but relative speeds of the simulators are representative.

Table A. Typical simulation models and techniques in processor designs.

Type of simulator	Typical simulation speed (cycles/sec)	Features
Instruction-accurate simulator	400,000	Fast simulation, easy to implement, but generates no timing information.
Cycle-accurate simulator	20,000	Detailed timing simulation with what-if analysis. Complex to implement and maintain as the RTL changes.
Software-based Verilog simulator	100	100%-accurate model of the design RTL with an easy design and debugging environment as well as accurate timing information. However, simulation speed is slow.
Simulation accelerator	10,000	Includes dedicated hardware to accelerate circuit simulation algorithms at the gate level. Fast but expensive; requires porting to a proprietary environment.
Emulator	500,000	Maps gate-level model into FPGAs, even faster and even more expensive; requires porting to a proprietary environment.

of simulation often depends on the design size, the amount of activity in the design, and the coding style. This prevents running nontrivial benchmarks on the RTL model using conventional simulators.

Alternatively, RTL simulations can be run using fast simulation machines built explicitly for simulating digital circuits.[6] A common use for such machines is to validate the RTL by booting the operating systems on an RTL model before tape-out. Booting an operating system is an important task because there is no substitute to running real-world code on the RTL model before tape-out to ensure the design is functionally correct. However, using such simulation machines is often a complex task, requiring porting of the simulation support to a proprietary, vendor environment, and reverifying the design to ensure that simulator differences do not cause different behavior under the new environment. Using these machines, one can expose bugs by running simulations that would take too long to run on software simulators. Table A in the adjacent box lists some of the pros and cons for various simulation techniques.

RTL parallel simulation

In view of the problems associated with developing and maintaining cycle-accurate simulators, we wanted an automatic way of using the RTL model itself for modeling the processor's performance. To enable this, we began by running the entire program on the instruction-accurate simulator and dumping the simulator's state to a checkpoint file at selected points during the execution.

Conventionally, checkpoints are implemented in a system to restart the same system from the same state at a later time. However, we implemented the capability to restart not

only the instruction-accurate simulator but also the RTL model from the checkpointed state. When we restart the RTL model from the checkpoint, it can run the program from that point on as if it had executed the program from the beginning. An important advantage of this methodology is that once the checkpoints have been generated, the RTL can be restarted from all checkpoints in parallel.

Figure 1 illustrates how a long-running program is run on the instruction-accurate simulator, generating multiple checkpoints. The RTL model is restarted from each checkpointed state. A restarted run on the RTL may continue up to the position of the next checkpoint or terminate after running for a fixed number of instructions.

Note that once the RTL is restarted from a checkpoint, it may be run in cosimulation mode, which allows us to catch bugs in the RTL during the restarted runs. Therefore, our methodology lets us perform verification at any stage of long-running programs without waiting for the RTL to get to that state. It also provides verification value for software, since software is exercised on a fully accurate model of the chip.

For performance analysis, we generated checkpoints at randomly selected points during execution of a benchmark. We then restart the RTL from each checkpoint and execute a fixed number of instructions. For validation of long-running programs, we usually generate checkpoints at regular intervals, restart the RTL model from each checkpoint, and have it execute instructions until just beyond the next checkpoint.

The checkpointed state from the simulator contains the architectural state of execution, and not the precise machine state. For example, information on pipeline stages or instruction-fetch queue status is not part of the checkpointed state because the instruction-accurate simulator does not model these entities. As a result, when the RTL restarts from a checkpoint, it starts with an empty pipeline, empty load-store queues, and empty write buffers. However, larger entities like caches are part of the checkpointed state since these are modeled by the simulator.

The processor structures not initialized using the checkpointed state are only a few entries deep and warm up within a few hundred instructions. Since we typically execute a checkpoint for 250,000 to 500,000 instructions, we discount the minor inaccuracies in the first few instructions. In general, for accurate performance analysis using this methodology, the instruction-accurate simulator must model all processor structures that significantly impact performance or take a long time to warm up. Alternatively, performance data collection can be started only after the RTL model has run long enough to warm up all these structures.

