Flexim: Facilitating User-Friendly 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. There is a permanent need of balancing speed and elegance while simulating multicore architectures.

In this paper, we present Flexim, a user-friendly cycle-accurate multicore archiectural 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 modualrity and extensibility of the simulator core. For the GUI-based ISE, Flexim uses XML files and setup wizards to ease the configuration of the simulator, and utilizes graph visualization to provide static and dynamic views of the multicore archietecture under simulation. Simulation results from a few popular benchmark suites such as Olden and CPU2006 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 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 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 study. There is a permanent need of balancing speed and elegance while simulating multicore architectures.

In this work, we present Flexim, which exploits interface-based hierarchical modularity and cycle-accurate callback-based 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 main contribution of this report 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. The idea of separation of structural and behavioral modeling of architectural components is adopted extensively during the construction of the Flexim simulator to make creating reliable simulator modules much easier than before.

    (a) Interface-based hierarchical modularity is used for enabling the construction of replaceable and composable implementations to greatly ease the users' burden during the design and implementation process of new architectural improvements.

    (b) Cycle-accurate callback-based eventing is used for modeling the behavioral facet of simulation to clarify the interaction between the execution-driven simulator and the timing simulation modules.

    (c) A common infrastructure that supports logging, eventing and XML-based IO for configuration specification and statitistcs output that fits the needs required by the simulation

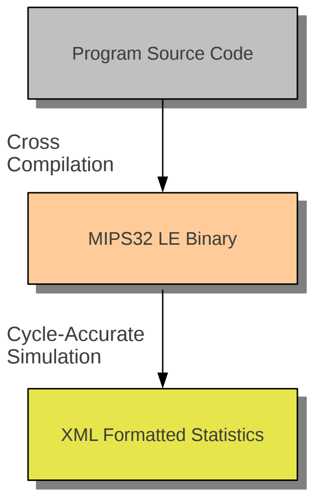

Fig. 1: Simulation Workflow

process as illustrated in Fig.1.

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 and cache hierarchies. Simulation results from some popular benchmark suites are shown in section 5. Finally, section 6 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.

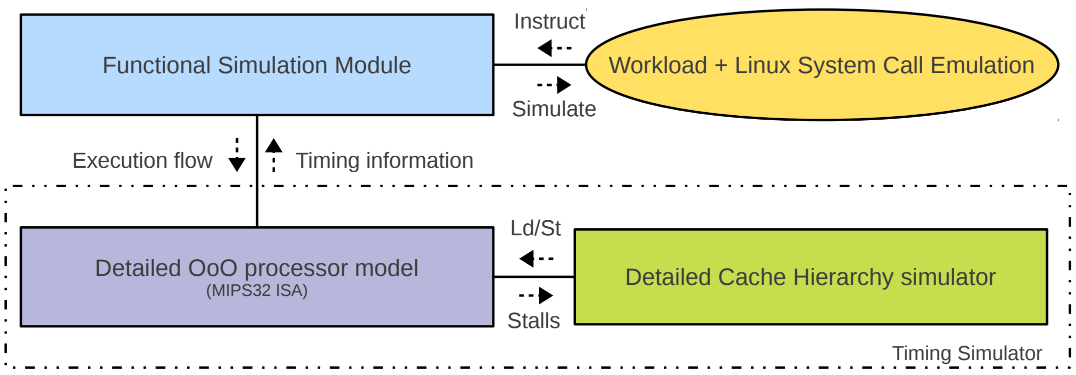
Functional Simulation Module
Instruct
Simulate
Workload + Linux System Call Emulation
Execution flow
Timing information
Detailed OoO processor model
(MIPS32 ISA)
Ld/St
Stalls
Detailed Cache Hierarchy simulator
Timing Simulator

Fig. 2: The relationship between workload, functional simulation and timing simulation.

3.3 Functional Simulation and Detailed Simulation

The relationship between Flexim modules used for functional simulation and detailed simulation are illustrated in Fig.2. 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.

3.3.1 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]

[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.

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.

3.3.2 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 Decoupled and Event-Driven Cycle-Accurate Simulation of Multicore Architectures

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.

