Automatic Verification of Instruction Set Simulation Using Synchronized State Comparison

Technical Report No: ARCTiC-00-10 October, 2000

Bob Glamm
David J. Lilja

UNIVERSITY OF MINNESOTA
Laboratory for Advanced Research in Computing Technology and Compilers
Department of Electrical and Computer Engineering • 200 Union St. SE • Minneapolis • Minnesota • 55455 • USA
www.arctic.umn.edu

Automatic Verification of Instruction Set Simulation Using Synchronized State Comparison*

Bob Glamm and David J. Lilja
Department of Electrical Engineering
Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN 55455
{glamm,lilja}@ece.umn.edu

Abstract

Instruction-level simulation is the basis for much research in computer architecture. Typically, the simulators used for this type of research are verified by comparing the outputs of a simulated benchmark program and the outputs of the benchmark program run on a real machine - the simulator is "verified" if the outputs are similar. In the case of some benchmark programs, however, it is possible that significant fractions of the benchmark would not be executed due to minor differences or errors in the simulator, which would limit the usefulness of the results of the simulations.

In this paper, a novel method for verifying instruction-level simulators via step-by-step register state comparison to a hardware implementation is outlined. Subsequently, a description of a sample implementation of this verification method is presented, along with a discussion of implementation issues. Speed of the verification simulator is a concern (at 200 instructions per second on a 200MHz MIPS R10000), but possible ways to address this limitation are described.

*This work was supported in part by National Science Foundation grants CCR-9900605 and EIA-9971666, and by the Minnesota Supercomputing Institute.

1 Introduction

Simulation is an important part of computer architecture research. As fabrication costs of processors, ASICs, and systems continue to increase, building research prototypes becomes an increasingly expensive and time-consuming prospect. Simulation of processors and systems, on the other hand, is much less expensive, takes less time, and provides a more flexible framework for comparing design alternatives. Part of successfully using research results to guide the design of new systems depends (in part) on how closely the simulation matches the actual hardware.

An important first step in processor simulation is verifying that the simulated instructions perform as expected. This verification has typically been accomplished with a set of test vectors or by comparing program outputs. With test vectors, each vector is composed of an instruction, its inputs, and its expected outputs. The instruction is simulated with the provided inputs, then the simulated outputs are compared to the expected outputs. Any mismatch means that the instruction was simulated incorrectly. This process is tedious, though, as each test vector needs to be generated and checked by hand (although there has been some automation of this process for hardware verification [6]).

Program output testing is done by comparing the overall output of a simulated program to the output of a program run on actual hardware, for ex-

ample, comparing outputs produced when executing a standard set of benchmark programs, such as the SPEC CPU benchmarks [15], [12]. While this method is faster than checking instructions individually, it is not possible to determine if the simulation was performed accurately. For example, a floating-point benchmark that is sensitive to minor variations in execution precision may very well not execute large sections of code when executed on a simulator that normally would be executed on hardware. The simulation results of running that benchmark are then tainted and may not be valid, depending on the simulation application [14].

An alternative to both of these verification methods is to automatically verify the results of each instruction of a benchmark as it is simulated. The rest of this paper explores this alternative, and is organized as follows: related work is presented in Section 2. The general method of verification is outlined in Section 3, followed by a discussion of the specific implementation of this method in Sections 4 and 5. Results, conclusions, and future work follow in Sections 6 and 7.

2 Related Work

Previous work in this area has been focused on hardware verification, simulation functionality, or simulation speed. Very little work has been done on verifying the correctness of processor simulators.

In general, most verification research in computer architecture is done with respect to verifying that the hardware behaves according to specifications. A method that is essentially the reverse of the method presented here is described in [3] in which state comparison between a target high-level description language (HDL) model and a software behavioral reference model is done while running real-world applications instruction by instruction. Other more formal methods of hardware verification [8, 9, 1] use state trees and theorem provers to "prove" the correctness of the implementation.

