Building an SMT Application Simulator

Richard Cox
rick@cs

Andrew Schwerin
schwerin@cs

March 15, 2002

1 Introduction

Simultaneous multithreading (SMT) is an active area of research into an alternative architecture for exploiting parallelism. It addresses the question of how to use the large number of transistors available on modern chips given the low IPC available from a single thread.

It also requires examination of various processor and architecture design decisions. SMT processors may exhibit different cache, branch-prediction, and utilization patterns than conventional processors [10, 9]. While studies of several of these factors have been undertaken, there are many more variables to be examined; each component found on a conventional chip may behave differently when several threads are competing for its resources. All past studies have been undertaken using simulations of SMT processors, however, the users of the existing simulators are frustrated by slow execution speeds and difficulty of modifications.

By simulating the application-level only, we aim to increase speed. By designing for extensibility, we hope to make modifications easier. And finally, by basing our simulator on a popular, verified, and well-understood Alpha simulator, we can provide more confidence in its accuracy and share code and documentation with other efforts.

2 Related Work

2.1 Application-Level Simulators

Our work is a modification to the SimpleScalar [1] Alpha simulator (sim-alpha). Other groups have also modified SimpleScalar to support various threading architectures, including Superthreading and multi-processors. They do not, however, support simultaneous multi-threading. We also have the advantage of starting our work with a validated simulator [3], lending some credibility to our results.

Previous SMT researchers have used both application level simulation [10, 9, 5] and full machine simulation [8]. While the later category of simulators are important for

studying certain workloads, the former can provide higher performance, allowing longer traces to be studied, and provide reasonable accuracy for many workloads [8].

2.2 SMT

It is our intent to support future research in simultaneous multi-threading by providing a flexible simulator. Though we cannot predict all possible experiments or architectural modifications, we have used the past research to help anticipate and design for changes.

A simulator-driven analysis of simultaneous multi-threading is presented in [10]; that work demonstrates the need for flexibility in the simulator. They present experiments examining the IPC available when several different restrictions on the issue stage are in place. They examine the effect of shared caches by modeling several different modes for the L1 cache. Finally, they compare SMT with chip multi-processors, varying issue bandwidth, register sets, and functional units. We expect that future simulations will also need to adjust these parameters and algorithms.

Our base-line simulator is intended to closely mimic the architecture presented in [9], which is very close to the Alpha 21164 with an added cycle of register latency. We allow the user to specify this latency. Also like their simulator, we provide detailed simulation of branch misprediction, including per-thread pipeline flush. Noting that fetch algorithms are a common experimental parameter, we have made the fetch stage especially easy to change.

Later work has looked at parallel processes and the thread scheduling and sharing effects on SMT processors [5]. To support this workload, we use a general model in which several different multi-threaded processes may be running concurrently (we do not currently implement the syscalls needed for spawning new threads, however). We do not support any novel synchronization primitives [11]. As described in [7], simulations with more active threads than hardware contexts reveal effects not seen when every thread has a hardware context. While our simulation does not currently support this configuration (we do not support swapping-out hardware contexts to memory) we have left that possibility open for future work.

Some research has been done into register deallocation schemes made possible or useful by SMT processors [6]. While we do not support any such extensions, we do not require all of a thread's architectural registers be mapped at any time, enabling experimentation.

Simultaneous multi-threading has also been examined for non-traditional uses, such as network processors [2]. Our work would be difficult to extend to this area because we simulate an application and do not model IO, which is performed via fixed-latency syscalls.

3 Design

Designing a computer hardware simulator presents interesting challenges from the standpoint of software engineering. Because it is a research tool, users are bound to need to customize it. Because of the detail of data that must be collected, it is liable to be slow. The simulator designer must choose throughout the design between flexibility of change and speed of execution.

The authors of sim-alpha had the goal of accurately simulating the application-level execution of an existing processor, and validating the simulator's results against the real thing [3]. Another goal of the authors was to provide a baseline simulator that could be extended by researchers in order to experiment with new microarchitectural ideas. Unfortunately, little documentation is currently available about the inner workings of the simulator, and the design is challengingly complicated.

Our goal has been to extend sim-alpha to simulate SMT architectures in a manner such that other researchers may easily implement modifications. Our first step in this direction has been to sharpen the rather blurry line between various modules within the simulator. By restricting the interface between modules, we hope to permit experimenters to make local changes without need for full global understanding of the system. Our second step has been to vigorously apply procedural abstraction in order to reduce the amount of code that must be changed during modifications. Our final, and probably most important step has been to improve the state of the documentation available. We have attempted to document how various global variables form part of the interfaces between different modules, to document the proper use of functions and, when necessary, to document how the code actually carries out those functions. The work is incomplete, but we believe it is still useful.

