Flexim: Streamlined Cycle-Accurate Simulation of 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 study. Oftentimes, users need to use a dozen of auxiliary tools and advanced scripting to automate the simulation process and do the housekeeping work such as statistics collection and reporting. There is a permanent need of balancing speed and elegance while simulating multicore architectures.

In this paper, we present Flexim, a streamlined cycle-accurate multicore architectural simulation platform, which consists of the simulator core and the GUI based Integrated Simulation Environment (ISE) . Within the core, Flexim remodels the core functionalities of the classic SimpleScalar simulator and extends it to enable configurable timing simulation of out-of-order cores and multi-level cache hierarchies of multicore processors. It exploits interface-based object orientation and cycle-accurate callback-based eventing to improve the modularity and extensibility of the simulator core. The GUI-based ISE uses componentization and visualization techniques to streamline the typical simulation life-cycle of architectural planning, simulator configuration and statistics collection & reporting. Various static and dynamic views are provided to show the user a consistent yet changing view of the simulation process and the multicore architecture under simulation. The pervasive use of XML files provide human readable and machine-friendly configuration and statistics representation that bridges the simulator core and the ISE. Simulations of a few popular benchmark suites such as Olden and CPU2006 are shown for illustrative purposes.

Keywords: Cycle-accurate simulator, multicore processor architecture, integrated simulation environment

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 together all the computing and storage resources on the chip.

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 life-cycle, 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 study. There is a permanent need of balancing speed and elegance while simulating multicore architectures.

In this paper, we present Flexim, a streamlined cycle-accurate multicore architectural simulator, which consists of the simulator core and the GUI based Integrated Simulation Environment (ISE) . Within the core, Flexim remodels the core functionalities of the classic SimpleScalar simulator and extends it to enable configurable timing simulation of out-of-order cores and multi-level cache hierarchies of multicore processors. It exploits interface-based hierarchical modularity and cycle-accurate callback-based eventing to improve the modularity and extensibility of the simulator core. For the GUI-based ISE, Flexim uses XML files to ease the configuration of the simulator, and utilizes graph visualization to provide static and dynamic views of the multicore architecture under simulation. Simulations of a few popular benchmark suites such as Olden and CPU2006 are shown for illustrative purposes.

The main contribution of this paper is highlighted as below:

1. To our knowledge, the proposed Flexim simulator is the first object-oriented architectural simulator written in the modern systems programming language D in the world. Having good software engineering practice in mind, a few D language features are utilized extensively to improve the simulator's code readability, maintainability and run-time reconfigurability.

2. Cycle-accurate callback-based eventing is used for modeling both the out-of-order pipeline model and multi-level cache hierarchy, clarifying the interaction between the execution-driven simulator and the timing simulation modules.

3. GUI-based integrated simulation environment is introduced to streamline the learning curve of architectural simulators. XML files and graph-based visualization are used to implement a reconfigurable yet user-friendly architectural simulator.

The rest of this paper 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 and cache hierarchies. Section 5 elaborates on the design and implementation details of the integrated simulation environment. Simulation results from some popular benchmark suites are shown in section 5. Finally, section 6 concludes the paper.

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 multi-processor 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.

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.