Simulators, such as SimOS [13] and SimpleScalar [2], which are the basis of much architecture research, have been used extensively but never systematically verified. The output of programs run under both simulators has been compared to the output of the same programs run on real hardware, but, as noted by Sargent [14], this is only part of the *dynamic testing* approach for verifying simulation software. Dynamic testing also involves checking the values generated during the execution of the programs. Ideally, in the case of simulators, this testing would check the values generated by each instruction as it is generated. Implementation of an entire instruction set is error-prone, even with the infrastructure provided in the simulators. Other methods to automatically generate simulators [10, 7], or a specific language for implementing instruction simulation code [11], do not completely address the issue of verifying the simulation implementation.

3 Methodology

Performing an exhaustive verification of a simulated instruction set architecture (ISA) is, in general, not possible. Given that the ISA has a word width of w bits, n instructions in the set, and 0 to 3 inputs per instruction, there are $O((2^w)^3 n)$ possible combinations of instructions and inputs.

However, for the purposes of high-level instruction and memory timing simulators, an exhaustive verification is not required. It is necessary only to verify that the (reasonable) set of benchmarks that will be used during simulation of new instructions or features executes correctly. Verifying that the set of benchmarks is simulated correctly provides a solid basis for further extensions, simulations, or analyses.

The basic premise of the verification method being proposed is simple: given the same (user-visible) initial register state, run the program, instruction by instruction, both natively (on the actual hardware) and by simulation.

After each instruction executes, the native register state and the simulated register state are compared. If a mismatch is detected, an error has occurred in the simulation of the instruction.

The memory state is never compared between instructions. For a load/store architecture, the lack of memory state comparison not a problem. If a

store instruction stores the wrong value into memory, when it is later read with a load instruction, the wrong value will be loaded and the register states will miscompare. This condition may not point directly to the offending instruction, but the class of instructions that failed is known by inference. However, if a store instruction stores the wrong value into memory, but the memory is never accessed again, the error will go undetected. Note that it is possible to extend the simulation technique to include memory state checking as well. Checking the memory state between instructions will catch incorrect implementations of stores and loads as soon as they occur, although at a substantial increase in overall verification time.

Additionally, both native and simulated instruction execution one at a time. This characteristic implies that this method is not able to verify instruction dependences in pipelined or superscalar execution.

This verification methodology has four requirements. First, the hardware being simulated must be present to execute instructions natively. Second, the initial memory state of both the native and simulated programs must match exactly. Third, it must be possible to single-step the execution of the program on the hardware, and to retrieve the register state after each instruction (although this requirement may be relaxed depending on the single-step interface, as discussed further in Section 4). Finally, any system calls that are made by the simulated program must match the native execution of the program. Note that this requirement can be satisfied by proxying the simulated system calls through the native system.

If memory state checking is to be included, a fifth requirement is added - it must be possible to read memory state from the natively executing program in order to perform memory state comparison.

A flow diagram summarizing this methodology is shown in Figure 1.

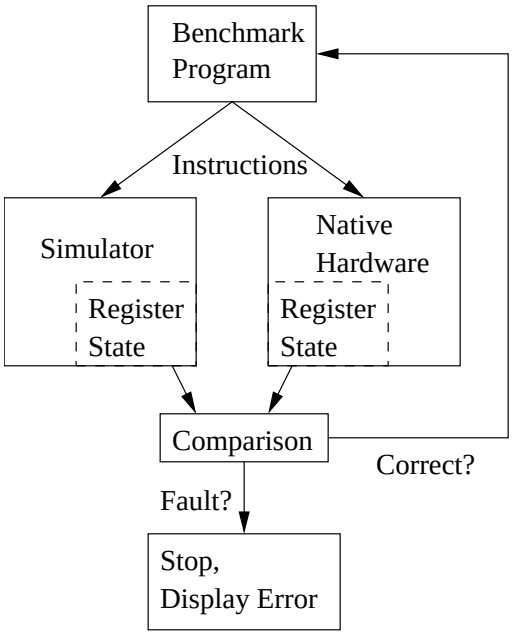

Figure 1: The basic flow of events in the verification methodology.