3.1 Pipeline and Blackboard

The smtsim simulator employs a hybrid pipeline-blackboard software architecture to model execution characteristics of a single hypothetical SMT processor. The blackboard (Figure 2) is the conceptual storage space for all threads and processes, for the physical register file, and for instructions. Threads store information about individual threads of execution, such as the current program counter for the thread. Processes store information about address spaces, and every thread belongs to exactly one process. Instructions store all of the information used by the various pipeline stages to simulate execution, such as op codes, dependent instruction queues, and architectural and physical register assignments. Every instruction belongs to one thread.

As shown in Figure 1, the pipeline consists of six execution modules and eight explicit queues or buffers. The fetch execution module constructs new instructions on the blackboard using data from the simulated memory. All of the other modules transform those instructions, removing them if they are no longer valid. All modules except

fetch receive references to instructions on the blackboard only via their input queues and the dependency queues of other instructions. All of these queues and buffers are accessed via a unified interface, called the SMTQ interface.

3.1.1 Map

After the fetch execution module adds instructions to the blackboard, it passes them on to the slot module via the FIFO Fetch Queue. Once the slot module has added its Alpha 21164-style slotting information to the instructions, it passes them on to the map phase via the FIFO Slot Latch. The map phase adds instructions to the Reorder Buffer, which is modeled as a queue with unusual dequeuing semantics. Specifically, the head of the reorder buffer is the oldest instruction in the buffer that has completed and has no older uncompleted instructions in its thread. Note that at times, there is no head of the buffer because no instructions in it meet these criteria. In the case where there is only one thread, the reorder buffer is a FIFO queue that has a head only when the oldest instruction in the queue is completed. Since the map module adds instructions to the reorder buffer in program order, the oldest instruction in a thread is always the one nearest the front of the queue. It is the special semantics of the reorder buffer that guarantee that instructions commit in order on a per-thread basis, and that any thread ready to commit instructions is allowed to make progress. Interestingly, these semantics also mean that the reorder buffer on a SMT processor must probably be searched frequently, making it challenging to keep off of the critical path.

Once the map phase has placed an instruction into the reorder buffer, it takes action based on the type of the instruction. If the instruction is a load or a store, it is added to the load or store queues, respectively. Notably, the load and store queues in the simulator's implementation provide no functionality not available from the reorder buffer, and could probably stand to be removed. If the instruction's operands are ready, map places the it into the appropriate issue queue.

3.1.2 Issue, Exec/Writeback

The issue and exec/writeback stages, and their associated queues, form the out-of-order core of the processor. While all other stages and queues process (or store) instructions in program order, these stages process as resources become available (resources may be functional units or the results of previous instructions).

The issue queues provide instructions to the issue execution module, which removes instructions that can be scheduled onto available functional units. Such instructions are placed onto the writeback input queue, prioritized by the time at which the functional unit will finish executing. The writeback execution module (Exec/Writeback in diagram) removes instructions from its input queue in priority order. For each instruction i that it removes, it marks as ready the appropriate operands of all instructions