4.1 Concept 1: Interface-Based Separation of Structural and Behavioral Modeling

The modeling of computer architecture can be classified into two types of modelings: structural modeling and behavioral modeling. Structural modeling refers to the modeling of static structural

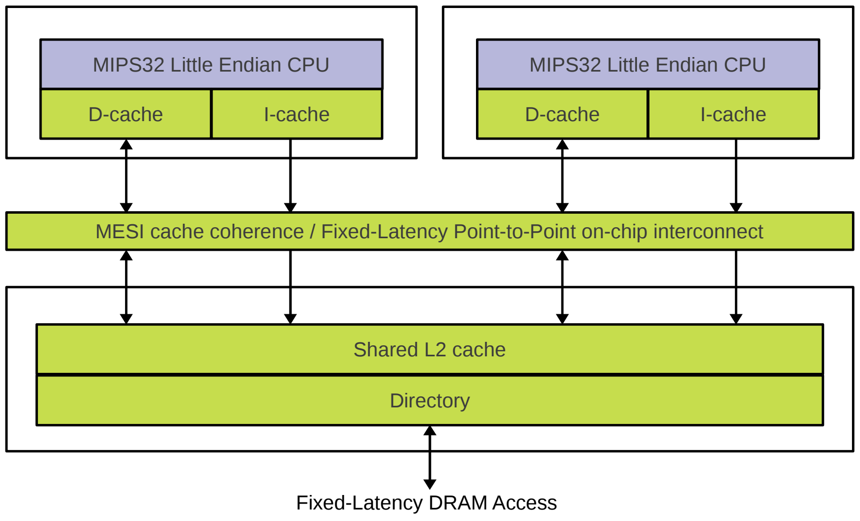

Fig. 3: Overview of the Simulated Multicore System

elements such as the internal structure of caches, whereas behavioral modeling refers to the modeling of dynamic behavioral elements such as the cache replacement policies and cache coherence protocols.

The complexity of multicore components and their interactions make it very hard to model the structural and behavioral aspects of modeling separately. Existing simulators written in traditional C or C++ programming languages inevitably makes the structural modeling elements are scattered among behavioral modeling elements. In other words, the structural modeling elements are entangled which means that its code is intermixed with code that implements behavioral modeling elements.

In order to solve the aforementioned problem of scattering and tangling of structural and behavioral aspects of architectural modeling, we use interface-based design methodology to abstract the implementation of structural elements, and use interfaces in place of concrete implementations of structural elements in the implementation of behavioral elements.

4.2 Concept 2: Cycle-Accurate Callback-Based Eventing

As explained above, most Flexim modules implement an execution-driven simulation, as SimpleScalar does. In such a model, function calls that activate some processor component (e.g., a cache or predictor) have an interface that receives a set of parameters and returns the latency needed to complete the access. Nevertheless, there are some situations where this latency is not a deterministic value and cannot be obtained in the instant when the function call is performed. Instead, it must be simulated cycle by cycle.

This is the case of interconnects and caches. In a generic topology, the delay of a message transference cannot be determined when the message is injected, because it depends on the dynamic network state. In addition, this state depends on future message transferences, so it cannot be computed unless advancing the simulation.

Because a cache access in a multithread-multicore environment may cause coherence messages transmitted across interconnection networks, the cache access latency cannot be estimated prior to the network access. In addition, the cache access latency can be affected by the internal state of accessed cache blocks, which is dynamically changed by cache writes and invalidates. Thus, the cache module is also implemented with an event-driven model. When the execution-driven simulator performs a cache access, it passes the handle of a callback method that specifies the action to be performed when the cache system completes the servicing of the cache access request.

4.3 Simulating Out-of-Order Cores

4.3.1 Behavioral Modeling

Flexim supports the mix of SMT and CMP simulation. As shown in Fig.4, 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 multi-threaded 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.

