MVTsim - Software Simulator for Multicore on Chip Parallel Computer Architectures

Jari-Matti Mäkelä, Jani Paakkulainen and Ville Leppänen

Abstract: *Designing a parallel computer architecture for the multi-core on chip environment involves a lot of architectural design issues. Actual hardware design based on ASIC and for demonstrational purposes on FPGA is very expensive method to study the cost of various design choices. Therefore, we have developed a software based simulator MVTsim for multi-core on chip parallel computers. Due to a special theme of our research project, the simulator is oriented towards supporting a very fine-grained moving threads approach.*

We describe the general software architecture of the multi-core on chip simulator. In the simulator, special emphasis has been put on concisely expressing the problem domain with a general purpose language, the modular structure of the simulator and target architecture, support for flexible combining of different granularity levels of components, and modelling relevant physical delay related properties. We demonstrate our simulator by describing a RISC-based configuration supporting the moving threads approach, and give some initial results concerning running actual programs with our simulator.

Key words: *Moving Threads, Multiprocessor on Chip Architecture, Cycle-accurate simulation.*

1 INTRODUCTION

Speeding up computers by increasing their clock rate is a development trend that has continued for decades. As the heating as well as general lithography based problems have become rather serious issues during the recent years, the processor manufacturing industry seems to be heading for multi-core architectures instead of aiming at higher and higher clock rates. Several kinds of multi-core processors have been presented by all the major processor manufacturers. However, an emerging trend has been that programming the contemporary multi-core processors is seen very problematic. The main reason is perhaps that the hardware designers have created a hardware environment, which is difficult to program efficiently. The programmers simply would need to consider too much of the low level architectural details when writing programs (e.g. how to optimize the usage of caches).

Completely new kinds of approaches for parallel computers have been proposed already a long time ago in the form of Parallel Random Access Model (PRAM) [6, 7]. Implementation of PRAMs was extensively studied in the 1990's [9], and there was even the SB-PRAM project [1, 8] providing an experimental hardware implementation.

Recently, such PRAM implementations have been studied in the multi-core on-chip context by Forsell [3, 4] (Eclipse architecture) and by Vishkin et al [14] (Paraleap architecture). In our MOTH[1] (Moving threads realization study) project we have considered an approach for multi-core on-chip architectures, where the programming model is based on the PRAM. The PRAM approach abstracts away all kinds of mapping problems by providing a shared memory abstraction with unit (amortized) access cost. It also provides flexible expression of threads as well as strong synchronous execution of different threads. These properties together provide an easy-to-program approach for programming multi-core systems, but at the same time ask for efficient implementations of the PRAM approach.

We have invented a new kind of approach for mapping the computation of an application to MP-SOC architectures [5] (some preliminary ideas appear in [9, 10]). Instead of moving data read and write requests, we move extremely lightweight threads between the processor cores. Each processor core is coupled with a memory module and parts of each memory module together form a virtual shared memory abstraction. Applications are written using a high-level language based on shared memory. As a consequence of moving

1 This research has been funded by the Academy of Finland project number 128729

threads instead of data we avoid all kinds of cache coherence problems. Another advantage is flexible and efficient creation of new threads. In our architecture, the challenge of having efficient implementation of an application reduces to mapping the used data so that the need to move threads is balanced with respect to the bandwidth of the communication lines. Writing an application to use lots of threads is rather easy (due to rich literature of parallel algorithms using the shared memory abstraction). This method also eliminates the need for separate reply network and introduces a natural way to exploit locality without sacrificing the synchronicity of the PRAM model.

Previous work on software based simulators covers various kinds of designs, with focus, among others, on performance, ease of development, and tool support. As we evaluated a wide range of available tools such as the XMTsim project (simulating the Paraleap architecture [14]), SESC [12], and QEMU [2], it occurred that a simulator combining the ideas of architecture independence, cycle-accurate precision, ease of use, modularity, and support for varying granularity levels, yet without sacrificing the general purposeness of a programming language was needed. MVTsim is the result of these ideas, implemented in a hybrid object-functional language Scala, extended with the domain specific constructs of the simulator framework. As performance was a secondary design goal, possible performance issues will be considered as the design stabilizes.

2 SOFTWARE ARCHITECTURE FOR SIMULATOR

2.1 General structure

On a high abstraction level, and from the user's point of view, it is convenient to see the simulator as a message passing system, extended with the domain specific constructs of the target architecture. Most active objects modelled with the framework derive from `Executor` or `Executable`. As the simulated devices often share common functionality, classes `Component` and `Command` extend these interfaces respectively, exposing a practical platform with command queues and event driven functional message passing.