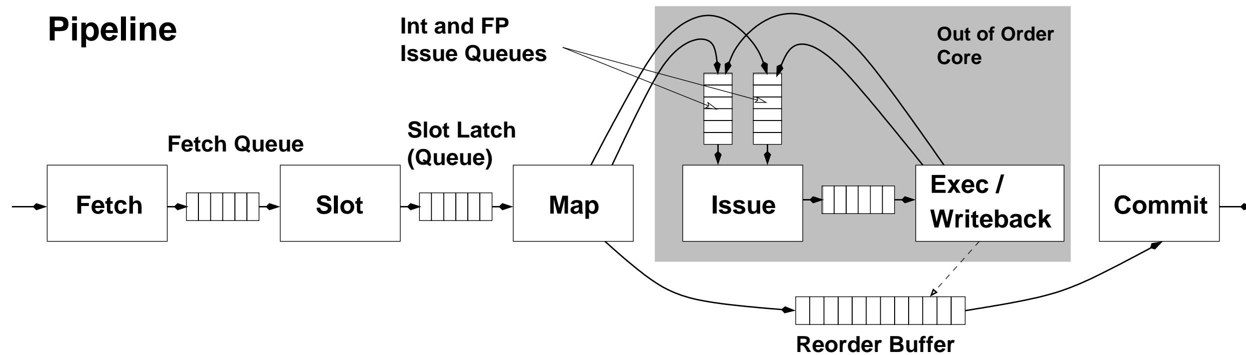

Figure 1: The Pipeline

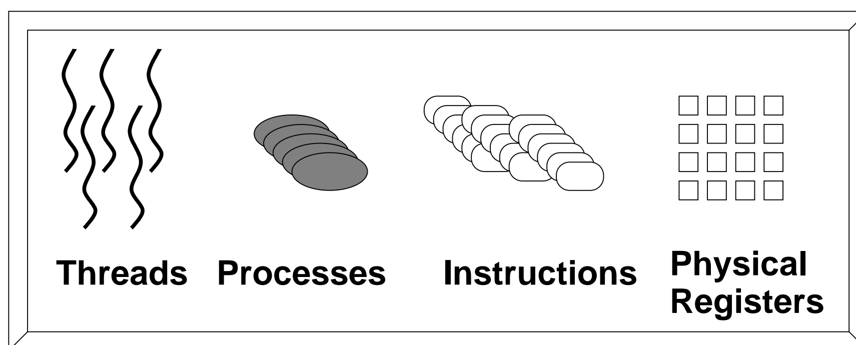

Figure 2: The Blackboard

dependent on the result of i. If those dependent instructions now have all of their operands ready, they are placed into the appropriate issue queue. Writeback also marks i as completed, possibly affecting the head of the reorder buffer.

3.1.3 Commit

Every cycle, the commit queue removes up to commit_width completed instructions from the head of the reorder buffer. For each completed instruction i that it finds, it runs the following procedure:

1. It checks to see if i is a trap, flushing the pipeline of i's thread's instructions if it is;

2. It retires any output registers by setting the committed register table entry for the architectural output register to the instruction's allocated physical register;

3. It checks to see if i is a mis-predicted branch, changing i's thread's program counter to the correct target and flushing the pipeline of i's thread's instructions if it is;

4. If the instruction is OK to retire, and it is a store, commit allows the store to go to memory;

5. If the instruction is OK to retire, commit removes it from the blackboard, effectively retiring it.

3.2 Instructions

In the current implementation, instructions are heavy-weight objects. Each instruction has three queues for holding references to other instructions that depend upon it. Two of those queues are for the standard operands. The third is for the destination of Alpha CMOV operations, since CMOV depends upon the producer of the old value of its destination register. Instructions also store their PC value, information used for the branch predictor, the physical and architectural registers that are their targets and sources, and various information that determines when they are ready to execute. It is important to realize that this information is combined in order to simplify the implementation of the simulator, not because all of this information is stored together (or even stored at all) in hardware implementations.

3.3 Multiple Thread Contexts

The biggest difference between a SMT and a traditional super-scalar is the presence of multiple thread contexts on the processor.

In our simulator, information about a thread that is used by many modules is handled by a special thread module, while information about a thread that pertains only to a single module is handled by that module. For example, the architectural register file is a property of a thread handled by the thread module, while the fetch status of each thread is managed entirely by the fetch module. In general, the thread module

handles only those aspects of threads that are shared between modules and necessarily thread- or process-specific. An example of something meeting the former criterion but not the latter would be caches, which may be globally shared or thread- private.

Within the thread module, data is either thread-specific or process-specific. Thread specific data are the data that pertain exclusively to a specific thread context. Examples of this are the architectural register file and the mapping from architectural registers to physical registers. Process specific data are primarily data relating to the virtual memory address space in which the thread is operating.

To enhance software clarity and readability, access to thread context information is performed through accessor and mutator functions. These functions are simple enough to be in-lined by the compiler or replaced with simple macros, if they prove to be a serious bottleneck. We believe, however, that other computations within the simulator will dominate its runtime.

3.4 Mediator and Eventq

Two types of indirect invocation are used to drive the simulator: implicit but immediate invocation, and delayed invocation. The former is provided by the mediator module, and the latter by the eventq module.

The mediator [4] is used in order to reduce the coupling between execution modules. It accomplishes this by allowing modules to post important events, so that interested modules may learn about them. All functions registered to listen for some kind of mediator event are executed after the event is posted and before the posting function resumes, though the order in which they are executed is not specified. For example, the mediator allows the commit stage to flush the pipeline without explicitly flushing every inter-phase queue. The reduction in inter-module coupling gained by this is substantial, and it seems likely that more extensive use of the mediator might simplify modules in the simulator as well.

