High speed OS simulation using high performance microprocessor

Mohammad Mahbubur Rahman,† Takashi Nakada †
and Hiroshi Nakashima†

Hardware-software co-design aims to provide an integrated environment for concurrent specification, validation and syntheses of both hardware and software. In order to ensure the accuracy and performance of hardware-software co-design, high speed simulation of system level is indispensable. Is spite of this, many simulators omit the system level simulation in order to make the simulation faster and simpler. Therefore, this paper presents the development of a fast and accurate simulator, based on SimpleScalar, which is capable of micro-architectural modeling and system level simulation. Our primary contribution is to implement the necessary modification into the simulator, and build necessary environment for running a real-time operating system. With this simulator we can observe task switching and interrupt processing in a real-time operating system.

1. Introduction

The emergence of the system-on-chip(SOC) era is creating many challenges at all stages of the design process. At the system level, engineers are reconsidering how designs are specified, partitioned and verified. Hardware-software co-verification aims to provide an integrated environment for concurrent specifications, validations and syntheses of both hardware and software. These tools allow designers to run software against a hardware design before a prototype is available. This is accomplished by executing the software on a logic simulation of the hardware design. Therefore, execution-driven simulation is a valuable tool for the evaluating new computer architectures, since it allows the researchers to modify virtually any part of the computer system without incurring the cost of developing new hardware.

Designing a detailed execution-driven simulator is a trade off between accuracy, simulation speed, and development effort. In many cases, assumptions are made to assure the simpler and faster simulation. One simplification that reduces simulation time is to model only user level code and not the machine's privileged operating system code. As a result, simulators ignore operating system and I/O activity. Unfortunately, removing the operating system from the simulation model degrades both the accuracy and the applicability of the simulation environment. For instance, simulation of hardware-software co-verification tools can give unexpected results as an effect of omiting simulation of operating system. Even though some simulators simulate operating systems, they simulate it in a very slow and complex manner. Therefore, simulator capable of fast and accurate micro-architectural modeling with system level simulation is badly needed.

In this paper, we describe the development of a fast and accurate simulator which is capable of micro-architectural modeling and system level simulation. With this simulator we can observe task switching and interrupt processing in a real-time operating system while the main software execution interacted with the hardware in the target system.

2. Related works

There exists a number of simulation tools that contain detailed models of today's high performance microprocessors. For example, Maynard et al. showed that commercial applications are typically more challenging with respect to their interaction with the memory system and branch prediction mechanisms. One distinctive factor is that they typically have more operating system interactions than scientific/engineering applications. By contrast, scientific/engineering applications typically spend a very small fraction of their execution in system calls. As a result, they often disregard the impact of operating system code on architectural decisions.

The handling of system calls by the microprocessor simulator determines which of two categories they fall under. The first category of simulators uses a method called system call by

† Toyohashi University of Technology

proxy, in which system calls are executed by the simulators with the help of the operating system of the host computer. These are the user level simulators. Because the handling of the system call is not performed within the simulation context, the simulator cannot monitor the effect of the system call. This approach reduces the accuracy of the results, but simplicity allows quick simulation. Several tools such as Asim, MINT, Rsim, Shade, and SimpleScalar belong to this group. Our paper implements a simulator based on SimpleScalar which is capable of operating system simulation. We will describe about SimpleScalar later in more details.

The second category installs a virtual operating system in it and when a system call occurs it uses the installed virtual operating system to process it. These are the system level simulators. The development of complete system simulators has been directory motivated by the inability of user-level simulators to target complex workloads. Although complete system simulation tools are extremely large and complex, these benefits are diverse and significant such as evaluation of hardware design, development of operating system, and performance tuning of workloads. Complete system simulation approaches such as SimICS, SimOS, MR-Rsim can functionally model the execution of applications with operating system interaction on the instruction-set architectural abstraction level.

