Flexim: A Modular and Event-Driven Cycle-Accurate Simulator to Evaluate Multicore Processors

Min Cai, Zhimin Gu

Beijing Institute of Technology, Beijing 100081, P.R.China

Abstract

Good cycle-accurate simulators are critical for conducting successful multicore processor architecture research nowadays. However, most of the existing simulators are written in C or C++ for speed considerations and the modeled computer structures and functionalities are too complicated to be implemented in a clear yet efficient way. The resulting unreadable code makes the simulator hard to use and extend. As a niche market, other not-so-realistic simulators are mostly used for educational and visualization purposes that they are written in traditional object-oriented languages such as Java or C#, which omit many machine details that are necessary for architectural research. There is a permanent need of balancing the art and science of simulating multicore architectures. In this report, we present Flexim, which exploits modularity and event-driven capabilities to remodel the core functionalities of the classic SimpleScalar simulator and extends it to enable detailed simulation of out-of-order cores and multi-level cache hierarchies of multicore processors. Simulation results of a few popular benchmarks are shown for illustrative purposes.

Keywords: Cycle-accurate simulator, multicore processor architecture

1 Introduction

Decades of technology advances and architectural innovation in microprocessors has led to complex multicore designs that combine multiple physical processing cores on a single chip. Such design consists of three major parts: the microprocessor core, the cache hierarchy and the interconnection network. The design and implementation of the microprocessor core and the cache hierarchy are highly coupled and interrelated. And the interconnection network provides a fast and efficient transmission media that glues all the computing and storage resources on the chip together.

In order to evaluate the impact on the overall performance of any design improvement on any of the three major parts, many cycle-accurate simulators are created to model the three major parts and their integration in a system working as a whole. Yet many of architectural simulators come and go with a short lifecycle, because they are unable to satisfy the stringent yet fluid needs of innovating new computer architectures.

Most of the existing simulators are written in C or C++ for speed considerations and the modeled computer structures and functionalities are too complicated to be implemented in a clear yet efficient way. The resulting unreadable code makes the simulator hard to use and extend. As a niche market, other not-so-realistic simulators are mostly used for educational and visualization purposes that they are written in traditional object-oriented languages such as Java or C#, which omit many machine details that are necessary for architectural research. There is a permanent need of balancing the art and science of simulating multicore architectures.

In this work, we present Flexim, which exploits modularity and event-driven capabilities to remodel the core functionalities of the classic SimpleScalar simulator and extends it to enable detailed simulation of out-of-order cores and multi-level cache hierarchies of multicore processors. Table 1 [] summarizes the main parameterizable options of Flexim, broken down according to the presented components classification.

The rest of this technical report is structured as follows. Section 2 provides an overview of existing processor simulators and their comparative features. Section 3 describes Flexim with significant development details. Section 4 discusses the provisioning of modular and event-driven simulation of out-of-order cores. Section 5 discusses the provisioning of modular and event-driven simulation of multi-level cache hierarchies. Simulation results from some popular benchmark suites are shown in section 6. Finally, section 7 concludes the report.

2 Related Work

Multiple simulation environments, aimed for computer architecture research, have been developed. The most widely used simulator during the recent years has been SimpleScalar [1], which serves as the basis of some Flexim functional simulation modules. It models an out-of-order superscalar processor. Lots of extensions have been applied to SimpleScalar to model certain aspects of superscalar processors in a more accurate manner. For example, the HotLeakage simulator [] quantifies leakage energy consumption. However, SimpleScalar is quite difficult to extend to model new parallel microarchitectures without significantly changing its structure. In spite of this fact, three SimpleScalar extensions to support multithreading and/or multicore have been implemented in the SSMT [], M-Sim [] and Multi2Sim [5] simulators. While SSMT and M-Sim are useful to implement designs based on simultaneous multithreaded processors, Multi2sim provides detailed simulation of multicore multithreaded processors in x86 ISA.

SimpleScalar is an *application-only* tool, that is, a simulator that executes directly an application and simulate its execution by providing a simplistic and fictitious underlying operation system via system call emulation. Such tools are characterized by not supporting the architecture-specific privileged instruction set, since applications are not allowed to execute it. However, application-only simulators have the advantage of isolating the application execution, so statistics are not affected by the simulation of a real operating system. The proposed simulator Flexim can be classified as an application-only simulator, too.