The eventq module is used in order to invoke an event at a specified, future cycle in the simulation. All modules that perform multi-cycle events, such as loads, stores or executions must use this module in order to arrange for the appropriate timing behavior. The memory system uses eventq to schedule the termination of operations on different levels of the memory hierarchy, and the issue execution module uses it to perform proper accounting relating to the execution of instructions.

3.5 Flexible Fetch

An interesting point of divergence from the design of sim-alpha is in the instruction fetching module. Because instruction fetching on a SMT system is conceptually more complicated than on a traditional superscalar, and because instruction fetch policies are an area of experimental interest, it has been a major focus of our design changes.

The smtsim fetch module allows multiple fetching policies to be linked in at compile time, leaving selection of a specific policy

until runtime. This is achieved through a single level of pointer indirection, and is invisible to the rest of the simulator. When a researcher wishes to test a new instruction fetching policy, she may implement it on top of a set of shared auxiliary methods without removing or destroying the existing fetching policies. The auxiliary methods calculate cache latencies, interact with the branch predictors, and interface to the memory module; the writer of a new fetch policy needn't have a global understanding of the entire simulator.

Conceptually, a fetch policy is an object that handles five messages:

- fstage_init,
- fstage_exec,
- fstage_instr_retire,
- fstage_instr_flush, and
- fstage_thread_flush.

The fstage_init message instructs a fetch policy to initialize itself. It takes no parameters, and is expected to use data that it received from the factory method that created it in order to perform its initialization. The fstage_exec message is invoked once every cycle, and is the entirety of the fetch stage. The fetch policy chooses one or more threads from which to fetch in a given cycle, and invokes smt_fetch_from_thread in order to fetch instructions from each of them. The remaining three messages are used to notify the fetch policy when instructions leave the pipeline, either because the pipeline was thread-flushed, because an instruction was retired, or because a single instruction was flushed from the pipeline.

Currently, the fetch policy object has two extensibility flaws. First, it does not allow for the storage of extra data in the fetch policy object. Second, due to an oversight, the policy object itself is not made available to the methods implementing the fetch policy messages. This could easily be changed by making the first parameter of every policy method be a pointer to the policy object, and since the fetch policy object is completely isolated within the fetch module, only minimal changes would be necessary to the source code. The only reason that these changes would be valuable would be if one were to build a CMP simulator as an extension to smtsim. For a single-core simulation, there is only one fetch policy object at any given time, so static data suffices.

3.6 Memory Architecture

The memory architecture in smtsim consists of simple modifications to the memory architecture used in simalpha. Every thread has a reference to the process within which it operates. Every process has its own virtual address space. There is a single, shared physical address space, and the MMU and TLB module maintain a mapping between (process, virtual-address)-pairs and physical addresses. Memory operations, such as memory and cache accesses, use a thread argument in order to determine which process space to use.

The current implementation of the MMU

and TLB module provides every process with its own complete MMU. This is unrealistic, and affects simulation results by reducing the latency of the MMU in cases where several threads are simultaneously attempting to access it. The modifications necessary to make a more realistic MMU should not be extensive, nor should they not require global knowledge of the system; they were omitted primarily so that we could focus on other matters.

It could be quite illuminating to study how shared memory IPC on a SMT compares with that of a traditional multiprocessor. We have therefore attempted to enable such an extension by eliminating certain simplifying assumptions used in sim-alpha. Namely, sim-alpha did not store correct values in its physical page table, because such values are not necessary in a simulator without shared memory.

At present, the only way that threads may share memory is if they operate within the same process space. Inter-process communication via shared memory could be provided by mapping virtual pages from different process spaces to the same physical page. Indeed, only the virtual-to-physical page mapping is necessary for an accurate calculation of memory access latencies. The most practical approach for adding this functionality would be to define new memory management system calls. This would require modifying existing programs, however, the alternative — supporting POSIX or SysV shared memory — would require the simulator to support many operating system-like features.

3.6.1 Challenges

The memory system, though updated aggressively from sim-alpha, is cumbersome. It contains some very long methods with highly complicated logic, much of which is implemented via side-effecting macros. It has significant amounts of dead code, and though the goto statements have been replaced, the principle functions (cache_timing_access and cache_translate_address) could stand to be re-written from scratch.

3.7 Branch Prediction

The branch predictor was left much as it was in sim-alpha, though the line predictor was removed. All attempts were made to preserve the line predictor code, but it was scattered so liberally throughout the execution models that it was too challenging to cope with while make other changes to the simulator. The result of this seems to be a drop in branch predictor accuracy to less than 75%, though the line prediction code muddled the statistics on this slightly. The traditional always-not-taken predictor should continue to operate (if it did in sim-alpha), as should the always-taken predictor, though the latter might be damaged due to integration of target address prediction with the line predictor.