- SimOS[1)]
  SimOS is an environment for analyzing performance of architectural and software design alternatives of computer systems. It can boot IRIX 5.3 or DEC UNIX and run realistic workloads. Furthermore, the large and complex nature of operating systems required SimOS to include multiple interchangeable simulation models of each hardware component that can be dynamically selected at any time during the simulation. The emulation of operating system in SimOS is complex than the emulation of a complete machine simulation approach.
- SimICS[2)]
  SimICS is an instruction-set simulator developed at the Swedish Institute of Computer Science(SICS). It simulates one or more SPARC V8 processors, and supports multiple physical address spaces, system level code, and emulation of the SunOS 5.x API for direct analysis of user-level programs.
- ML-Rsim[3)]
  ML-RSIM is an event-driven cycle-accurate simulator that integrates detailed processor and cache models with a complete I/O subsystem. Combined with the Unix-compatible Lamix operating system, ML-RSIM provides a unique tool that allows researchers to study the interaction of computer architecture, I/O activity, system software and applications. ML-RSIM executes static SPARC V8 binaries. The Lamix system call interface is compatible with Solaris 2.8. In general, applications compiled for ML-RSIM can execute on native Sparc/Solaris systems without modification. No special libraries or include files are required to compile applications for the simulator.
- SimpleScalar[4)]
  SimpleScalar toolset provides an infrastructure for simulation and architecture modeling. The toolset can model a variety of platforms ranging from simple unipipelined processors to detailed dynamically scheduled micro-architectures with multiple-level memory hierarchies. It is the most widely used user-level simulator because of its flexibility in modeling a wide range of micro-architectural design points. But since the effect of the operating system is ignored in SimpleScalar, the accuracy and applicability of the simulation model is degraded.
  This paper introduces a simulator which is based on SimpleScalar and is capable of supporting system level simulation and cycle level micro-architecture simulation.

3. Overview of the proposed simulator

The proposed simulator is based on SimpleScalar and capable of simulating a real-time operating system on it. SimpleScalar runs as a target machine on the host, which consists simulated models of CPU instructions and pipelines. On the top of the machine, we port a real-time operating system.

3.1 SimpleScalar simulator

This subsection briefly describes the functionality of SimpleScalar and memory and the memory address mapping.

SimpleSclar models many different instruction set architectures(ISAs), the ALPHA and MIPS excluding the privileged instructions. An

ISA is the protocol for the commands, a microprocessor implements. In this project, we modified the simulator's implementation that models a pipeline similar to MIPS R10000. This simulator uses a instructions set of MIPSIV. The toolset implements six simulators, which can model the microprocessors in different levels of details. The functional model of simulator is sim-safe and the out-of-order simulation is sim-outorder. This paper focused on sim-safe to simplify the modifications.

Before SimpleScalar executes a benchmark program, it first creates a memory space for the program. The simulator loads the program from the hard drive into this memory space. Along with the actual program, the simulator provides space for data storage mechanisms, including the stack and a data segment. To reference this memory space, the simulator creates a memory page table for each process. This is similar to the manner in which Unix systems load programs. Treating the emulated operating system as a benchmark program is the logical extension of this scheme. The emulated operating system is given its own memory space.

3.2 Operation system choice

There are several embedded operating systems suitable for interaction into the targeted simulator. However, only few operating systems have support for multiple processors and have freely available source code. One such operating system is TOPPERS/JSP .

TOPPERS/JPS kernel is developed by Nagoya university/Toyohashi Institute of Technology Real-time System Laboratory as a part of the TOPPERS project which is a real-time operating system based on the specifications of μ ITRON4.0 . JSP is the abbreviation of Just Standard Profile which indicates that it follows the μ ITRON4.0 specifications.

We will use JSP1.3 kernel that targets MIPS. We port this kernel and simulates it on the SimpleScalar simulator.

4. Methods of simulation

There are two main tasks to be performed in order to simulate an operating system on SimpleScalar. First, we need to modify SimpleScalar so that it becomes capable of running an operating system on it. Secondly, we have to port an operating system in SimpleScalar.