Miscellaneous utility classes, such as the clock classes emitting clock ticks that trigger commands, form the third category of classes. Each active component is connected to some clock. Each clock has an independent frequency, but a shared controller clock is used to coordinate the execution in a globally synchronized stepwise manner. Figure 1 visualizes the class relations when implementing a simple target architecture.

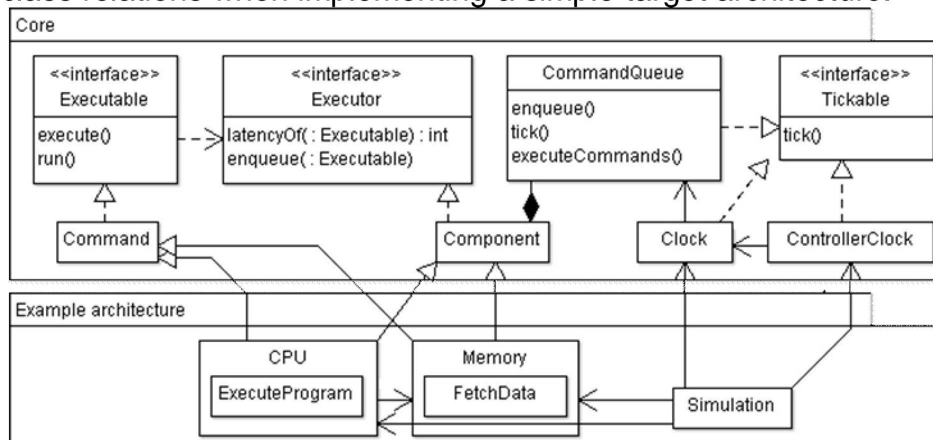

Figure 1: Class diagram of the general structure of the simulator, as a message passing system.

There is no single correct rule to map the target architecture into simulator code. We have emphasized the compositionality and exchangeability of components. Thus, all high level physical entities of our design have a corresponding `Component` object in the simulator. The actions performed by components and propagation of data between components are done with the message passing facilities, the derivatives of the `Command` class. These two classes describe the static and dynamic behaviour of the target architecture.

The simulated components are free from many physical constraints, except for the functional dependencies between messages. The concept of latency associated with the execution of each command and commands' ability to synchronize the state received from their dependencies form the temporal building blocks for the semantics of the target architecture. Mapping the natural flow of control to commands can lead to unnatural designs. However, commands are powerful enough to express most (if not all) concepts on a logic circuit simulation, including varying operational frequencies. The framework also uses Scala's type checker to enforce the (type) compatibility of a message channel between a sender and a receiver, with a simple interface declaration at both ends.

Normal object oriented design methods such as composition and associations can be used to capture the modular structure of the architecture and other desired properties. Various levels of granularity can be achieved by careful choice of reusable, generic message interfaces and mixing components representing different abstraction levels.

2.2 Simulation of physical components

When implementing various physical components, the chosen abstraction level has a large impact on possible implementation issues. We explain shortly some of the implementation design choices based on our experiences.

First, the execution pipeline is rather straightforward to implement. It can be modelled as a single component, independent pipeline stages, or something between the two opposites. The pipeline design has major impact on simulation performance. A monolithic approach lacks the logical coherency between the physical and the virtual model, but can provide even a tenfold speedup [13]. We chose to use an auxiliary class hierarchy for decoded instructions. Passing a higher level construct between stages improves code reuse and aids the independent development of executable instructions.

If fetching of data from a memory has non-uniform latency, or the processing of data takes variable amount of time, the estimated latency delay may need to be updated dynamically. The same reasoning applies to e.g. routing networks. This might require some extra code in command queues since the latencies are constants by default.

Both heterogeneous and homogeneous multiprocessors can be modelled. The first uses distinct classes for processors, the latter can be simply made by constructing multiple instances of the processor class comprising all dependencies and processor internals. Additionally, some development effort can be saved by using the library components provided by the simulator's core package. Listing 1 demonstrates building a simple pass-through cache component and attaching it to a processor unit.

```
trait MemComponent {
  def loadFrom(a: Address): Returns[Data]
}
class CPU(c: Clock, val memory: MemComponent) extends Component(c)
class Cache(clock: Clock, source: MemComponent) extends
      Component(clock) with MemComponent {
    def loadFrom(a: Address) = fun { source.loadFrom(a) } init()
}
object CPU1 extends CPU(new Clock, new Cache(c, new Memory(c))
```

Listing 1: An example of adding a dummy pass-through cache for a memory.