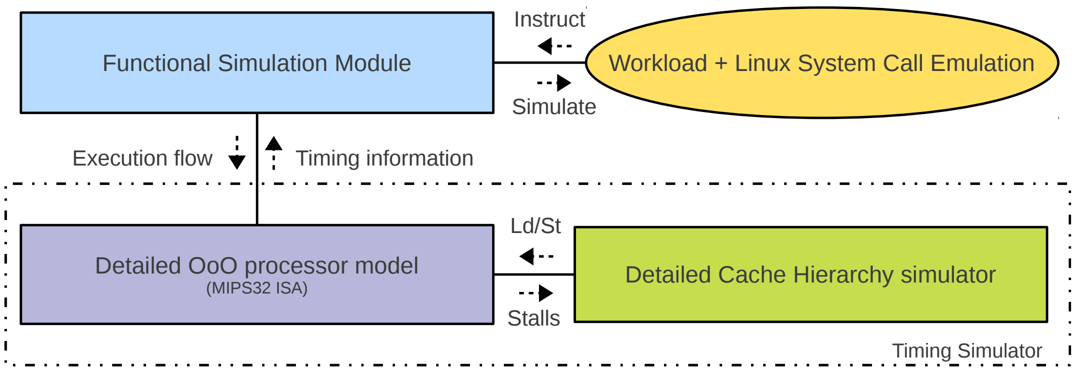

Functional Simulation Module
Instruct
Simulate
Workload + Linux System Call Emulation
Execution flow
Timing information
Detailed OoO processor model
(MIPS32 ISA)
Ld/St
Stalls
Detailed Cache Hierarchy simulator
Timing Simulator

Fig. 1: The relationship between workload, functional simulation and timing 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 Functional Simulation and Detailed Simulation

The relationship between Flexim modules used for functional simulation and detailed simulation are illustrated in Fig.1. The simulated workload and the system call emulation module provides the input instruction stream to be executed by the functional simulation module. And the functional simulation module feeds the execution flow information to the timing simulation module, and the timing simulation module provides feedback containing the requested timing information of instructions and memory requests.

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

3.2.1 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.

3.2.2 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 Object-Oriented Based Event-Driven Cycle-Accurate Simulation of Multicore Architectures

4.1 Overview

4.1.1 Simulated Multicore Architecture

As shown in Fig., our simulated baseline multicore system design consists of two OoO cores, a two-level cache hierarchy and a fixed-latency point-to-point on-chip interconnect. Both cache levels are lockup-free and store the state of outstanding requests via callbacks. Each of the private first-level split caches are currently implemented as write-back, but should be write-through with a coalescing store buffer. The shared second-level cache is write-back and maintains inclusion with respect to the first-level cache. Each cache maintains a on-chip directory.

[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.

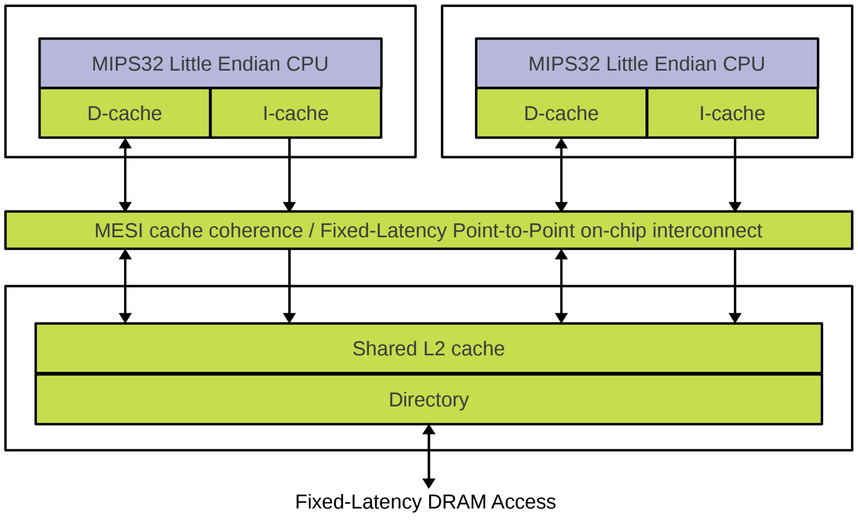


Fig. 2: Overview of the Simulated Multicore System

4.1.2 Object Orientation and Callback-Based Cycle-Accurate Eventing

4.2 Simulating Out-of-Order Cores

4.2.1 Behavioral Modeling

Flexim supports the mix of SMT and CMP simulation. As shown in Fig.3, the simulated out-of-order pipeline is divided into five stages: `fetch`, `decode/rename/dispatch`, `issue`, `execute`, and `commit`. The buffers between pipeline stages such as decode buffer (`DecodeBuffer`), reorder buffer (`ReorderBuffer`) and load/store queue (`LoadStoreQueue`) are explicitly modeled.

