Flexim: A Modular and Evolvable Simulator to Evaluate General-Purpose Multicore Processors

Min Cai, Zhimin Gu

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

Abstract

Architectural simulators are critical for successful multicore processor research nowadays. Most of the existing simulators are written in C or C++ for speed considerations, and the modeled computer structures and functionalities are intermixed, which are very difficult to maintain and evolve. Many simulators come and go with short longevity, because they can not fulfill the needs of studying fast-paced evolving computer architectures. In this paper, we take a modern approach and treat the quality of the simulator as a first-class citizen when we are constructing our Flexim simulator.

Flexim's functionalities are clearly partitioned into three categories: functional simulation, performance smulation and supporting infrastructure. It models a classic five-stage out-of-order issue processor core, directory-based two-level coherent cache hierarchy, and interfaces to external interconnection and DRAM controller simulators.

1 Introduction

Flexim[1] is an open-source, modular and highly configurable architectural simulator for evaluating emerging multicore processors. Based on the past experience and guided by a few key software engineering concepts, it is developed from scratch in the object-oriented D system programing language to obtain good maintainability, reusability and modularity of the simulator code, and advocates further extensions via plugins. It can run statically compiled MIPS32 Little-Endian (LE) programs.

1.1 Key Features

1. Architectural
   - Simulation of a classic five-stage superscalar pipeline with out-of-order execution.
   - Multi-level memory hierarchy with the directory-based MESI cache coherence protocol.
   - Support for Syscall-emulation mode simulation (i.e., application only, no need to boot an OS).
   - Correct execution of several state-of-the-art benchmark suites, e.g., wcet_bench, Olden and CPU2006.
2. Non-architectural
   - Developed from scratch in the object-oriented system programming language D 2.0. Great efforts are made to advocate software engineering practices in the simulator construction.
   - A powerful infrastructure that provides common functionalities such as eventing, logging and XML I/O.
   - Pervasive use of XML-based I/O for architectural, workload and experiment configurations and statistics.
   - Easy to use. No scripting. Only required are a statically compiled simulator executable and a few XML files.

The whole development of the Flexim simulator encompasses three main categories of functionalities: functional simulation, performance simulation and the supporting infrastructure.

The rest of this technical report is structured as follows. Section 2 focuses on functional simulation. Section 3 elaborates on performance simulation. Section 4 focuses on the supporting infrastructure. And Section 5 provides the evaluation, limitations and future work of Flexim.

[1] For the latest Flexim code, please visit the project's website on Github: http://github.com/mcai/flexim.

2 Functional Simulation

Functional simulation encompasses the abilities to load and parse MIPS binaries, decode and execute instructions, and emulate system calls.

2.1 ELF-Formatted MIPS Little-Endian Executable Loader

The Executable and Linkable Format (ELF) is a standard binary file format for Unix and Unix-like (such as Linux) systems. Each ELF file is made up of one ELF header, followed by file data. The file data can include:

- Program header table, describing zero or more segments
- Section header table, describing zero or more sections
- Data referred to by entries in the program header table or section header table

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Each byte in the entire file is taken by no more than one section at a time, but there can be orphan bytes, which are not covered by a section. In the normal case of a Unix executable one or more sections are enclosed in one segment.

In Flexim, the tasks of loading and parsing of ELF files are done through the classes `ELFReader` and `ELF32Binary` and a few supporting code elements. Each process in Flexim is associated with one `ELF32Binary` object.

2.2 Instruction Decoding and Execution

In Flexim, there are two kinds of instructions, i.e., static instructions and dynamic instructions. A static instruction represents a decoded instruction that fetched from memory, and a dynamic instruction represents a dynamically-scheduled instruction.

Below are the details of types of the implemented MIPS32 instructions.

2.3 System Call Emulation

System calls are the services that provided by the Linux kernel. To simulate MIPS32 LE/Linux applications, a few system calls are emulated for the correct execution of the whole wcet_bench benchmark suite, and mst and em3d from the Olden benchmark suite.

3 Performance Simulation

Performance simulation encompasses the detailed simulation of out-of-order processor cores and multi-level memory hierarchies.

3.1 Processor

Flexim supports the mix of SMT and CMP simulation. Some processor structures are shared among threads within a processor core, and others are private to each thread.

3.1.1 Cycle-Accurate Modeling of Pipeline Structures

Flexim explicitly models the Reorder Buffer (`ReorderBuffer`), the Issue Queue (`IssueQueue`), the Load/Store Queue (`LoadStoreQueue`), separate integer, floating-point and miscellaneous register files, register renaming, and the associated rename table.

The reorder buffer is modeled as a FIFO buffer. Its entries are pushed during instruction `dispatch()` and are popped during instruction `commit()`.