4 Results

Because our primary goal was the construction of a simulator, we have focused on an-

alyzing it (see Appendix A for information on using the simulator).

Having added 3,900 lines of code in 28 new source files, removed 19 files containing 9,900 lines, and generated 12,800 lines of differences in pre-existing files, we were curious to know how we had affected the efficiency of the simulator[1]. Thus, we ran both sim-alpha and sim-smt on the same hardware (a 2GHz P4). We used the first 1 billion instructions in an execution of gzip to calculate the simulation rate. Sim-alpha processed 279 thousand instructions per second, while sim-smt clocked in at 154 thousand instructions per second.

The speed difference may be due to a problem with excess pipeline flushes in sim-smt; our simulator caused 3.2 times as many flushes as sim-alpha. Not only do flushes require re-processing several instructions, but data must be moved to and from the L1 cache again. While sim-alpha generated about 1GB of L1 data cache traffic, sim-smt generated over 2GB.

One aspect we do not believe significantly influences our performance is the introduction of abstractions and data hiding.

5 Conclusions

We were quite surprised at the extent of modifications required to create an SMT simulation. We were even more surprised, though, to find that most of the work was in implementing the per-thread pipeline flush logic; it was the only SMT feature that required global coordination between the phases. It was also this feature that prompted us to re-factor sim-alpha because the number of data structures — instruction queues, load/store queues, the eventq, cache accesses, TLB entries, etc. — that required modification was inordinate with its conceptual complexity. We wonder whether pipeline flush complexity increases similarly in transitioning a real super-scalar processor to an SMT.

5.1 Directions

An interesting question for hardware simulation is how SMT and CMP architectures compare under various circumstances. The smtsim simulator could be extended to do CMP application simulation, and even multi-core-multi-threaded simulation with a reasonable amount of effort.

Another very useful extension to this simulator would be to improve the statistics gathering process. Although the system is quite useful as is, it is used in a manner that is heavily reliant on global data that is often modified in subtle ways. This makes it difficult to add new statistics and thread-specific statistics.

Acknowledgments

We would like to thank Mark Oskin, Tim James, Susan Eggers, Luke McDowell, Dimitriy Portnov, and Steve Swanson for their help understanding the details of simultane-

[1]For reference, the sim-smt totals 37,900 lines of C code

ous multithreading. We would like to thank Steve Swanson for the powerful host machine.

References

[1] Douglas C. Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[2] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[3] Rajagopalan Desikan, Doug Burger, Stepen W. Keckler, and Todd Austin. Sim-alpha: A validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, University of Texas at Austin Department of Computer Sciences, 2001.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[5] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, August 1997.

[6] Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Henry M. Levy, and Dean M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):922–933, September 1999.

[7] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, 2000.

[8] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[9] Dean M. Tullsen, Susan Eggers, Joel Emer, Henry M. Levy, and Jack Lo. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 192–202, May 1996.

[10] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th*

Annual International Symposium on Computer Architecture, pages 392–403, June 1995.

[11] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous mulitthreading processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 54–58, January 1999.

A sim-smt

The source for sim-smt is available from the authors by request. (It is covered by the SimpleScalar license, which prohibits commercial use of the software.) Only Linux/x86 hosts and OSF/Alpha binaries in ECOFF format are currently supported. We do not currently support fast-forwarding of simulated program initialization.

A.1 Usage

Sim-alpha users should find the interface of sim-smt very familiar. The build process is identical. The invocation is the same save for the simulated program specification; all sim-alpha configuration options are supported by sim-smt. Where sim-alpha treated all arguments following the options as a program name and simulated arguments, we allow several program name/argument sets to be specified. This means the user must escape the spaces separating each program and its arguments. As an example, the following would run two threads, one with gzip and one with bzip:

sim-smt -config mem.cfg ./gzip\ input.gz ./bzip\ input.bz2

By default, sim-smt is compiled with support for up to 8 thread contexts. This can be changed by editing the THREAD_MAX_CONTEXTS constant in thread_types.h.

A.2 Bugs

sim-smt currently has a unlocated bug involving some form of simulated memory or register corruption for certain multi-process workloads (the exact workloads vary from host to host). The authors believe the issue to be a memory corruption problem, but neither tools nor scrutiny have revealed the offending code.

The cache simulation never generates a "fast-hit," a lookup found in a hash-table instead of via linear search, and returned slightly faster than a regular hit; this should be easy to fix.