A set of parameters are provided to specify how the pipeline stages are organized in a multithreaded design. Stages can be shared among threads or private per thread. Moreover, when a stage is shared, there must be an algorithm which schedules a thread each cycle on the stage. The modeled pipe stages are described briefly as below.

The Five Pipeline Stages.

1. Fetch. The `fetch` stage takes instructions from the L1 instruction cache and places them into a decode buffer.

2. Decode, Rename & Dispatch. The `decode/rename/dispatch` stage takes instruction from a decode buffer, decodes them, maps architectural register dependencies (`RegisterDependency`) into allocated physical registers (`PhysicalRegister`), assigns them a reorder buffer entry and places them into a ready queue (`ReadyQueue`) or a waiting queue (`WaitingQueue`) depending on whether their input operands are available or not.

3. Issue. Then, the `issue` stage consumes the instructions from the ready queue and sends them to the corresponding functional units, and the data cache is accessed for memory instructions.

4. Execute. During the `execute` stage, the functional units operate and write their results into the register file. This stage is implicitly modeled in Flexim.

5. Commit. Finally, the `commit` stage retires instructions from the reorder buffer in the program order.

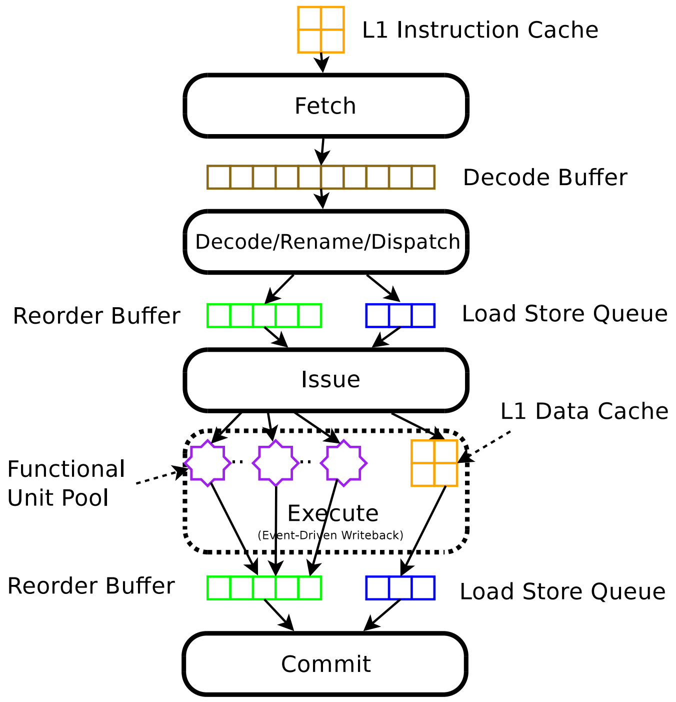


Fig. 3: Overview of the Simulated Out-of-Order Processor Pipeline

This architecture is analogous to the one modeled by the SimpleScalar tool set, but uses a decode buffer, a reorder buffer and a few physical register files to replace an integrated `Register Update Unit` (RUU). In the current implementation, the functional unit pool, physical register files, ready queue, waiting queue are shared among threads in a processor core; and the decode buffer, reorder buffer, and load/store queue are private per thread.

4.2.2 Structural Modeling

Physical Register File and Rename Table. The integer, floating point and miscellaneous physical register files (`PhysicalRegisterFile`) are modeled separately. All of them contain an array of physical registers (`PhysicalRegisters`). Each of the physical registers can be in one of four states:

1. the physical register is free (`PhysicalRegisterState.FREE`);
2. the physical register has been allocated to an instruction but has not been written to (`PhysicalRegisterState.ALLOC`);
3. the physical register has been allocated to an instruction and the value has been written (`PhysicalRegisterState.WB`);
4. the physical register is in the architectural state (`PhysicalRegisterState.ARCH`).

Physical registers are allocated during instruction dispatch, and deallocated during instruction commit and reorder buffer recovery when branch misprediction is detected.