4 Implementation

A functioning verification simulator was created by modifying a baseline SimpleScalar (version 3.0) simulator [2]. The SimpleScalar simulator is a modular, extensible instruction set simulator. It is capable of anything from basic instruction simulation through complete, detailed timing simulations of a superscalar processor and memory hierarchy. The goal was to implement the user-level portion of the MIPS IV ISA [4, 5] in the simulator. This simulator should then be able run natively-compiled benchmark programs exactly as they run on the native hardware.

4.1 Preliminary Modifications

In order to obtain a simulator that runs straight MIPS N32 binaries directly, several modifications needed to be made to the baseline simulator.

Loader. The portable instruction set architecture-specific (PISA) loader, which loaded Common Object File Format (COFF) executables, needed to be replaced with a MIPS-specific Executable and Linking Format (ELF) [1] loader. ELF-format executables

[1] System V Application Binary Interface, Edition 4.1, http://www.sco.com/developer/devspecs/gabi41.pdf, and System V Application Binary Interface

under MIPS are compiled with an *interpreter* section. This section contains the name of the library or runtime linker that performs dynamic linking and/or library initialization. This *interpreter* is loaded into the simulator's memory along with the executable.

Instruction Set. The MIPS instruction set is not radically different from the PISA instruction set. PISA-specific machine files (pisa.h, pisa.c, and pisa.def) were replaced with their MIPS counterparts. The mips.h file contains macros for assisting in decoding MIPS instructions and register structures. The mips.c file implements a few miscellaneous functions for debugging the simulator and finishing the instruction decoding. Finally, mips.def is a nearly complete implementation of the MIPS IV ISA.

The mips.def file does not implement all MIPS IV ISA instructions, however. Privileged instructions, which never appear in user programs, are not implemented. The load linked (LL) and store conditional (SC) instructions, used to implement atomic memory operations in multiprocessors, work as normal load and store instructions since the simulator only simulates a single processor (the SC instruction always completes successfully, modifying the stored register to indicate as such). LL and SC would not have been implemented in this simulator except for the fact that they are used in Fortran library initialization code. Finally, the prefetch (LWC3) instruction is currently ignored, since the prefetch instruction does not modify any user-visible state (except for the program counter) and the verification simulator does not model any memory hierarchy timings.

System Calls. Finally, MIPS-specific system calls were implemented to replace the fictional PISA system calls. Especially important was the implementation of the `mmap()` system call, as `mmap()` is used by the interpreter to map the runtime linker into the address space of the process. It also is used by the runtime linker itself to map the remaining libraries required by the program into the process image. For the most part, the system calls are completed by proxy execution on the host machine. Certain system calls that would affect the simulator state or would query the state of the simulated machine (e.g. `brk()`, `mmap()`, `getpagesize()`, `syssgi()`) are intercepted and emulated on behalf of the simulated program.

(MIPS RISC Processor Supplement, 3rd Edition, http://www.sco.com/developer/devspecs/mipsabi.pdf

4.2 Verification Simulator Implementation

In order to verify each instruction as it was simulated, the program must be executed natively on the actual processor along with the simulated program.

Native Step Interface. The step-by-step execution of the native program is accomplished through the system's debug interface. For MIPS, the debug interface is a file named `/proc/xxxxx`, where `xxxxx` is the process ID of the program being stepped or debugged. Opening this file and performing certain `ioctl()` operations on the file descriptor allows the controlling process, which is the simulator, to control execution of the program, including single-stepping, reading register state, and reading or writing memory.

In order to start the native executable, a normal sequence of `fork()` and `exec()` is performed. However, between the `fork()` and `exec()`, the parent process uses the native process ID (obtained from the `fork()` call) to open up the process debug interface file. Then, by using the `ioctl()` operations provided, the simulator marks the native process to be stopped just after the `exec()` system call returns. Another call to `ioctl()` then waits until the native process is stopped (i.e., returns from the `exec()` system call).

In order to do complete verification, system call return values (both register and memory), as well as memory layout and values must, match exactly. After the native process is set up with the aforementioned procedure, the starting stack, its address, and the starting register state are copied into the corresponding memory addresses in the simulated process. Figure 2 shows the simulated and native programs, the stacks, the `ioctl()` control channel, and the main simulation loop.

