SMASim: A Cycle-accurate Scalable Multi-core Architecture Simulator

Jari-Matti Mäkelä and Ville Leppänen *

Abstract— **The computer industry has tried to mitigate the problem of achieving computationally more efficient hardware on three fronts: increasing the execution speed by increasing the operating frequency, decreasing the amount of required time to issue a single instruction by enhancing *instruction level parallelism* (ILP), and increasing the "computational volume" by adding more computational units. Developing complete physical architectures or even experimental FPGA prototypes has turned out to be expensive and to require relatively great amount of resources. Software archtitecture simulators are seen as an efficient way of lowering these costs.**

SMASim is a software based simulator, motivated by an experimental moving threads architecture that attempts to lower the costs of rapidly designing new architectures. It is based on a general purpose, cycle-accurate event-driven message passing framework between the described hardware architecture elements. Its relatively simple cost model captures the essential properties of many hardware designs. The simulator's design allows easy monitoring of the system and provides execution performance comparable to other cycle-accurate hardware simulators.

The focus on SMASim has been to speed up declaring new target architectures with expressive domain specific notations and to decrease the amount of design errors with the static type checker of the implementation language. As a result, the implementation supports modular architecture descriptions on various granularity levels. A graphical user interface is provided to simplify the task of modifying parameters of a simulated system and to provide interactive feedback.

Keywords: multi-core, simulator, cycle-accurate, domain specific language

1 Introduction

As the performance requirements of modern software have steadily increased, the computer industry has faced a constant need for more efficient hardware to perform these tasks. Previous generations of microchips have tried to mitigate the problem on three fronts: (i) increasing the amount of executed instructions per time unit by increasing the operating frequency, (ii) decreasing the amount of required time to issue a single instruction (CPI) by enhancing *instruction level parallelism*, and finally (iii) growing the "computational volume" by increasing amounts of computational units.

During the last few years the first two approaches have reached the point of diminishing returns and the operating frequencies have stabilized. Complex ILP mechanisms begin to occupy on-chip space in ways that make multi-core solutions appear more feasible. The amount of parallelism on instruction level is also limited by data dependencies. Another physical limitation is the power dissipation of the processor, which has increased over the years and pushed the requirements for efficient cooling system even further.

The third approach has started a new era of multi-core and multi-processor parallel programming. Several vendors have provided their first revisions of multi-core architectures. Efforts that stand out from the traditional *von Neumann model* seem to yield most promising performance improvements. Developing new hardware architectures has turned out to be expensive and requires relatively great amount of resources even when building experimental FPGA prototypes.

SMASim is a pure software based simulator, motivated by an experimental multi-core moving threads architecture [9] that is based on a virtual *parallel random access machine* (PRAM) [3, 4] model and differs from contemporary designs regarding the thread abstraction and handling. It attempts to lower the costs of rapidly designing new architectures by supporting new multi-core architectures at various abstraction levels. The software and its component libraries were originally built as a rapid development tool for the architecture description and for measuring its performance. The modular design was later extended to also support other kinds of architecture models.

The simulator uses a cycle-accurate event-driven simulation core. Its relatively simple cost model captures the essential properties of many hardware designs. The simulator's design allows easy monitoring of the system

*This research has been funded by the Academy of Finland project numbers 128729 and 128733. University of Turku, Department of Information Technology Joukahaisenkatu 3-5 B, 20520 Turku, Finland; E-mail: {jmjmak, ville.leppanen}@utu.fi

and provides execution performance comparable to other cycle-accurate hardware simulators with high accuracy typical to the cycle-free simulators.

The simulator is implemented in hybrid object-functional language Scala[7]. The flexibility of Scala allows using more declarative domain specific notation when specifying parts of the architecture, yet provides static verification of the model via a strong type system. The name SMASim has two meanings — the simulation is built in Scala and also adapts the Scala's idea of a scalable language. Components built on SMASim have the flexibility of a modern general purpose language and can be implemented in a declarative manner.

The core framework and building blocks of the simulator are tuned for high performance and to minimize memory allocations. However due to limitations of Java virtual machine and strict sequential cycle-accurate execution, some performance is sacrificed for other high level features.

2 Related work

Architecture simulators are a widely studied subject. The implementations can be roughly split into two categories, based on their focus on functionality or performance. Most of the existing work encompasses a simulator core built in some high-performance, low-level systems programming language and a target architecture description either written in some declarative markup language or using the same core language. Typically, the simulators have mainly focused on existing commercially available single-core architectures.