The integer, floating point and miscellaneous rename tables are modeled separately. They maintain the current mappings of each architectural register to a physical register. Each thread maintains its own rename table because it has its own set of architectural registers.

Functional Unit Pool. The functional unit pool (`FunctionalUnitPool`) is a collection of various categories of functional units (`FunctionalUnits`). Each category has a few number of functional units (specified by `quantity`), and has a fixed issue latency (`issueLat`) and execution latency (`execLat`). Each of the functional units can be one of two states (indicated by `busy`):

1. the functional unit is free (`busy` is set to false);
2. the functional unit has been allocated to execute an instruction (`busy` is set to true).

Upon instruction issue, the functional unit pool is queried to find a free functional unit of the specified functional unit category (`FunctionalUnitType`) for a ready instruction from the ready queue. If a free functional unit is found, the instruction is issued. Otherwise, the instruction is pushed back into the ready queue.

Branch Predictors. Branch predictors provides the prediction of branch direction and targets. The branch predictors in Flexim. are modeled after Marss86. Three kinds of branch predictors (interface `Bpred`) are implemented: the bimod predictor (`BimodBpredDir`), the two level predictor (`TwoLevelBpredDir`) and the combined predictor (`TwoLevelBpredDir`). Separate branch predictors are implemented per thread.

Decode Buffer. The decode buffer is used as a FIFO queue between the instruction fetch stage and the decode/rename/dispatch stage.

Reorder Buffers. In order to maintain the correct ordering of the committed instructions, the reorder buffer is explicitly modeled as a FIFO (First-In-First-Out) buffer. Its entries (`ReorderBufferEntry`) are pushed during instruction dispatch and are popped in the program order during instruction commit.

load/store queues. Loads and store access the L1 data cache shared among threads within a core. In order to keep track of pending memory accesses, the load/store queue (`LoadStoreQueue`) is used. Separate load/store queue is maintained per thread, so that an unresolved address from one thread does not prevent loads in other threads from issuing.

4.3 Simulating Multi-Level Cache Hierarchies

A two-level cache hierarchy is modeled for the moment. Cache coherence is enforced with the directory-based MESI protocol between the private level one caches owned by each core and the level two cache that shared among cores.

Currently, constant-latency on-chip interconnect is modeled, the cycle-accurate simulation of on-chip interconnect and DRAM access is left as future work. The integration of DRAMSim with Flexim is planned.

4.3.1 Structural Modeling

Besides the data, tag, and state, a cache block has a corresponding directory entry that contains the owner and sharers information of the block. The geometry parameters of each cache in the memory hierarchy are configured via XML files. The cache subblock granularity is currently not supported but planned for future work. The LRU cache replacement policy is implemented for the moment, the implementations of more advanced policies are left as future work.

Cache. Each cache (`Cache`) can be partitioned into a number of sets (`CacheSets`), the number of sets in a cache is defined by the parameter `numSets`. And each set can be partitioned further into a few blocks (`CacheBlocks`), the number of blocks in a set is defined by the parameter `assoc` (associativity). And the cache replacement policy tries to select a victim cache block within a set to evict when the requesting cache block is being brought from the lower cache hierarchy.

Directory. Each cache maintains its own directory (`Dir`) which is a set of directory entries (`DirEntrys`). Each directory entry keeps the track of owner and sharer information for the associated cache block or subblock, depending on the availability of subblock modeling.