At this point, both the native and simulated processors are ready for verification. Until the programs are complete or verification fails, the native processor and the simulated processor alternately single-step through the execution of the instructions. Register

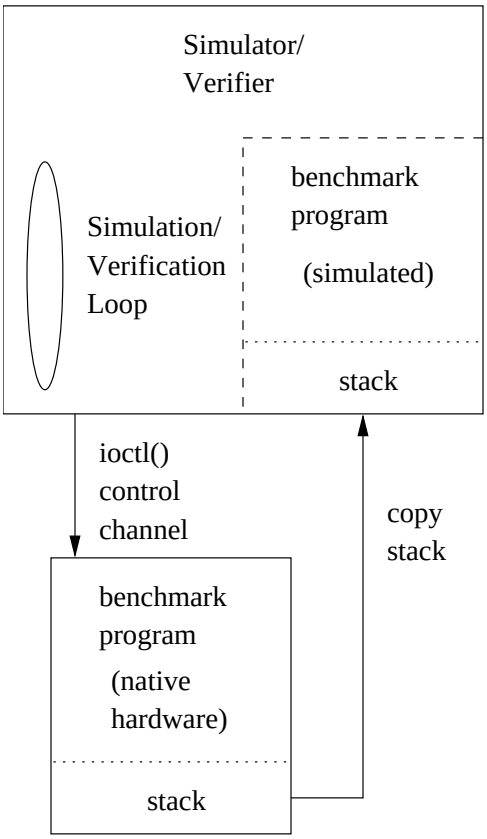

Figure 2: The sample implementation of the verification methodology.

results from the instructions are compared after each step to ensure that the results match exactly. Any discrepancy that occurs causes the verification process to terminate after displaying information about the offending instruction and which register(s) miscompared. The complete register contents, comprising the previous valid register state, the new valid register state (from the native processor), and the invalid register state (from the simulated processor), are output to help identify the cause of the miscomparison.

5 Implementation Difficulties

There were a few quirks in the native step interface that caused some trouble for the verification simulator. Note that these are in addition to errors in the simulated instruction set implementation (of which there were plenty).

Registers Not Preserved. Oddly enough, the first difficulty encountered was on the first executed instruction. Certain registers in the MIPS ISA are not preserved across system calls, or for the first instruction of a natively executed program. These include not only the kernel registers, k0 and k1, but also several temporary registers. Provisions to detect the first executed instruction and system calls were put into place. The kernel and modified temporary registers were then copied out of the native processor into the simulated processor.

Single-Step not always Single-Step. The MIPS ISA includes (possible) execution of a branch delay slot instruction. Thus, when single-stepping through the native process, stepping through a branch instruction executes two instructions instead of one. To accommodate this behavior, the register state is verified only when the simulated process catches up to the native process. Furthermore, the native process only single-steps through an instruction when the simulated and native program counters are equal.

A *divergence counter* is used to verify this branching behavior. Every time the native and simulated program counters are equal, the divergence counter is set to zero. Every time the simulated program exe-

cutes an instruction, the divergence counter is incremented. If the divergence counter reaches n, meaning the simulated process has advanced n more instructions than the native process has, the simulator concludes that a previous branch was evaluated incorrectly. It then aborts the verification process after dumping out the history of the last 48 program counter values for both the simulated and native process ($n = 32$ instructions is an arbitrary value that was used in this implementation and is discussed further below).

Single-Step/Instruction Conflicts. The LL and SC instructions also conflicted with the single-step interface to the native program. According to the MIPS R10000 specification, an exception that occurs between the LL and SC instructions will cause the SC to fail. Since each single-step operation on the native program causes an exception, execution of any SC instruction will fail. These instructions are typically located in a loop that repeats until the atomic memory transaction is completed successfully. As a result, the loop will never complete (since SC always fails).