In order to simulate an real-time operating system on SimpleScalar, implementation of interruption is the most necessary task. Thus, Simulator needs an extension including implementing interrupt controller and inclusion of privileged instructions. Here the privileged instructions are move from/to Coprocessor0, eret, syscall. MIPS specification gets 17 kinds of interruptions and exceptions that should be implemented. In addition, modifying the present compiler is also needed so that it can compile the newly added privileged instructions and thus can compile the SimpleScalar-ported JSP1.3 kernel.

Porting of an operating system requires rewriting of the interrupt handler according the instruction set of the target system. It also requires porting of boot loader and initialization of kernel.

In the JSP1.3 specification, the preemptive, priority-based task scheduling is conducted based on the priorities assigned to tasks. If there are a number of tasks with the same priority, scheduling is conducted on a "first come, first served" (FCFS) basis.

The precedence for executing each processing unit and the dispatcher is specified as follows:

(1) interrupt handlers, time event handler
(2) dispatcher(one of the kernel process)
(3) tasks

Task management functions include the ability to create and delete a task, to activate and terminate a task, to cancel activation requests, and to reference the state to a task.

As JSP1.3 kernel does not have any procedure for file management and network management, we do not emphasis into these at this stage of simulation. We will implement timer interruption in order to perform the task management of the JSP1.3 and we will use the proxy system handler of SimpleScalar in order to provide environment to perform I/O tasks needed by JSP1.3. Figure 1 shows the structure of SimpleScalar with the proposed implementation of operation system.

Here we describe, the design of interrupt controller and privileged instructions.

4.1 Interrupt controller

In addition to its normal computation functions, any CPU needs units to handle interrupts, and some way of observing or controlling on-chip functions like caches and timers. In MIPS architecture to perform this task, there is a Co-processor, Co-processor0. This Co-processor0 has 32 registers. Among these registers the following registers are essential for our implementation of interruption.

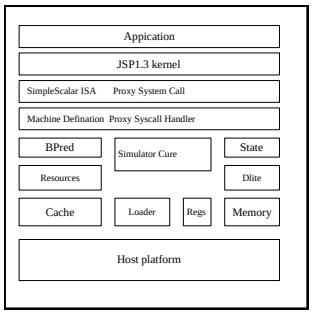

Fig. 1 Modified structure of SimpleScalar

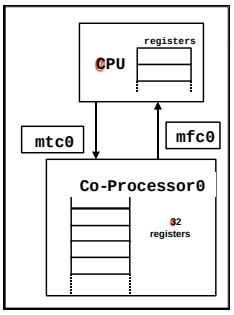

Fig. 2 Implemented interrupt controller and privileged instructions

Now, we will describe these registers with relevant fields of interruptions.

- Count/Compare registers
  These registers provide a simple general-purpose interval timer that runs continuously and that can be programmed to interrupt.
  The Count register acts as a real-timer. It is incremented at exactly half the CPU's pipeline clock rate. When it reaches the maximum 32-bit value it overflows quietly back to zero. The only way to disable the timer interrupt is to negate the interrupt mask bit, IM[7], in the Status register. timer interrupt cannot be disabled without also disabling the Performance Counter interrupt, since they share IM[7].
  The Compare register can be programmed to generate an interrupt at a particular time, and is continually compared to the Count register. Whenever their values equal, the interrupt bit IP[7] in the Cause register is set. This interrupt bit is reset whenever the Compare register is written.
- Status register
  The Status register(SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. From bits no. 15 to 8 is the IM(Interrupt Mask)field which Controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register. The bits no. 4-3, 2, 1, 0 are fields that define Kernel, Supervisor, User mode, Error Level, Exception Level and enable of interruption respectively.
- Cause register
  The Cause register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. This 32-bit read/write Cause register describes the cause of the most recent exception. The bits no. 15 to 8 are IP field that defines Interrupt Pending and indicates an interrupt is pending. These bits follow the CPU inputs for the six hardware levels. Any of the bits active when enabled by the appropriate Status (IM) bit and the global interrupt enable flag IE will cause an interrupt. The setting of bit 15 indicates timer interruption. The bits no. 5 to 2 are ExcCode field that defines what kind of exception happened.
- EPC register
  The Exception Program Counter(EPC)is a read/write register that contains the address at which processing resumes after an exception has been serviced. The processor does not write to the EPC register when EXL bit in the Status register is set to a 1. Figure 2 shows the implemented interrupt controller and privileged instructions.