The issue queue is modeled as an array of issue queue entries. Each issue queue entry (`IssueQueueEntry`) can be in one of two states: either free(`IssueQueueEntryState.FREE`) or already allocated(`IssueQueueEntry.ALLOC`). Its entries are allocated at instruction `dispatch()` and are released at `issue()` and on branch mispredictions on `recoverReorderBuffer()`.

The integer, floating point and miscellaneous physical register files are modeled separately as `intRegFile`, `fpRegFile` and `miscRegFile`, respectively. All of them contain an array of physical registers(`PhysicalRegister`). Each of the physical registers can be in one of four states:

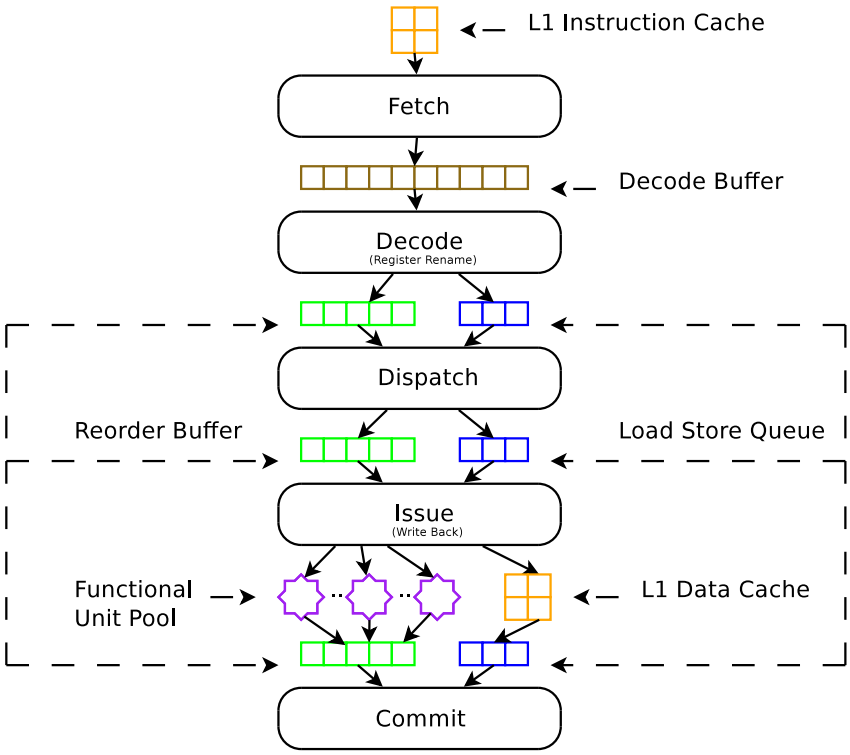

Fig. 1: Overview of the Simulated Out-of-Order Processor Pipeline

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

Physical registers are allocated at the `dispatch()` stage, and deallocated at `commit()` and during branch mispredictions in `recoverReorderBuffer()`.

The integer, floating point and miscellaneous rename tables are modeled separately. They maintain the current mappings of each architectural register to a physical register.

3.1.2 Details of the Processor model and Implementation

Threads within a core maintain separate program counters, but share the fetch unit and icache. Threads within a core share the available bandwidth in the front end including fetch, decode and rename. The time slice based fetch policy is implemented for the moment, more advanced policies such as icount is left for future work. Separate branch predictors are implemented per thread.

Each thread maintains its own rename table because it has its own set of architectural registers. After renaming, instructions from all threads are dispatched into the shared issue queue.

In the issue queue, instructions from all the threads participate in the instruction `wakeup()` process and compete for the issue bandwidth in `selection()`. Instructions that are selected for issue continue to perform register file accesses. All the physical register files are shared among the threads. After register file accesses are done, instructions begin execution on the functional units, which are also shared among the threads.

Loads and stores access the shared dcache among the threads within a core. In order to maintain the correct orderings of 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.

After execution, instructions write back to the register files. Commitment (or retirement) is done in order for each thread out of the re-order buffers (`ReorderBuffers`). Separate reorder buffers are maintained per thread.

3.2 Memory Hierarchy

3.2.1 Internal Cache Structure and Cache Coherence

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.

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.

3.2.2 On-Chip Interconnect

Currently, constant-latency on-chip interconnect is modeled, the cycle-accurate simulation of on-chip interconnect is left as future work.

3.2.3 Interface to External DRAM Simulators

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

4 Supporting Infrastructure

There are various supporting modules aside the aforementioned main components to maintain the Flexim's reusability.