1. Run the program on the instruction-accurate model. At selected points during the program execution, generate checkpoint files. Also, compute a total instruction count. Now perform steps 2 to 6 for every checkpoint file.
2. Convert the simulator checkpoint file into memory initialization files for the RTL. In our implementation, we first stored checkpoint states that could be restored by software running on the processor into a preallocated buffer in memory.
3. Start the RTL simulation in cosimulation mode, initializing the RTL memories from the files generated in step 2. Since we ran the simulator and the RTL in lockstep, we also initialized the equivalent memory elements in the simulator.
4. Start execution on the simulator and the RTL by running a reset stub routine that sets up the rest of the architectural state such as registers, reading values from the preallocated buffer in memory as required. Caches are off at this stage.
5. Enable caches and switch to the point in the program where the checkpoint was taken by jumping to the next instruction after the checkpoint.
6. Run the simulation for a predetermined number of instructions or until the beginning of the next checkpoint.
7. Compute the CPI of the RTL model over the instructions executed for all checkpoints, eliminating the start-up overhead for each run. Multiply the CPI by the total number of instructions generated in step 1 to arrive at the overall execution time.

Figure 2. Steps for restarting the RTL from a simulator checkpoint.

To restart the RTL model from a checkpoint, we let the RTL chip model come out of reset and execute a reset stub routine, which initializes the registers to their checkpointed values and enables caches. Having the stub routine handle the architectural state that can be restored in software is easier than finding all the right places in the RTL code in which to initialize the appropriate state. However, the checkpointed state that cannot be initialized by software must be restored directly by the simulation environment. For the picoJava environment, this was straightforward and required adding only three commands to the simulation environment.

Figure 2 describes the steps for restarting the RTL from a simulator checkpoint.

The state of the processor, after restarting from a checkpoint, is not precisely the same as

Our methodology is particularly effective when a large group of machines is available for running simulations.

the state the processor would have reached had it run continuously up to the same point. However, this does not degrade the verification value of the methodology, since, in an actual system, asynchronous events like interrupts are likely to perturb the state of the processor at various points.

A major advantage of our methodology is that it requires only minimal development. The instruction-accurate simulator must support a transparent checkpoint-restart mechanism. Operating system and tool developers require this feature anyway because they must simulate programs that run on the simulator for hours or days. It makes for much more efficient testing and debugging if software developers can restart the simulator from some point during program execution instead of running it from reset each time. The only additional effort involved is in translating the simulator checkpoint files into a form that can be used to initialize the RTL.

Server pool

Our methodology is particularly effective when a large group of machines is available for running simulations. At Sun's Microelectronics site, we have a pool of about 1,200 server machines with over 2,000 CPUs. A software system manages the entire pool; users can submit jobs to one of several project queues. The software then takes over queueing, executing, and reporting results from the job.

This system is ideal for methodologies like ours, where a large number of checkpoints can be restarted on the RTL in parallel. If we use all 2,000 CPUs in our server pool to restart 2,000 checkpoints on the RTL, and each simulates 100 cycles per second, we have an effective simulation throughput of about 200,000 cycles/second. This is well within the range of hardware emulator speeds. Therefore, for the duration of these runs, we have effectively converted our server pool to a small, flexible emulation machine at no additional cost!

Results

We used this methodology to validate the processor by booting the JavaOS operating system on the RTL model before tape-out. We split the booting sequence into chunks of 250,000 instructions each and ran all of the RTL restarts in parallel. This process caught four bugs in the RTL and one bug in the operating system one week before scheduled tape-out. At that stage, the RTL was passing all other tests, but the operating system boot-up exposed bugs involving complex interactions between blocks or use of instruction sequences in modes that had not been heavily tested.

We estimate that if we had booted the operating system directly on the RTL, we would have hit the first of these bugs after running for 12 days on the RTL. After fixing the problem, we would have had to wait for an equal amount of time to verify the fix, the clock being reset to zero every time we found a bug or made an incorrect fix. Using the checkpoints, we could restart the RTL from the last checkpoint before the failure, and then quickly debug the problem and iterate over bug fixes. As a result, the total time from the first attempt until a successful operating system boot-up on the RTL was only four weeks, including the time taken to analyze and fix all the bugs. A successful operating system boot-up using the RTL simulator on a single workstation would have taken about three weeks by itself, even if no bugs had been found.

One of our goals was to predict the performance of the processor on the SPECjvm98 benchmarks[8] before tape-out. Using our methodology, we not only achieved that goal, but also derived feedback on benchmark profiles and located performance bottlenecks.

To predict performance on the SPECjvm98 suite, we ran samples of each of the seven SPECjvm98 benchmarks on the processor RTL using our methodology. A typical SPECjvm98 benchmark took approximately 10 billion instructions to complete. We attempted to simulate about 2% to 4% of the total benchmark, drawn from statistically sampled parts of the program, which amounted to

about 200 million to 400 million instructions. We typically split up the benchmark into approximately 800 checkpoints, each running about half-a-million instructions.