Table 1 Registers necessary for interruption handling

mnemonic	no.	Description
Count	9	Timer count
Compare	11	Timer compare
Status	12	Processor Status register
Cause	13	Cause of the last exception
EPC	14	Exception Program Counter

4.2 Privileged instructions

Processing of interruption requires privileged instructions. In order to processing interruption, the following instructions are needed.

- MFC0:Move From Co-processor0
  - Format:
    MFC0 rt,rd

– Description:
The contents of Co-processor register rd of the Co-processor0 are loaded into general register rt.

- MTC0:Move To Co-processor0
  – Format:
  MTC0 rt,rd
  – Description:
  The contents of general register rt are loaded into Co-processor register rd of Co-processor0.
- ERET:Exception return
  – Format:
  ERET
  – Description:
  ERET is the instruction for returning from an interrupt or exception. Unlike a branch instruction, ERET does not execute the next instruction. ERET loads the PC from the EPC, and clear the EXL bit of the Status register.

5. Implementation

5.1 Modification of SimpleScalar

We modified SimpleScalar according the specification of MIPS R10000. We implemented an interrupt controller which consists of 32 registers. We introduced privileged instructions into the ISA of SimpleScalar in order to make it capable of running interrupt handler. We also modified the compiler and implemented a timer that keeps track of time in the hardware and generates OS tick.

5.1.1 Modifying compiler

For implementing new instructions in ISA needs modification of compiler. The compiler used for SimpleScalar is sslittle-na-sstrix-gcc. Instead of modifying the compiler itself, we modified the assembler portion of the compiler. We will write interrupt handler by using these instructions by using asm inline so that these instructions can be compiled without interpreting. Thus the modified assembler gas will change appropriate machine code for these instruction making the simulator CPU to execute them.

5.1.2 Implementation of timer

We implemented a timer in SimpleScalar. The timer has a single register which increases at every other clock of the processor. We set the timer interruption to be occurred at every 1 milli sec according the specification of JSP kernel.

Here, we describe implementation of timer interruption which is responsible for behaviour of timer.

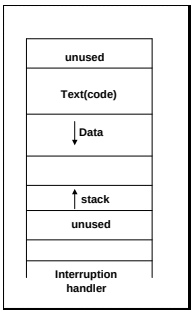

Fig. 3 Memory implementation of SimpleScalar

- Cause: A Timer interrupt is raised when the Count and Compare registers reach the same value.
- Processing: When a timer interruption occurs hardware does the following tasks.
  – ExcCode of Cause register is set.
  – The IP7 bit of the Cause register is set.
  – The EPC register contains the address of the following instruction.
  – EXL bit of Status register is set.
- Servicing:
  When a timer interruption occurs, interrupt handler does the following tasks
  – Compare register is reset.
  – Reseting Compare register will automatically clean the IP7 bit in Cause register.

Figure shows the memory implementation of SimpleScalar.

5.2 Porting of JSP1.3

Porting of an operating system needs task management, file management, I/O management and network management. The most important of them is task management. We also ported the system of interrupt handler since it differs on the target hardware interface. We ported kernel initialization and boot loader.