2.1 Cycle-accurate simulators

The list of traditional cycle-accurate simulators is long. Probably the best known simulator is the SimpleScalar [1] project, which has already reached its fourth version. SimpleScalar allows defining single-core system with a definition language. Another widely used simulator is SESC [8], which uses C++ to model the architecture. PTLSim [10] and Bochs [6] simulate the x86 architecture.

2.2 New generations

More recently, the focus has been on more efficient simulations that either sacrifice the cycle-accuracy by introducing some statistical measurement based on random sampling or implement advanced techniques for speeding up the event based system.

For instance, FaCSim [5] uses three methods to increase its speed: chunked pipeline events, caching of decoded instructions, and parallelization of the simulator. A lesser known simulator, tsim, takes advantage of a mechanism called two-phase trace-driven simulation (TPTS) [2]. This technique may bring two orders of magnitude larger simulation speed and shrink the memory consumption over 90%.

2.3 Implementation

The structure of SMASim is divided into independent subsystems which are described in more detail below.

2.4 Message passing core

The core of SMASim is an event-driven message passing system. Each relevant entity on various abstraction levels in the simulated target architecture correspond to an object in the architecture model. A set of event handlers is associated with each active entity in the system. The purpose of the handlers is to control the dynamic data and control flow via data dependencies and event latencies.

The simulator's operation is execution driven. It provides a cycle-accurate simulation of the whole architecture. The runtime behavior is exact and emulates the actual hardware, although at a loss of runtime performance. The runtime cost is proportional to the amount of simultaneous active messages, not to the component count of the simulated system. Recent studies [2] and [5] have shown that the execution speed could be improved drastically by parallelizing the simulation code, using statistical sampling, and dynamic translation. Since SMASim still requires time to mature as a technology, we have focused more on its flexibility.

The distinguishing feature of SMASim is to leverage (i) the implementation language's type system to enforce modularity, safety, and reliability of the simulation and (ii) the language's flexible syntax to build a domain specific language notation to improve readability and productivity.

The event handlers resemble pure functional building blocks with well-defined input and output interfaces. The type checker can thus statically guarantee the lack of incompatibilities between components. The handler's abstract interface allows arbitrary connections between the components. Although the simulator's core only provides a simple DSL-like notation for the event handlers, the target architecture's description language can be extended with new notations provided by the subsystems described in the next section.

The core architecture supports pluggable runtime monitors. The mechanism follows the C language's tradition of only paying for what one actually uses. In other words, this allows flexibly adjusting the amount of runtime checks made and statistics gathered during the simulation. The monitoring framework is discussed in more detail in Section 2.6.

To illustrate the simplicity of composing architecture de-

scriptions, the following example in Figure 1 presents the construction of a 4-port logical AND gate from three 2-port AND gates defined in the same example. The example also discreetly reveals how components implicitly connect to the event system for convenience. The corresponding code follows in Algorithm 1.

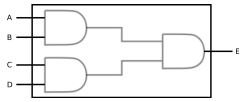

Figure 1: Construction of a 4-port logical AND gate from three 2-port AND gates.

Algorithm 1 Implementation of the 4-port logical AND gate from three 2-port AND gates.

```
implicit val clock = ControllerClock @> 100 Hz

class LogicalComponent extends SimpleComponent {
  type B = Boolean
} // also implicitly connects to the clock above

class AND extends LogicalComponent {
  def execute = value { (A: B, B: B) ⇒A & B }
}

def createAND = new AND execute

class Four_Port_AND extends LogicalComponent {
  def execute = fun { (a: B, b: B, c: B, d: B) ⇒
    ((a,b) ⇒: createAND,
     (c,d) ⇒: createAND) ⇒: createAND
  }
}
```

2.5 Subsystem libraries

The simulator is accompanied by a set of subsystem libraries. The current set of libraries includes basic components for building CPUs and their internals, multi-level memory hierarchies with cache units, black-box interprocessor networks, and a subset of MIPS32® instruction set, extended with moving threads architecture specific thread fork/join instructions.

The driving force behind the provided libraries and the whole simulator has been the goal of implementing a software simulator for a novel moving threads based multicore architecture [9]. As both, the architecture and the simulator, mature, the libraries will be suplemented with a larger set of general purpose components.