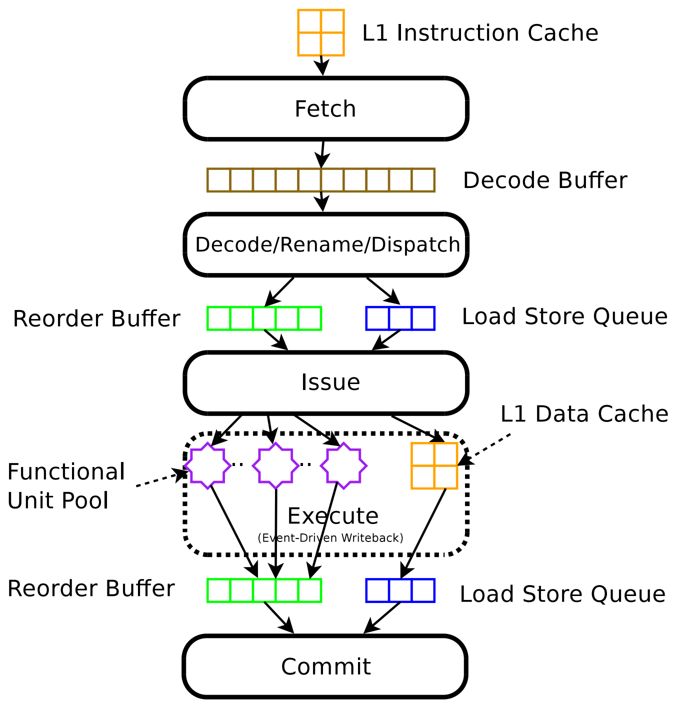

Fig. 4: Overview of the Simulated Out-of-Order Processor Pipeline

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.

4.3.2 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 funcitonal 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 commtted 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.

4.4 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.

4.4.1 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 bloks 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.

4.4.2 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.5.

5 Results

In progress.

6 Conclusion

In progress.

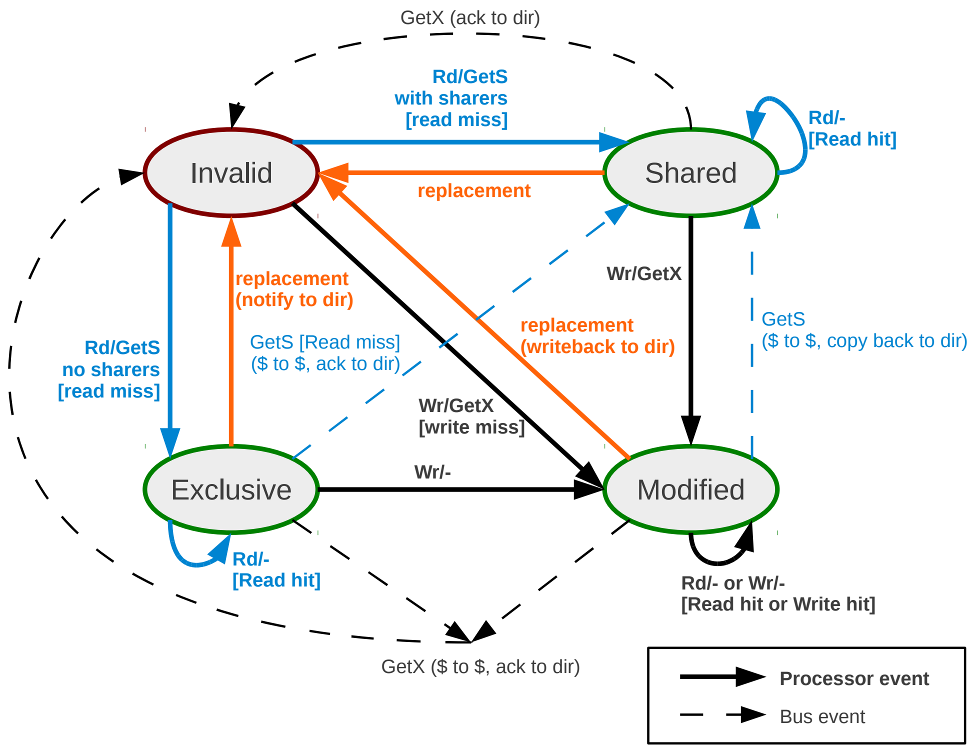
GetX (ack to dir)
Rd/GetS with shared [read miss]
Rd/- [Read hit]
Invalid
Shared
replacement
Wr/GetX
replacement (notify to dir)
GetS [Read miss] ($ to $, ack to dir)
replacement (writeback to dir)
GetS ($ to $, copy back to dir)
Rd/GetS no sharers [read miss]
Wr/GetX [write miss]
Exclusive
Wr/-
Modified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

Fig. 5: Directory based MESI Cache Coherence

- Further works
  - Functional simulation: parallel workload support

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.

[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.