4.1 Eventing and Callback Mechanisms

In flexim, cycle-accurate simulation is driven by event signals per cycle. Generally speaking, an event can be any piece of code that is scheduled to execute at the specified time. This piece of code can be represented as a delegate in the D language. The delegate-based event queue (`DelegateEventQueue`) is illustrated as below:

```
class DelegateEventQueue: EventProcessor {
  this() {
  }

  void processEvents() {
    if(currentCycle in this.events) {
      foreach(event; this.events[currentCycle]) {
        event();
      }
      this.events.remove(currentCycle);
    }
  }

  void schedule(void delegate() event, ulong delay = 0) {
    this.events[currentCycle + delay] ~= event;
  }

  void delegate()[][ulong] events;
}
```

The delegate-based callback mechanism is used extensively in the implementation of the MESI cache coherence protocol. This way, the code is clean and readable.

4.2 Categorized Logging Mechanism

The logging component supports configurable logging functionalities that can facilitate development and even be useful after release. The code skeleton of the Logger class is shown as below:

```
class Logger {
  static this() {
    singleInstance = new Logger();
```

```
  }

  bool enabled(LogCategory category) {
    return category in this.logSwitches && this.logSwitches[category];
  }

  string message(string caption, string text) {
    return format("[%d] \t%s%s", currentCycle,
      caption.endsWith("info") ? "" : "[" ~ caption ~ "] ", text);
  }

  void infof(LogCategory, T...)(LogCategory category, T args) {
    debug {
      this.info(category, format(args));
    }
  }

  void info(LogCategory category, string text) {
    debug {
      if(this.enabled(category)) {
        stdout.writeln(this.message(category ~ "|" ~ "info", text));
      }
    }
  }
...

  bool[LogCategory] logSwitches;

  static Logger singleInstance;
}
```

4.3 XML-Based Input/Output for Configurations and Statistics

Extensible Markup Language (XML) is a standard and pervasive mechanism for representing structural data in a machine-readable and human-friendly way. Here XML-based configuration specification and statistics output are provided. For example, the following code shows how to serialize and deserialize a cache configuration using the provided XML I/O support:

```
class CacheConfigXMLSerializer: XMLSerializer!(CacheConfig) {
  override XMLConfig save(CacheConfig cacheConfig) {
    XMLConfig xmlConfig = new XMLConfig("CacheConfig");

    xmlConfig["name"] = cacheConfig.name;
    xmlConfig["level"] = to!(string)(cacheConfig.level);
    xmlConfig["numSets"] = to!(string)(cacheConfig.numSets);
    xmlConfig["assoc"] = to!(string)(cacheConfig.assoc);
    xmlConfig["blockSize"] = to!(string)(cacheConfig.blockSize);
    xmlConfig["hitLatency"] = to!(string)(cacheConfig.hitLatency);
    xmlConfig["missLatency"] = to!(string)(cacheConfig.missLatency);
    xmlConfig["policy"] = to!(string)(cacheConfig.policy);

    return xmlConfig;
  }

  override CacheConfig load(XMLConfig xmlConfig) {
    string name = xmlConfig["name"];
    uint level = to!(uint)(xmlConfig["level"]);
    uint numSets = to!(uint)(xmlConfig["numSets"]);
    uint assoc = to!(uint)(xmlConfig["assoc"]);
    uint blockSize = to!(uint)(xmlConfig["blockSize"]);
    uint hitLatency = to!(uint)(xmlConfig["hitLatency"]);
    uint missLatency = to!(uint)(xmlConfig["missLatency"]);
    CacheReplacementPolicy policy = cast(CacheReplacementPolicy) (
      xmlConfig["policy"]);

    CacheConfig cacheConfig = new CacheConfig(
      name, level, numSets, assoc, blockSize, hitLatency, missLatency, policy);

    return cacheConfig;
```

```
  }

  static this() {
    singleInstance = new CacheConfigXMLSerializer();
  }

  static CacheConfigXMLSerializer singleInstance;
}
```

4.4 Plotting and Table Generation for Experiments

(Pre-release, Documentation-in-Progress)

5 Evaluation, Limitations and Future Work

(Pre-release, Documentation-in-Progress)

5.1 Benchmark Evaluation

(Pre-release, Documentation-in-Progress)

5.1.1 Criteria

(Pre-release, Documentation-in-Progress)

5.1.2 Results

(Pre-release, Documentation-in-Progress)

5.2 Comparison to Other Simulators

(Pre-release, Documentation-in-Progress)

5.2.1 Results

(Pre-release, Documentation-in-Progress)

5.3 Limitations and Future Work

Here are some of the limitations of Flexim. This list is not intended to be comprehensive, but rather aims to provide the user with an initial understanding of the capabilities of the Flexim simulation environment. These capabilities are subject to change in future releases.

- Only statically compiled MIPS32 binaries are supported by Flexim. Multi-ISA support and loading support of dynamically compiled programs are remained as future work.
- The explicit modeling of Miss Status Holding Registers (MSHRs) is not implemented and left as future work.
- Cycle-accurate on-chip interconnect and dram controller modeling is not implemented and left as future work.