CPU subsystem The CPU subsystem contains functionality for decoding and encoding instruction streams and loading binary images from the file system. It also provides components for defining simple CPUs, instruction sets, and registers files. Ideally the framework provides all low-level facilities for loading programs, and the responsibility of choosing an instruction set and execution pipeline configuration is delayed to the target architecture description.

Memory subsystem The memory subsystem provides components for modeling various kinds of basic memory hierarchies. Currently supported are memory components with a fixed latency property, memory banks comprising other storage components, simple multi-level directly mapped or N-way associative cache components. Memory components often have a differing operating frequency, which is naturally supported via the clock system of the framework.

As an example of memory component usage, the code listing in Algorithm 2 shows an interface used when constructing cache components.

Algorithm 2 N-way set associative cache constructor interface.

```
class CacheNWaySetAss(
    fetchLatency: Int,
    signalingLatency: Int,
    cacheSize: Int, // in bits
    rowSize: Int, // in bits
    setSize: Int, // in bits
    maxBlockSize: Int,
    concurrentReadCount: Int,
    concurrentWriteCount: Int,
    concurrentReadWriteCount: Int,
    source: SignalingMemoryComponent
) extends SignalingMemoryComponent
```

Network subsystem The network subsystem is much simpler than the other subsystem. The reason is that the network is currently modeled only as a black-box in the moving threads architecture description. The subsystem thus provides a simple buffered network nodes and a star shaped network topology with a constant latency property and broadcast functionality.

MIPS32 subsystem As an extension to the CPU subsystem, the MIPS32® subsystem provides a framework for defining or extending MIPS-like instruction sets with the help of a statically constrained domain specific language. Also encoders, decoders, and full semantic definitions are provided for a set of 69 core arithmetic, logic, and memory instructions. The set currently lacks signaling, atomic and floating point operations.

The instructions use a five stage pipeline abstraction for semantics and the actual implementation is left open.

The provided abstraction makes other RISC instruction sets or subsets of MIPS32® trivial to implement. As an example, the Algorithm 3 clarifies the pipeline connection with imaginary pipeline semantics. A set of instruction mnemonics and an assignment operator (:=) are given to encourage the use of domain specific notations and to cut down the number of potential subtle bugs.

Algorithm 3 Instruction template

```
object X extends InstructionGenerator(TypeICmd(id) {
  override def loadMemory = (Address(1), 4, _:Int ⇒...)
  override def loadRegisters = List(rs ⇒rs, ..., hi ⇒hi)
  override def execute {
    rt := pc + rs
  }
  override def storeRegisters = List(rt ⇒rd, ..., lo ⇒lo)
  override def storeMemory = (Address(2), 4, 65535)
})
```

A more practical example is given in Algorithm 4. First, a logical AND operator is defined and later two trivial instruction sets are constructed from the instruction.

Algorithm 4 AND instruction and the associated instruction set.

```
object AND extends InstructionGenerator(
  new SpecialCommand((4 << 3) + 4) {
    override def execute = rd := rs & rt
})

object AddInstructionSet extends InstructionSet(AND)

val ZeroInstructionSet = AddInstructionSet − AND
```

Serializing and deserializing As an example of instructions deserialization, the simulator ships with a minimalistic simple disassembler, inspired by the GNU objdump. Algorithm 5 shows the main steps modulo error handling required to build a similar tool. Additionally, the tool also re-encodes the instruction stream and writes it to another file.

Algorithm 5 Instruction serialization and deserialization

```
import core.instructions.InstructionEncoder
import arch.mips32.MIPS32Decoder

val program = MIPS32Decoder load "/tmp/input.bin"
val insStream = MIPS32Decoder decode program
val program2 = InstructionEncoder encode insStream
InstructionEncoder save ("/tmp/output.bin", program2)

for (i ←instructionStream)
  println(i)
```

2.6 Monitoring framework

The modular clock system of the simulation framework allows plugging arbitrary amount of runtime monitors to the message passing core. The feature can be used to construct a trace of events executed in the system, which can possibly played back at some point later. However, it can also be used by higher level components to extract very specialized data from the stream of events.

The library provided features of the simulator are accompanied with correnponding monitoring subsystems. This way the monitoring is implicitly available for each subsystem, but for performance reasons has to be explicitly enabled during the initialization of the simulation. The monitors can also be connected to the user interface for interactive and real-time execution monitoring. The rationale behind the existing monitors was to ease the evaluation of the simulated architecture and to better tune the CPU parameters. The monitors also support command line output.