5.2.1 Interrupt handler

We implemented the interrupt handler of the jsp1.3 according to the instruction sets of SimpleScalar. This program writes the interrupt handler code in a fixed address of memory (0x80000180) before enabling interruptions. When an interruption is occurred the program control will be transferred into the interrupt handler and after processing the interruption, the normal execution will restore. The program uses k0, k1 register of general purpose register which are reserved for kennel use. The algo-

rithm of this interrupt handler is described as follows:

- The processing of interruption needs using of registers. We use some general purpose registers for that. Before using these registers, the contents of them are saved.
- Check the ExcCode of the Cause register to find out the cause of interruptions.
- The processing code is different for interruptions. The program then goes the fixed address according to the interruptions.
- While processing exceptions, it is possible that interrupt pending bit of the Cause register is set for timer interruptions. So, processing of exceptions follow the check of IP7 bit of the Cause register as IM7 (mask bit of Status resister is set) to find out whether the timer interruption processing is needed.
- If there is a timer interruption, necessary service is taken.
- The saved general purposed registers are restored.
- Before restoring the main program, ExcCode of the Cause register is cleared.
- Restore the normal program execution.

5.2.2 Initialization of kernel

Kernel of JSP1.3 should be initialized in order to make the system started executing. Kernel initialization consists initialization of interruption mask in the task context, initialization of interrupt handler and initialization of task management. Therefore, initialization of timer occurs and then starts the execution of kernel. We also ported the link loader file of JSP1.3 into SimpleScalar. We define the .text section, .data section and .bss section of JSP1.3 kernel according the specification of SimpleScalar and MIPS.

6. Simulation of JSP1.3 kernel in SimpleScalar

We compiled a SimpleScalar-ported JSP1.3 kernel with the compiler for SimpleScalar in order to make a binary of it. The programs to be run on JSP1.3 kernel is also compiled during the compile of the kernel and have a common binary. Therefore, before entering the main loop of instruction execution we write this binary into a fixed defined place of the memory structure of SimpleScalar. Thus, the program counter begins to execute from that defined address. In this way, we simulate that binary running on sim-safe, a simulator of SimpleScalar. The operating system prints a message and waits for a further instruction.

7. Conclusion and future work

SimpleScalar tool set, provides an infrastructure for simulation and architectural modeling, is enable to simulate only user-level contexts and omits the system level contexts. In this paper, we discussed our efforts to port a real-time operating system to SimpleScalar in order to make it enable of simulating system level contexts with a high performance micro-architecture system. We modified SimpleScalar in order to make it enable of simulating the operating system. We aim to simulate system level contexts in high-speed machine simulation by combining the increasing power of today's computers in advance. In future, we aim to simulate the operating system into a complex pipelined system. Furthermore, our future plan is to process tasks parallelly by using clusters and then hope to ensure high-speed simulation of the operating system.

Acknowledgments This research is partially supported by Semi-conductor Technology Academic Research Center(STARC) Japan, under project name Verification of performance of the software designing with SpecC.

References

1) Mendel Rosenblum, Stephen A. Herrod, E. W. and Gupta, A.: Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology*, Vol. 3, No. 4, pp. 34–43 (1995).
2) SimICS: http:/www.simics.com.
3) ML-Rsim: http://www.cse.nd.edu/ mlrsim.
4) Burger, D. and Austin, T. M.: The SimpleScalar Tool Set, Version 2.0 (1997).
5) Burger, D. and Austin, T. M.: SimpleScalar: An Infrastructure for Computer System Modeling,*IEEE Computer*, pp. 59–67 (2002).
6) Heinrick, J.: MIPS R10000 Microprocessor User's Manual: Version 2.0, *MIPS Technologies Inc.* (2001).
7) MIPS, T.: MIPS64 Architecture for Programmers Volume I,II,III: Resision 1.00, *MIPS Technologies Inc.* (2002).
8) Hennessy, J. and Paterson, D.: *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publisher (1998).
9) Sweetman, D.: *See MIPS Run*, Morgan Kaufmann Publisher (1997)
10) Sakamura, k., Takada, H: μ ITRON4.0 Specification,*TRON ASSOCIATION,JAPAN*, ver 4.0(2002).