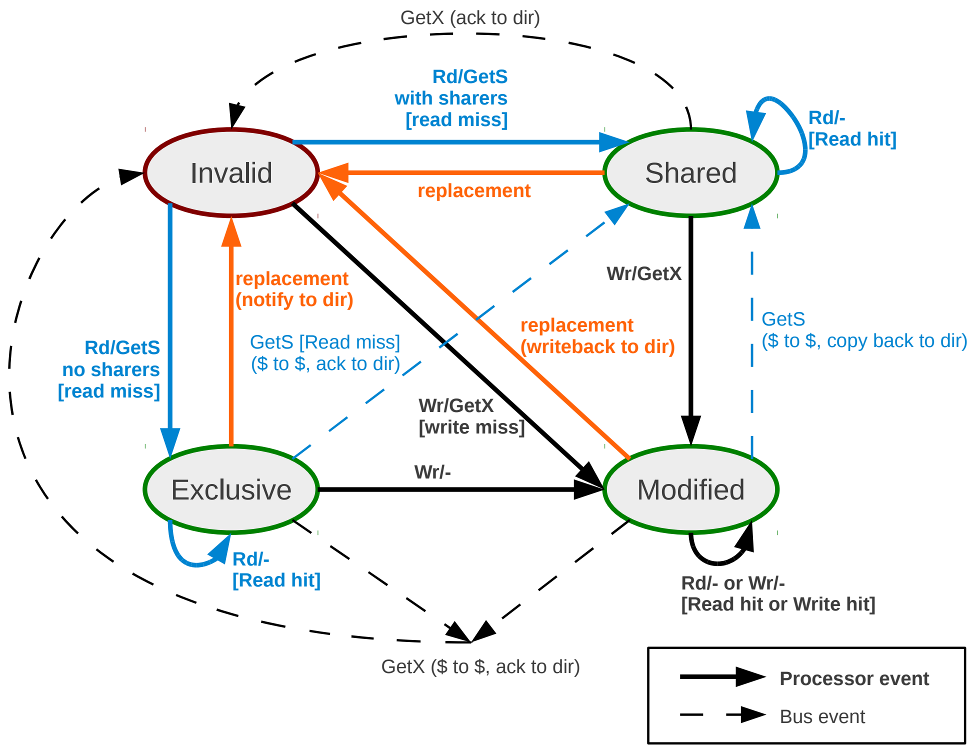

GetX (ack to dir)
Rd/GetS with shares [read miss]
Rd/- [Read hit]
Invalid
Shared
replacement
replacement (notify to dir)
Wr/GetX
GetS ($ to $, copy back to dir)
Rd/GetS no sharers [read miss]
GetS [Read miss] ($ to $, ack to dir)
replacement (writeback to dir)
Wr/GetX [write miss]
Wr/-
Exclusive
Modified
Rd/- [Read hit]
Rd/- or Wr/- [Read hit or Write hit]
GetX ($ to $, ack to dir)
Processor event
Bus event

Fig. 4: Directory based MESI Cache Coherence

4.3.2 Behavioral Modeling

Cache Coherence (Directory Based MESI as an Example). A directory based MESI cache coherence protocol running on a simple fixed-latency point-to-point interconnect is modeled for the moment. the cache block state transition diagram is shown in Fig.4.