Execution monitor A very basic monitor for monitoring the execution of various commands is provided in the monitoring framework. The mechanism is low level and is mostly useful for constructing other monitors or debugging the low level operation of the framework or the system used in the simulation. By default it just accumulates a textual execution history based on the captured events

The monitor works by connecting to a controller clock as a proxy. The simulation is told to use the monitor in place of the original controller clock. Most monitors can be chained in this fashion as if they were fully transparent. A logical setup with one execution monitor is described in Figure 2.

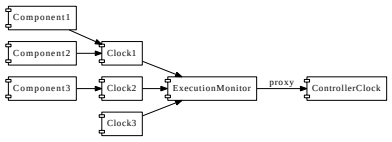

Figure 2: Simple execution monitor setup.

Derived monitors The message passing system forms the main communication infrastructure inside the simulator. It separates functional units (components) from each other. One can always build custom mechanisms for monitoring programs, but the generic execution monitor interface often suffices. One advantage of the built-in monitoring system is that it is pluggable and thus also separated from other concerns in the simulator's design.

The execution monitor is triggered by three kinds of events: clock ticks, executable enqueuing and execution.

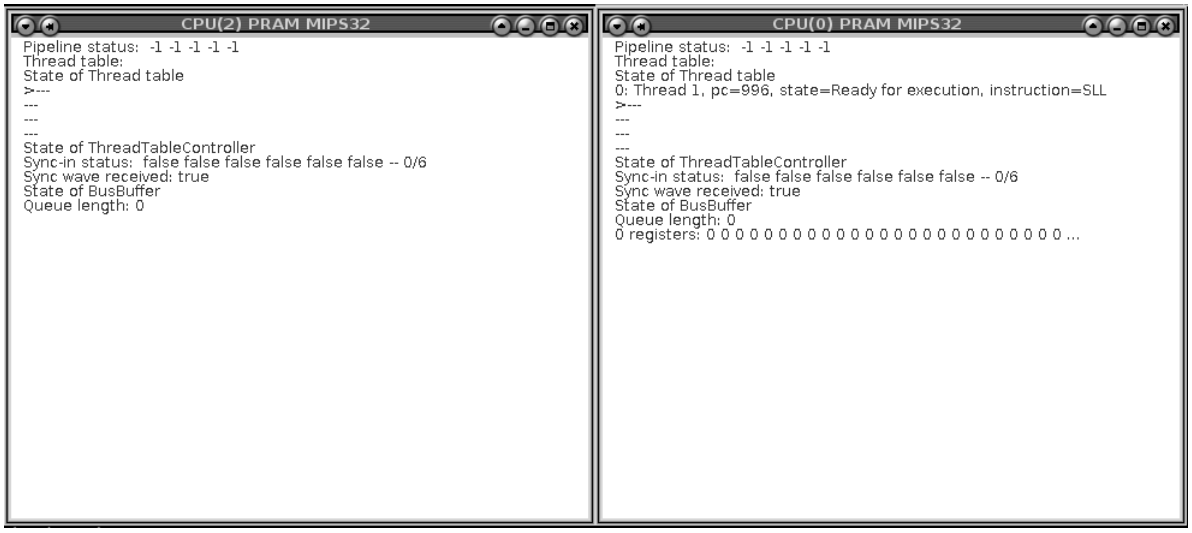

Figure 3: Core monitor for the moving threads architecture.

If we want to monitor a simulated component, the idea is to use this interface to find changes in the state of the system. The component model should be fine-grained enough for the monitoring system to be able to capture relevant information during the simulation. The components can also comprise internal state that can be monitored in this context.

Usually the tick interface is used to determine, how many cycles have elapsed since the last timestamp or since the beginning of the simulation. On the other hand, pattern matching can be used to pick messages relevant to the monitor from the stream of new events.

Cache monitor The cache monitor can be attached to a memory cache unit. During the simulation the cache monitor collects information about cache accesses, hits, and misses. Cache hit ratio and latency characteristics are computed from the monitored data at any point of time. Cache statistics are also collected from the beginning of the simulation until the last status query. All caches in the target architecture connected to the controller clock will be monitored and the output will group the cache events by a cache name.