On a single CPU, this approach would take about two months for each benchmark. By taking advantage of perfectly parallel speedups on our server pool, and using only 10% of the pool (200 CPUs), each server needed to restart only four checkpoints on the RTL. The entire run lasted just six hours. Thus, if we started a benchmark job at night, the results would be ready for us by the next morning. The fast turnaround provided us ample opportunity to experiment with and tune our software.

Our predicted performance results were within 3% of the actual results from running benchmarks on the actual hardware when the silicon arrived. Part of the variation was due to software changes that took place between the time the simulations were run and delivery of the silicon. We therefore got highly accurate performance predictions as well as verification value in return for a modest investment in development effort.

Advantages and disadvantages

Using the RTL for performance analysis has the following advantages:

- It is a completely accurate model of the actual processor.
- As the RTL design evolves, its performance characteristics are tracked automatically. There is no need to maintain a cycle-accurate simulator along with the RTL and debug discrepancies between the two.
- It is possible to detect difficult-to-spot performance bugs (which cause the program to execute correctly, but affect its performance) on the RTL. For the picoJava core, we generated several statistics while running the benchmarks on the RTL; if one of them was outside the range that the architects expected, we could examine the problem further.
- It provides important verification value by allowing any part of long-running programs to be exercised on the RTL model.

The disadvantages are

- The RTL model is relatively inflexible to change, so quick what-if analyses by varying different architectural parameters is not very easy.
- The RTL model needs to be executing instructions and must be robust before performance runs can be made. This does not allow performance evaluation during the early stages of the design.
- The overall amount of compute resources required is higher compared to the resources for a cycle-accurate simulator.

> We used this methodology to validate the processor by booting the JavaOS operating system on the RTL model before tape-out.

This methodology accomplishes the twin goals of accurate performance analysis and validation of processor designs on real-world software simultaneously. For most processor design projects, implementing this methodology involves only minimal additional development effort.

Readers interested in obtaining more details about this methodology can download the source code for the picoJava processor RTL model and simulation environment from http://www.sun.com/microelectronics/communitysource. The source code is available under the terms of the Sun Community Source Licensing agreement. MICRO

Acknowledgments

We thank the members of the picoJava and microJava-701 project teams at Sun, especially David Skinner and Shing-Sheung Tse, for their cooperation and help in implementing this methodology.

References

1. M. Tremblay et al., "A Fast and Flexible Performance Simulator for Micro-Architecture Trade-off Analysis on UltraSPARC-I," *Proc. 32nd Design Automation Conf.,* ACM Press, N.Y., June 1995, pp. 2-6.

2. T. Diep et al., "Performance Evaluation of the PowerPC 620 Microarchitecture," *Proc. 22nd Int'l Symp. Computer Architecture,* May 1995, pp. 163-174.
3. G. Maturana et al., "Incas: A Cycle Accurate Model of UltraSPARC," *Proc. 1995 Int'l Conf. Computer Design,* IEEE Computer Society Press, Los Alamitos, Calif., Oct. 1995, pp. 130-135.
4. H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine for Java Bytecode," *Computer*, Oct. 1998, pp. 22-30.
5. M. Kantrowitz et al., "Functional Verification of a Multiple-Issue, Pipelined, Superscalar Alpha Processor—the Alpha 21164 CPU Chip," *Digital Tech. J.,* Vol. 7, No. 1, 1995, pp. 136-144.
6. J. Kumar et al., "Emulation verification of the Motorola 68060," *Proc. 1995 IEEE Conf. Computer Design,* Oct. 1995, pp. 150-158.
7. G. Lauterbach, "Accelerating Architectural Simulation by Parallel Execution of Trace Samples," Tech. Report SMLI TR-93-22, Sun Microsystems Laboratories, Inc., Dec. 1993; www.sunlabs.com/technical-reports/1993/abstract-22.html.
8. SPEC JVM98 Benchmarks: http://www.spec.org/osg/jvm98.

Sudheendra Hangal is a member of technical staff, working on Java processor designs at Sun Microsystems. His interests are in processor design and verification, compilers and parallel programming. Hangal has a Bachelor of Technology in computer science and engineering from the Indian Institute of Technology, Delhi, India.

Mike O'Connor is the principal architect of Sun's second-generation picoJava-II core and a key member of the team that designed the original picoJava-I core. His research interests include computer architecture, hardware-software codesign, and high-performance multimedia. O'Connor has a BSEE from Rice University and an MSEE from the University of Texas, Austin. He is a member of the IEEE Computer Society.

Direct comments about this article to Sudheendra Hangal at Sun Microsystems, Inc., 901 San Antonio Road, MS USUN03-311, Palo Alto, CA 94303; hangal@eng.sun.com.