5 GUI-Based Integrated Simulation Environment

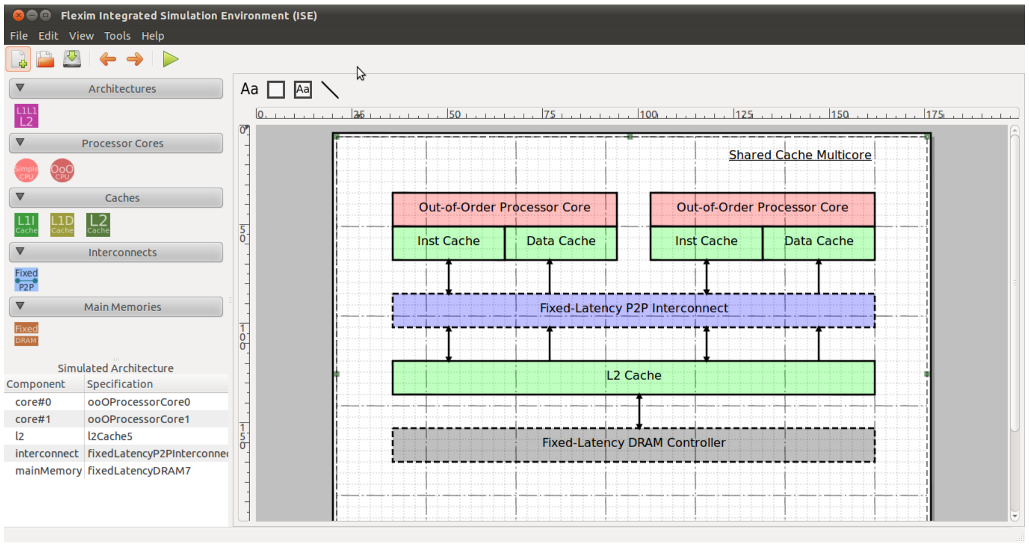
5.1 Overview

Based on the the language D's GTK binding (gtkd), a GUI-based integrated simulation environment (ISE) has been developed to make Flexim's users life easier. A typical use case of Flexim ISE is outlined as follow:

1. The user opens Flexim ISE, and draws some architectural blueprints on the canvas. Some common drawing tools such as texts, textboxes and arrows are provided in Flexim ISE.

2. The user select and assign the architectural simulation role to each desired blueprints. Blueprints themselves are not simulator configurations. Therefore, in order to make them simulatable, we need to assign them simulation roles. An overall architectural description is also needed to specify the target architecture requirements. For example, the ISE view of a shared L2 cache based dual core architecture is shown in Fig.6, in which boxes with dotted borders represent functional simulation components, such as a fixed-latency P2P interconnect and a fixed-latency DRAM controller, and other boxes represent cycle-accurate simulation components, such as two out-of-order processor cores and various kinds of caches.

3. The user composes different workload configurations which are composed of benchmarks from preset popu-



Simulated Architecture

Component	Specification
core#0	ooOProcessorCore0
core#1	ooOProcessorCore1
l2	l2Cache5
interconnect	fixedLatencyP2PInterconne
mainMemory	fixedLatencyDRAM7

Fig. 5: Flexim Integrated Simulation Environment (ISE)

lar benchmark suites, such as CPU2006, mst, MediaBench. Then assign a workload configuration to each hardware context or processor core based on whether simultaneous multithreading is enabled or not.

4. The user pushes the "start simulation" button to wait and see the simulation progress that updated periodically on the ISE's dynamic simulation view. You can now see the previously drawn architectural blueprints display their respective architectural state. The dynamic simulation view can be customized to your research needs.

5.2 Design and Implementation

6 Simulation Results

7 Conclusion

In this paper, we have detailed the Flexim multicore simulator core and its GUI-based integrated simulation environment.

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.

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.

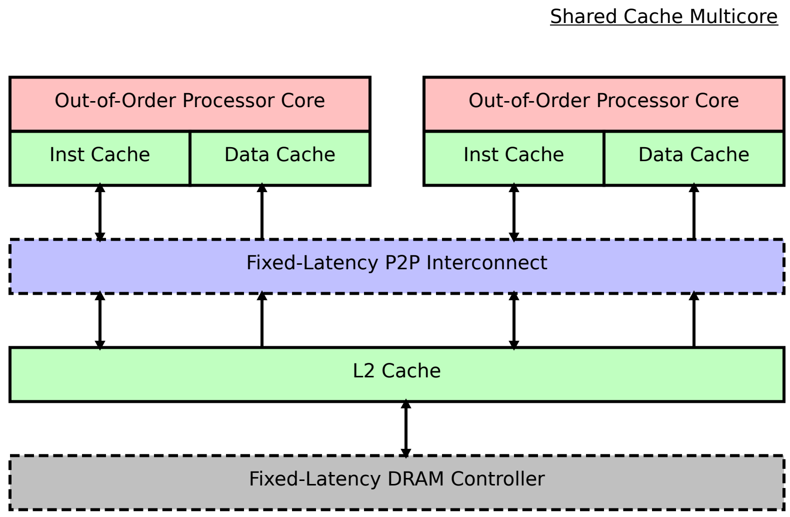

Shared Cache Multicore
Out-of-Order Processor Core
Inst Cache
Data Cache
Out-of-Order Processor Core
Inst Cache
Data Cache
Fixed-Latency P2P Interconnect
L2 Cache
Fixed-Latency DRAM Controller

Fig. 6: ISE View of a Simulated Shared L2 Cache Based Dual Core

[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.