Network monitor A network monitor traces the packet traffic on the inter-processor network or more generally any kind of data moving within a generic network abstraction. The latencies of packet transmissions and amount of packets and bytes transferred from node to node are being monitored. All networks implementing the network abstraction connected to the controller clock will be monitored and the output will group the network traffic by a network name.

Core monitor For the moving threads architecture, the simulator also provides an experimental general core monitor. The monitor simultaneously shows the state of all internal registers, the threads in execution, in thread queue and the thread packets in various inter-processor network buffers. A snapshot of the simulation state via the core monitor in an imaginary 5-core system is shown in Figure 3.

2.7 Graphical user interface

The simulator can be controlled with a graphical user interface. The default user interface is drawn using the Java's Swing framework. The interface provides functionality for:

- configuring the various subsystems in the architecture model,
- loading executable binaries from the file system,
- controlling and exporting the simulation state,
- observing the state of the subsystems, the simulation trace, and the gathered statistics, and
- miscellaneous utility functions such as disassembling the binaries or displaying instruction sets.

For easy customizability, all user interface components can be extended and replaced with the expressive langauge. The framework also provides a set of helper classes for constructing new user interfaces for the 3rd party subsystems and for controlling the target architecture models.

Controllers The simulator can be controlled with the controller frames shown in Figure 4. Subsystem functionality can be added to the controllers with a single line of code. The default controls include controls for resetting and stepwise advancing of the simulation. A persistent snapshot feature of the simulation state is also planned.

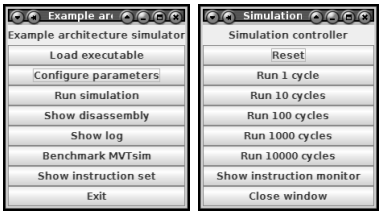

Figure 4: Controller frames for the simulation.

Configuration The user interface can easily be connected to any ad-hoc configuration data structure. The feature encourages the use of parameterized architecture model since only a single line of code is required to make all related configurable parameters visually tunable via the GUI and to connect them to a persistent storage for later use. A preliminary version of the configuration data for the moving threads architecture is shown in Figure 5.

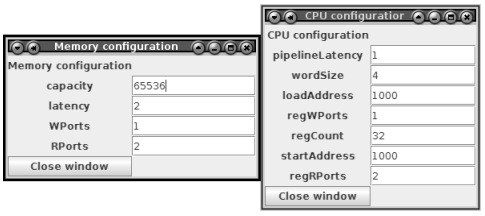

Figure 5: Simulator configuration frame.

3 Future work

SMASim 's main goal is to provide a CPU simulator for the evaluation of the aforementioned moving threads architecture. Since the purpose of the architecture's simulator is to validate the operation of the model and to produce meaningful test results from various multi-core benchmark, the goal can be restated as having (i) a more complete library of general purpose and moving threads architecture components, (ii) more detailed monitoring capabilities for extracting more relevant runtime datal, and (iii) improved execution speed because a larger benchmark suite may take drastically longer to execute than on the most modern simulation engines.

4 Conclusions

So far SMASim has remained as one of the only simulators that use the target language to express the target architecture model. Although SMASim loses in speed to the latest simulator inventions, it remains unique due to the hybrid solution that allows combining both functioal and object oriented features of the simulator's implementation language to form a modular and verified target architecture model.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, pages 59–67, 2002.

[2] Sangyeun Cho, Socrates Demetriades, Shayne Evans, Lei Jin, Hyunjin Lee, Kiyeon Lee, and Michael Moeng. Tpts: A novel framework for very fast manycore processor architecture simulation. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 446–453, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[4] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[5] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S.Y. Han. FaCSim: a fast and cycle-accurate architecture simulator for embedded systems. *ACM SIGPLAN Notices*, 43(7):89–100, 2008.

[6] D. Mihocka and S. Shwartsman. Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure.

[7] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[8] PM Ortego and P. Sack. SESC: SuperESCalar Simulator, February 2007.

[9] J. Paakkulainen, J-M Mäkelä, V. Leppänen, and M. Forsell. Outline of risc-based core for multiprocessor on chip architecture supporting moving threads. In *CompSysTech '09: Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, pages 1–6, New York, NY, USA, 2009. ACM.

[10] MT Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. Performance Analysis of Systems and Software, 2007. ISPASS 2007. In *IEEE International Symposium on*, pages 23–34, 2007.