In contrast to the application-only simulators, a set of so-called *full-system* simulators are available. In such environments, an unmodified operating system is booted over the simulator and applications run at the same time over the simulated operating system. Thus, the entire instruction set and the interfacing with functional models of many I/O devices need be implemented, but no emulation of system calls is required. Although this model provides higher simulation power, it involves a huge computational overhead and sometimes unnecessary simulation accuracy.

Simics [3] is an example of full-system functional simulator which is commonly used for multiprocessor systems simulation although it is not freely available. Simics provides a powerful set of APIs that a variety of its extensions have been created for specific purposes in this research area. GEMS [4] is a popular Simics extension which provides timing simulation capabilities to model the architectural details of multiprocessors such as instruction fetch and decode, branch prediction, dynamic instruction scheduling and execution and speculative memory hierarchy access. GEMS also provides a specification language for defining cache coherence protocols. However, any simulator based on Simics, like GEMS, must boot and run an operating system, so high computational load is increased with each extension.

GEMS provides an important feature of processor simulators which is called *timing first simulation*. In this scheme, for example, one timing simulation module traces the state of the processor pipeline while instructions traverse it, and another timing simulation module traces the state of the caches while requests traverses them. In the pipeline case, the functional module is only called to actually execute the instructions when they reach the commit stage, so the correct execution paths are always guaranteed by a previously developed robust simulator. The timing-first approach confers efficiency, robustness, and the possibility of creating a series of simulators gradually with more and more details and accuracy.

And the last well-known simulator that we mention here is M5 [2] . This simulator provides support for simple one-CPU functional CPU, out-of-order SMT-capable CPUs, multiprocessors and coherence caches. It integrates the full-system and application-only simulation modes.

As summary, Flexim has been developed integrating the most significant characteristics of important simulators, such as separation of functional and timing simulation, SMT and multiprocessor support and directory-based cache coherence. Table 2 gathers Flexim's main features and marks the differences from existing simulators. Additional features of Flexim are detailed in further sections.

3 Basic Simulator Description

This section details the main implementation issues that lead to a final simulation environment, and exposes some tips to bring it into use with workloads that created from the combination of some popular benchmarks. These aspects are addressed by showing some usage scenarios, describing briefly the process of loading an ELF MIPS32 executable into a running process's virtual memory, and analyzing the simulator structure that divided by functional and detailed simulation.

3.1 Simulator and Workloads Compilation

Flexim is written in the systems programming language D, so in order to compile Flexim code, you need to obtain the latest DMD 2.0 compiler from DigitalMars' website[1], and follows the provided instructions to install the compiler. The latest Flexim code can be downloaded at the project's website on Github[2], as a compressed tar file, and has been tested on x86 and x64 machine architectures, with Linux OS. The following commands should be entered in a command terminal to compile it:

```
tar xzf mcai-flexim-<checkout_version>.tar.gz
cd mcai-flexim-<checkout_version>
make
```

Flexim simulates final executable files, statically compiled for the MIPS32 Little Endian architecture, so a cross-compiler is also required to compile your own program sources. There is a workable MIPS32 cross-compiler (cross-compiler-mipsel.tar.bz2) available on the Flexim's project website on Github[3].

Dynamic linking is not supported, so executables must be compiled statically. A command line to compile a program composed by a single source file named program.c could be:

```
mipsel-linux-gcc program.c -Wall -o program.mipsel32 -static
```

Executables usually have an approximate minimum size of 4MB, since all libraries are linked with it. For programs that use the math library or `pthread` library, simply include `-lm` or `-lpthread` into the command line. Sample Makefiles for a few benchmarks are provided within the Flexim distribution.

3.2 Executable File Loader

In a simulation environment, program loading is the process in which an executable file is mapped into different virtual memory regions of a new software context, and its register file and stack are initialized to start execution. In a real machine, the operating system is in charge of these

[1] http://www.digitalmars.com/d/download.html.
[2] http://github.com/mcai/flexim/.
[3] http://github.com/mcai/flexim/downloads/.

housework. However, Flexim, as other widely used application-only simulators (e.g., SimpleScalar), is not aimed at supporting the simulation of an OS, but only the execution of target applications. For this reason, program loading must be managed by the simulator during the initialization.

The executable files output by `gcc` follow the ELF (Executable and Linkable Format) specification. This format is aimed for shared libraries, core dumps and object code, including executable files. An ELF file is made up of an ELF header, a set of arguments and a set of sections. Typically, one or more sections are enclosed in a segment. ELF sections are identified by a name and contain useful data for program loading or debugging. They are labeled with a set of flags that indicate its type and the way they have to be handled during the program loading.