To compensate for this behavior, whenever the native step interface encounters an LL instruction, it searches forward for the corresponding SC instruction. It then executes all instructions from the LL instruction up to and including the SC instruction instead of single-stepping each instruction. The simulated processor then is allowed to execute single instructions up to and including the SC instruction. Verification of this case is handled by the divergence counter as explained above. Since LL/SC atomic memory operations typically have only a few instructions between the LL and SC instructions, a divergence limit of $n = 32$ instructions should be sufficient to accommodate this behavior.

System Call Differences. System calls need to return absolutely identical information whether the returned values are in registers or are placed in memory via pointers passed to the system call. Even little things, like the number of the file descriptor returned by the `open()` system call, have to match exactly between the native and simulated processes. This turned out to be a problem as the simulator, in addition to having standard input, standard output, and standard error opened on behalf of the simulated process, also had a file descriptor opened in order to perform the native process tracing. This additional file descriptor meant that the numbers returned by the first `open()` call did not match. Using `dup2()` to duplicate the native process file descriptor onto a high-numbered descriptor and closing the original descriptor solved the problem.

6 Results and Performance

As of this writing, two SPEC benchmarks, `129.compress` and `132.ijpeg`, have been completely verified running the `test` input set [15, 12]. Incorrect implementations of several simulated instructions were detected and corrected during these simulations.

Benchmark verification, even on the relatively small `test` input set, takes quite some time. The single-step interface limits verification performance to about 200 instructions simulated, executed, and verified per second. For comparison, the basic, functional SimpleScalar simulator (`sim-safe`), can simulate approximately one to two million instructions per second. Thus, the verification simulator is approximately 50,000 times slower. Estimates of simulation verification times for other SPEC95 benchmarks using the `test` input sets range from several hundred hours to several thousand hours.

7 Conclusions and Future Work

This method of verifying instruction-level simulators does work, as evidenced by the successful instruction-level simulation and verification of the two SPEC benchmarks, albeit very slowly. This method, combined with a wide range of smaller programs, can be used to adequately verify the instruction set implementation for architectural research purposes. After a reasonable set of programs and instruction mixes has been verified, research and simulation can proceed with confidence that instructions and programs are executing properly and completely.

Possible future extensions to this work could speed up execution time to verify larger benchmark programs and input sets. For example, it should be possible to do hierarchical verification. This would involve scanning the code section of the benchmark once before execution and placing a breakpoint at the beginning of each function. At each breakpoint, comparison between the simulated and native register states could be performed. Any mismatch should show the function in which the instruction simulation error occurred. The simulator could then either restart the verification from the last known good state (checkpoint), or it could restart the entire process with a single breakpoint at the last known good state. The single-step verification then can proceed to pin down the simulation error.

References

[1] Jerry R. Burch. Techniques for verifying superscalar microprocessors. In *Proceedings of the 33rd ACM/IEEE Conference on Design Automation*, pages 552–557, 1996.

[2] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, June 1997.

[3] You-Sung Chang, Seungjong Lee, In-Cheol Park, and Chong-Min Kyung. Verification of a microprocessor using real world applications. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 181–184, 1999.

[4] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[5] Joe Heinrich. *MIPS R10000 Microprocessor User's Manual*. MIPS Technologies, Inc., Mountain View, CA, 1995.

[6] Richard C. Ho, Han Yang, Mark A. Horowitz, and David L. Dill. Architecture validation for processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 404–413, 1995.

[7] Tor E. Jeremiassen. Sleipnir – an instruction-level simulator generator. In *2000 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 23–31, 2000.

[8] Robert B. Jones and David L. Dill. Efficient validity checking for processor verification. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, pages 2–6, 1995.

[9] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Proceedings of the 33rd ACM/IEEE Conference on Design Automation*, pages 558–563, 1996.

[10] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *1998 International Conference on Computer Languages*, pages 80–89, 1998.

[11] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[12] Jeff Reilly. A brief introduction to the SPEC CPU95 benchmarks. *Computer Architecture Technical Committee Newsletter*, pages 1–8, June 1996.

[13] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[14] Robert G. Sargent. Validation and verification of simulation models. In *Proceedings of the 1999 Winter Simulation Conference*, pages 39–48, 1999.

[15] SPEC CPU95 benchmarks, 1995. http://www.specbench.org/osg/cpu95/.