Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA

Nirav Dave*, Michael Pellauer*, Arvind*
Computer Science and Artificial Intelligence Laboratory*
Massachusetts Institute of Technology
Cambridge, MA 02139
Emails: {ndave,pellauer,arvind}@csail.mit.edu

Joel Emer*†
VSSAD†
Intel Corporation
Hudson, MA 01749
Email: joel.emer@intel.com

Introduction

When creating a microarchitectural simulator, one desires three things: confidence in correctness, speed of design, and speed of simulation. The first requirement is necessary for accurate experimentation. The second impacts the architect's ability to perform microarchitectural exploration by rapidly describing a range of systems. The third affects the number of simulations that can be profitably run for a particular systems, and thus the statistical confidence in our evaluation results.

One approach is to divide the simulator into two partitions: a functional partition and a timing partition. The functional partition is essentially a pipelined abstract execution engine, which allows dataflow-valid executions through an idealized pipeline. The timing partition, on the other hand, is responsible for modeling microarchitectural details such as operation timing and branch predictor accuracy. The timing partition is responsible for directing *when* instructions should be executed, but does not need to concern itself *how*.

ASim is a software simulator which makes use of this partitioned organization [6]. ASim's functional partition is an infinitely-renamed, infinitely-buffered pipeline. The functional partition does not perform instructions immediately upon fetch (the "execute-on-fetch" simulator model) but progresses the instruction through various pipeline stages (Decode, Execute, Commit, etc) upon direction from the timing partition. This multi-staged approach allows the simulator to model processors which are superscalar, speculative, or out-of-order, solely by altering the timing partition.

This split organization increases confidence in simulator correctness, as verification effort is focused on the functional partition. Similarly it reduces implementation effort, as the functional partition can be reused across multiple timing-partition implementations. However this approach incurs a communication overhead which can degrade simulation performance. Since the timing model and the functional model move in lockstep, opportunities for parallelism in a software simulator are limited [2]. One approach as in the FAST simulator [4] is to decouple the partitions as much as possible, thus minimizing communication overhead and increasing opportunities for parallelisation.

In this paper we present HASim (pronounced HAY-sim), an alternative approach to this problem. In HASim the timing and functional partitions are tightly coupled, and are jointly implemented in hardware, rather than software. The intent is to place the resulting design onto an FPGA, thus allowing us to improve overall simulator performance. We will use this project to explore optimizing the sequential lock-step movement of a partitioned simulator into an optimized parallel hardware structure.

Currently HASim implements a toy ISA as a proof-of-concept. In the future we hope to expand the framework to realistic ISAs and architectures.

Methodology

HASim is implemented in Bluespec SystemVerilog [3] and follows the partitioned architecture outlined above. The major difference is that the functional partition and the timing partition are organized as hardware pipelines (as shown in Figure 1) rather than software modules. This allows them to execute in parallel and should allow it to achieve throughput comparable to that of an actual pipelined processor.

The Timing Partition

The timing partition contains microarchitectural-specific information, such as the resource limitations of the target design and the timings of the various operations. Thus we expect it will be reimplemented many times across different experiments and models. Note that one physical FPGA clock cycle does not need to correspond to a clock cycle in architecture being modeled (a *target* clock cycle). As the timing partition will likely be pipelined, we expect that different pipeline stages will contain instructions on a range of target clock cycles. We will investigate adding safeguards to ensure that causality is maintained and that information from future target cycles cannot affect instructions in the "past."

When the timing partition encounters a new instruction, it provides a unique token to the functional model to refer to that instruction. Any time the timing model wishes to execute a particular stage of an instruction it must provide the associated token. Later it will get an acknowledgment that the action occurred along with any associated pertinent information for timing.