Flexim embodies the needed classes and methods to list the executable file sections and access their contents. The loader module sweeps all of them and extracts their main attributes: starting address, size, flags and content. When the flags of a section indicate that it is `loadable`, its contents are copied into memory after the corresponding fixed starting address.

The next step of the program loading process is to initialize the process stack. The stack is a memory region with a dynamically variable length, starting at the virtual address `0xc0000000` and growing toward lower memory addresses. The aim of the program stack is to store function local variables and parameters. During the program execution, the stack pointer (register $sp) is managed by the own program code. In contrast, when the program starts, it expects some data in it. This fact can be observed by looking at the standard header of the main function in a C program:

```
int main(int argc, char **argv, char **envp);
```

When the main function starts, three parameters are expected starting at the memory location specified by the stack pointer. At address [$sp], an integer value represents the number of arguments passed through the command line. At [$sp+4], an integer value indicates the memory address corresponding to a sequence of argc pointers, which at the same time represent each a null-terminated sequence of characters (program arguments).

Finally, at address [$sp +8], another memory address points to an array of strings (i.e., pointers to char sequences). These strings represent the environment variables, accessible through envp[0], envp[1]... inside the C program, or by calls to getenv functions. Notice that there is no integer value indicating the number of defined environment variables, so the end of the envp array is denoted with a final null pointer.

Taking this stack configuration into account, the program loader must write program arguments, environment variables and `main` function arguments into the simulated memory.

The last step is the initialization of the register file. This includes the [$sp] register, which has been progressively updated during the stack initialization, and the PC and NPC registers. The initial value of the register PC is specified in the ELF header of the executable file as the program entry point. Register NPC is not explicitly defined in the MIPS32 architecture, but it is used internally by the simulator to ease the branch delay slot management.

3.3 Functional Simulation

The functional simulation engine provides an interface to the rest of the simulator and implements functionalities such as creating or destroying software contexts, performing program loading, enumerating existing contexts, consulting their status, executing a new instruction.

The supported machine architecture is MIPS32 Little Endian. The main reasons for choosing the MIPS32 instruction set [4] is the availability of an easy-to-understand architecture specification and the simple and systematic identification of machine instructions, motivated by a fixed instruction size and an instruction decomposition in instruction fields.

As a remark, the difference between the terms `context` and `thread` should be clarified. A `context` is used in this work as a software entity, defined by the status of a virtual memory image and a logical file. In contrast, a `thread` is used as a processor hardware entity, and can comprise a physical register file, a set of physical memory pages, a set of entries in the pipeline queues, etc. The simulator kernel only handles contexts, and does not know of architecture specific hardware, such as threads or cores. [work should be done here]

Since Flexim simulates target applications, the underlying operating system services (such as program loading or system calls) are performed internally by the simulator. This is done by modifying the memory and logical registers status so that the application sees the result of the system call.

3.4 Detailed Simulation

The Flexim detailed simulator uses the functional engine to perform an execution-driven simulation: during each cycle, a sequence of calls update the existing contexts states. The detailed simulator analyzes the nature of the recently executed machine instructions and accounts the operation latencies incurred by hardware structures.

The branch predictor implementation is based on Marss86. Caches and TLBs are implemented, including the MESI cache coherence protocol. Current interconnection network is a constant-latency simplification. MMU (Memory Management Unit) is provided to map virtual address spaces of contexts into a single physical memory space. Physical addresses are then used to index caches or branch predictors, without including the context identifier across modules.

The out-of-order pipeline modeling module defines and manages a few processor structures such as decode buffers, reorder buffers, load store queues and functional units. And the pipeline stages defines the behavior of the configured multithreaded multicore processor.

Configuration specification and statistics output for the simulated entities such as cores, threads and caches are implemented via XML-formatted I/O, which is easy to use and machine-readable.

[4] The MIPS32 instructions that are not used by the `gcc` compiler are excluded from this implementation. Also instructions belonging to the privileged instruction set are not implemented.

4 Modular and Event-Driven Simulation of Out-of-Order Cores

5 Modular and Event-Driven Simulation of Multi-Level Cache Hierarchies

6 Results

7 Conclusion

Acknowledgments

This work was supported by the Ministry of Education of China-Intel Special Research Foundation for Information Technology program under the contract No. MOE-INTEL-08-10, and the National Science Foundation program under the contract No. XXX.

References

[1] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[3] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[4] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[5] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *19th International Symposium on Computer Architecture and High Performance Computing*. Citeseer, 2007.