2.3 Instruction set

What we describe here applies to the architectures, such as our moving threads based example, that use the built-in utility classes for the MIPS32 instruction set. Other implementations may define their own instruction set models since the platform does not treat instructions as special built-in constructs. By default all instructions inherit the

`Instruction` trait. The instructions inherit a more specialized class such as `TypeRCommand`, according to their type. These utility classes provide functionality for encoding and decoding the instruction, and default functionality in various pipeline stages.

```
Class Decoder {
    ...
    ADDU
}
class ADDU extends SpecialCommand((4 << 3) + 1) {
    def apply(e: ExecuteStage) = e.regs(2) = e.regs(0) + e.regs(1)
}
object ADDU extends MIPS32Instruction(() => new ADDU)
```

Listing 2: A mechanism for adding support for a new instruction called *ADDU*.

Listing 2 presents the steps required to implement a simple instruction supported by our architecture. The parameter to `SpecialCommand` defines the function field of the instruction; its opcode is deduced from the type. The apply function for `ExecuteStage` defines the actions to take when executed in the 3rd pipeline stage. Supported instructions are enumerated in a `Decoder` class. The last line is boilerplate code for factory code.

3 MOVING THREADS BASED ARCHITECTURE

Next we will describe the implementation of moving threads based target architecture. The implementation closely follows the design principles introduced in Section 2.1, i.e. the abstraction level follows the physical architecture, and there is a clear mapping between each physical component and logical `Component` object. The intra- and inter-component semantics are encoded in command objects. The following sections focus on the physical architecture, but are applicable to the simulator as well. A more detailed description of the moving threads based architecture is presented in [11].

3.1 Architectural overview

Our system consists of processor cores that are connected to each other with some sparse network [9]. In the moving threads approach, the messages between cores move a thread consisting of a program counter, an id number, and a small set of registers.

A cache-based access to the memory system is provided via each processor core, but each core sees only a unique fraction of the overall memory space, thus removing cache coherence problems. When a thread makes a reference out of the scope of its memory area, the referencing thread must be moved to the core that can access that part of the main memory. Besides the data cache, each core connects to an instruction cache.

Each of the cores contains two buffers for independent threads of execution. By taking an instruction cyclically from each thread, the core can wait for memory accesses taking a long time (and even tolerate the delays caused by moving the threads).To hide the memory and other delays, the average number of threads per core must be higher than the expected delay of executing a single instruction from any thread.

3.2 Short description of the execution pipeline

Our execution pipeline is based on well-known basic RISC architecture. The conventional RISC architecture consists of fetch, decode, execute, memory access, and write-back stages. The major differences to the basic architecture are the program counter structure and fetching of instructions from the instruction memory. In addition, the functional model of the pipeline is reorganized and extended from the original setup. Each stage is separated with extended pipeline registers. These registers contain extra fields that are re-

served to store thread's local id number and the program counter value. Both fields are carried through all pipeline stages.

Instead of one program counter the processor has a large number of parallel program counters, each dedicated with additional information fields (prefetched instruction, thread id and state fields) to a thread. In every cycle, one executable entry is selected and fetched to further processing in the processor's pipeline.

In the first phase of the pipeline the selection control finds out next instruction based on the thread id and state of these threads. When suitable entry is located, command, id and program counter value of selected thread are forwarded to the decode stage.

The decode phase operates with each instruction as the conventional pipeline does. Operation code (opcode) bits tell what kind of command is under decoding. In addition, the thread id number is used to select correct set of registers from the register file.

An arithmetic logic unit (ALU) executes demanded operations in the next pipeline phase. Terms of branch instructions are also tested. The result of ALU operations, the next program counter value, and the thread id are passed to the next pipeline registers.

A memory access stage is heavily extended from the conventional model. The data memory access operates with preloaded CAM (Contents Addressable Memory) data buffer. Parallel to data access, the instruction fetch is carried out in this step. If an instruction reference request causes an instruction cache miss, issuing thread's status is changed.

The pipeline is finalized by the writeback and predecode stage. The writeback part saves the result of arithmetic and logic operations and data memory requests. Predecoding detects, if just fetched instruction's address is inside the processor's address space. If the address belongs to address space of another processor core, a thread moving process begins. Finally the new program counter value and the instruction are stored to a table entry pointed by the thread id. The thread's status is updated depending on the type of predecoded instruction.

3.3 Preliminary performance results

Although high execution speed is not one of the primary motivators for the project, a good performance is always desirable. Especially in the multi-core context, the number of simulated components grows linearly with the number of cores. Thus, increasing the number of active processor cores will eventually decrease the potential clock frequency.