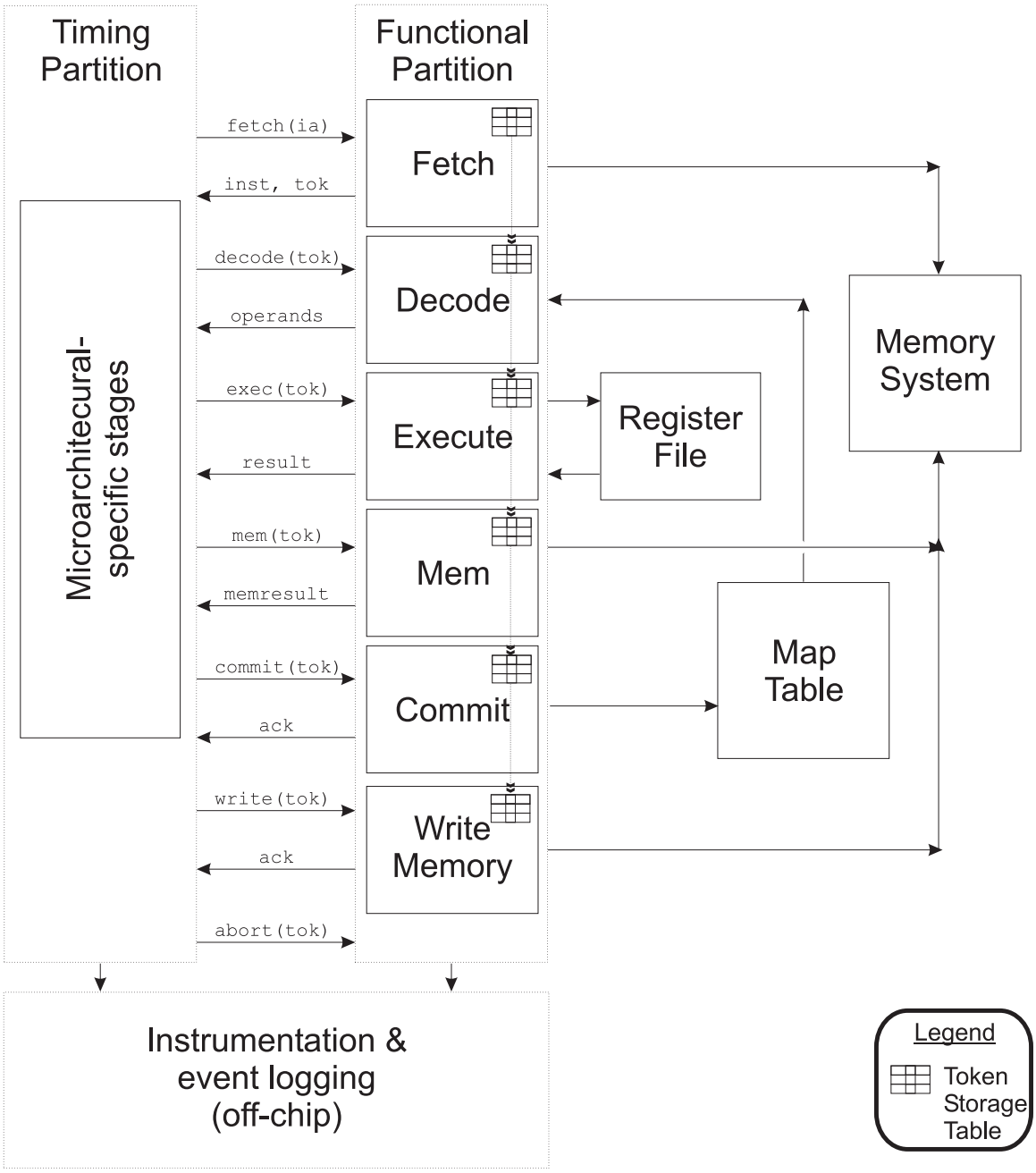

Fig. 1. HASim High-level Organization

The Functional Partition

The functional partition represents the microarchitecture-independent portion of the simulator. Thus we expect that it will be implemented once, optimized and verified, and reused many times with many different timing partitions. Currently, it is implemented as a 6-stage pipeline: Fetch, Decode, Execute, Mem, Commit, and WriteMemory. (This organization is for simplicity — a more complex organization may ultimately prove to be more efficient.) Each stage in the pipeline contains a table of instructions currently waiting at that stage. The functional partition contains no logic for issuing or stalling - instructions simply do not pass to the next functional pipeline stage until the timing partition directs it. However, within each stage instructions are free to execute as far possible without actually committing their result. This helps the simulator to hide the latency of expensive operations such as multiplication or division.

To deal with misspeculation, each functional partition unit has an interface which allows the timing partition to invalidate wrongpath instructions. This method simply invalidates all associated entries in the token tables for each unit.

A possible optimization opportunity is to allow allow differ-

ent units to slip in regard to target cycles with each other. This would allow speculative execution of the functional partition. To do this we would have to tagged each interaction between the timing and functional partitions with the current target clock cycle [1]. This increased asynchrony allows us to hide latencies of individual different stages of the functional units. For instance, one could imagine the functional partition's Fetch stage running ahead of the rest of the design to minimize the physical load latencies (as opposed to the simulated latency of the target design).

Whereas a software functional partition typically has unbounded buffer sizes and register renaming, in hardware we have no such luxury. Currently, the functional model is parametrized along various buffer sizes. We leave it to the designer to ensure that all sizes are sufficiently large. If during run-time a buffer is found to be too small, preventing forward progress, an error is recorded.

Instrumentation

Instrumentation is paramount in using any emulation platform like HASim. Additionally, it is desirable to reuse instrumentation and tracing facilities across multiple implementations. Our approach is to record information in the functional partition, since this partition must already keep track a great deal of per-instruction information. This gives us a unified place for all off-FPGA communication and ensures that such code does not need to change across microarchitectural implementations.

Because communication off-FPGA communication is so expensive, we provide a mechanism to prevent the latching of new trace values. When trace values are needed, values are stored in a buffer and serially read off the chip. Because this process will likely be much slower than execution of a cycle, reducing the rate of target cycle ticks will be necessary. However, if we carefully pick out when to instrument, very little slow down will need to happen.

Future Work

In the future we hope to be able to extend our functional partition model whole systems. To do this we will exploit possible asynchronies to improve performance. We will also explore how to best implement timing partitions for superscalar and speculative architectures. Support for multiprocessor systems could be achieved via duplicating functional partitions, or perhaps by sharing them. We also hope to use large FPGAs platforms to run models previously which are difficult to simulate by current software techniques, such as multi-megabyte caches.

Additionally, we hope to extend our functional model to incorporate realistic ISAs such as x86. One possibility is to combine this project with UNUM [5]. Alternatively, because the functional model implementation is sufficiently detached from the timing model, we may choose to incorporate a number of different ISAs into a single reusable partition via some universal micro-operation ISA.

References

[1] Kenneth C. Barr, Ramon Matas-Navarro, Christopher Weaver, Toni Juan, and Joel Emer. Proceedings of 3rd annual Boston Area Architecture Workshop. Providence, RI, 2005.

[2] Kenneth C. Barr, Heidi Pan, Michael Zhang, and Krste Asanovic. Accelerating a Multiprocessor Simulation with a Memory Timestamp Record. In *International Symposium on Performance Analysis of Systems and Software*, Austin, TX, 2005.

[3] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[4] Derek Chiou, Huzefa Sunjeliwala, Dam Sunwoo, John Xu, and Nikhil Patil. FPGA-based Fast, Cycle-Accurate, Full-System Simulators. Number UTFAST-2006-01, Austin, TX, 2006.

[5] Nirav Dave and Michael Pellauer. UNUM: A General Microprocessor Framework Using Guarded Atomic Actions. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms held at HPCA-11*, San Francisco, CA, February 2005.

[6] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.