In MVTsim, the execution speed is determined by the amount of active messages passed between components during a clock cycle, independent of the size of the architecture. In the moving threads based architecture, active threads act as global event activators. When the processor executes a thread, new messages are generated throughout the execution pipeline. On a larger time frame the system can be seen stable: as new thread execution depends on earlier finished execution, the number of active messages has a local upper bound as long as the simulated program does not start new threads.

Presently, no good estimates of the number of messages per active thread execution can be given since the target architecture is in constant flux. Moreover, several parts of the simulator for moving thread based architecture are still unimplemented, and finally, the number of messages will also depend on the instructions executed by the simulator.

However, since the simulation framework is already in usable state, we made some preliminary test runs with it. On a 2.66 GHz Core 2 Duo, running Linux 2.6.29.2, Sun JVM 6u14, and Scala 2.7.5, the framework ran on average a modest result of 600.000 virtual cycles per second, with one component and one message per cycle, once the JIT compilation had stabilized. This is about 4 times slower than the SESC simulator [12], running fully implemented R10K virtual CPU on a Pentium III. Further optimizations, e.g. utilizing several processor cores on supporting host systems, are yet to be considered.

4 CONCLUSIONS

MVTsim can be used to implement the proposed RISC-based architecture for a processor supporting the moving threads approach. The main advantages of MVTsim are its general purposeness, extensibility, modularity and support for variable granularity levels, provided by the component based message passing framework. In addition, the simulation is cycle-accurate, which allows analysing the execution of the virtual instructions precisely.

So far the flexible design has supported the iterative development of the target architecture. Parts of the proposed features, e.g. those enabling the use of multiple cores, have not yet been fully implemented, and thus a thorough analysis of the simulator is not yet possible. Additionally, the current architecture performs somewhat slower than existing simulators considered in this paper, but our focus will be on the performance issues once the general design of the underlying architecture has stabilized.

REFERENCES

[1] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau, Building the 4 processor SB-PRAM prototype, In Proceedings of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Vol. 5, 1997.

[2] F. Bellard, QEMU, a fast and portable dynamic translator Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41-46, 2005.

[3] M. Forsell, A Scalable High-Performance Computing Solution for Network on Chips. IEEE Micro22(5) (September-October 2002), pp. 46-55.

[4] M. Forsell and V. Leppänen, High-Bandwidth On-Chip Communication Architecture for General Purpose Computing, Proceedings of the 9th World Multi-Conference on Systems, Cybernetics and Informatics, WMSCI'2005, pp. 1–6, 2005.

[5] M. Forsell and V. Leppänen, Moving Threads: A Non-Conventional Approach for Mapping Computation to MP-SOC, In Proc. 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'07), pp. 232-238, 2007.

[6] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, In Proceedings of the 10th ACM Symposium on Theory of Computing, pp. 114-118, 1978.

[7] J. Jájá, An Introduction to Parallel Algorithms, Addison Wesley, 1992.

[8] J. Keller, C. Kessler, and J. Träff, Practical PRAM Programming, Wiley, 2001.

[9] V. Leppänen, Studies on the Realization of PRAM, PhD thesis, University of Turku, Department of Computer Science, TUCS Dissertation 3, November, 1996.

[10] V. Leppänen, Balanced PRAM Simulations via Moving Threads and Hashing, Journal of Universal Computer Science, 4:8, pp. 675–689, 1998.

[11] J. Paakkulainen et al., Outline of RISC-based Core for Multiprocessor on Chip Architecture Supporting Moving Threads, Proceedings of the conference on Computer Systems and Technologies, CompSysTech' 09, to be published.

[12] J. Renau et al., SESC simulator, http://sesc.sourceforge.net, January, 2005.

[13] P. Strazdins, CycleCounter: an Efficient and Accurate UltraSPARC III CPU Simulation Module, Technical reports, The Australian National University, Department of Computer Science, 2005.

[14] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-On-Chip processor, Computer Frontiers 2008, May 5-7, 2008.

ABOUT THE AUTHORS

Researcher Jani Paakkulainen, MSc, Department of Information Technology, University of Turku, Finland, E-mail: jani.paakkulainen@utu.fi

Researcher Jari-Matti Mäkelä, BSc, Department of Information Technology, University of Turku, Finland, E-mail: jmjmak@utu.fi

Assoc. Prof. Ville Leppänen, PhD, Department of Information Technology, University of Turku, Finland, E-mail: ville.leppanen@it.utu.fi