Lessons from Implementing a FAST Prototype

Derek Chiou, Dam Sunwoo, Nikhil Patil, Joonsoo Kim, Bill Reinhart, Hari Angepat, D. Eric Johnson
Electrical and Computer Engineering, UT Austin
{derek,sunwoo,npatil,turo,wreinhar,angepat,dejohnso}@ece.utexas.edu

***Abstract*— Over the past three and a half years, we have been working on defining and developing FPGA-Accelerated Simulation Technologies (FAST)[2], [3], [4], [5], a novel methodology that uses FPGAs to build very fast simulators of complex computer systems. We have built a prototype that simulates the x86 ISA, boots Windows XP and Linux and runs unmodified applications at an average of 1.2MIPS, with the expectation of being able to run significantly faster.**

This talk describes some of the issues that we ran into while building our prototype and how we addressed them. It also describes areas where things worked out right, either due to foresight or blind luck.

I. Introduction to FAST

Like many other simulators[7], [6], [1], FAST is partitioned into a functional model and a timing model. The functional model simulates the computer at the functional level, including the instruction set architecture (ISA) and peripherals, and executes application, operating system and BIOS code. To first order, the timing model simulates only the micro-architectural structures that affect the metrics that are being modeled by the simulator. For example, to predict performance, we need to model such structures as pipeline registers, arbiters and associativity. Functionality is generally not modeled by the timing model.

The functional model pushes an instruction stream to the timing model that then uses that instruction stream as a first cut as to what instructions will be fetched and executed. Of course, most modern microprocessors use branch predictors that are, unfortunately, not always accurate. Because the functional model does not model timing, most functional model implementations will, at some point, produce an instruction stream that is inconsistent with what the target micro-architecture[1]. At that point, the timing model signals back to the functional model what instructions it expects to see, in the order it expects to see them and the functional model redelivers the instruction stream as requested. Such corrections occur, for example, during branch misspeculation and resolution.

[1]We use the term *target* to mean the machine being simulated and the term *host* to mean the machine (*e.g.*, computer/FPGA) that the simulation runs on.

FAST is similar to a pure software simulator, FastSim[7], whose functional model executes instructions and passes the trace to the timing model that can trigger a roll-back in the functional model to model target misspeculation recovery. However, *FastSim's functional model itself does speculatively execute.* The functional model uses the timing model branch predictor to determine if a branch will be misspeculated, thus eliminating the possibility of functional model misspeculation due to the functional model not misspeculating a branch that the target system would. However, such tight coupling between the FastSim functional model and timing model make it difficult to parallelize.

Unlike any other simulators that operate in such a fashion, FAST is parallelized at two levels: between the functional model and timing model and within the timing model by implementing it in one or more FPGAs. Our current timing model is implemented using Bluespec SystemVerilog and runs on a DRC Computer Prototyping System that contains a Xilinx Virtex-4 LX200 FPGA that sits in a HyperTransport socket on a dual-processor motherboard with an Opteron processor in the other socket. Our current functional model is a heavily modified version of QEMU that supports both instruction trace as well as checkpoint/rollback to support timing model corrections. We support not only the x86 instruction set, but also the PowerPC instruction set. More details can be found in our previously published work[2], [3], [4], [5].

II. Lessons

We learned several lessons during our development of our FAST prototype, some of which we list here.

A. Simulator performance tracks target performance

When we first started on our timing model, we operated under the assumption that we had plenty of host cycles on the FPGA. Though that assumption is fundamentally

true, since 10 host cycles at 100MHz is still 10MHz of simulator performance, the number of host cycles used can very quickly get out of hand. Our initial version of the Fetch stage that included a branch predictor and an L1 cache that could execute in a single host cycle consumed an average of approximately 30 host cycles and sometimes consumed up to 50 host cycles. These large numbers of cycles were due to doing too many things sequentially in a single target cycle. If our target was designed in the same way, it would have long cycle times and correspondingly poor performance. Our next generation Fetch module is partitioned into three distinct stages — branch prediction, instruction TLB and instruction cache — essentially modeling a more aggressive target. We expect significantly faster simulation from that timing model.

B. Multiple Target Cycles

Our initial prototype was limited in target L2 cache size because we could not meet timing beyond a certain size even though the FPGA was far from full. The timing problem was caused by the way the cache module was written. Though it would use multiple host cycles to model arbitrary associativity, it was also written to complete its search in a single target cycle which, for our implementation, implied a single set of n comparators, where n is equal to the number of comparisons that could occur in a single host cycle. The result of those comparators were ORed together to a single point on the FPGA. The place-and-route tool faced a dilemma; put the comparators close to the block RAMs and thus further from the OR point or closer to the OR point and thus further from the block RAMs (Figure 1.) Thus, as the L2 cache became larger, the lines grew in length.

One way to address this problem while potentially improving simulator performance leverages the observation that a large L2 cache will generally take multiple target cycles to access. Our next generation cache is implemented to be able to use multiple target cycles to perform a single search, pipelined, of course, so that a request can also be launched each target cycle. The implementation will permit the ORed signal to be done in multiple target cycles as well, resulting in scalable caches that meet timing.

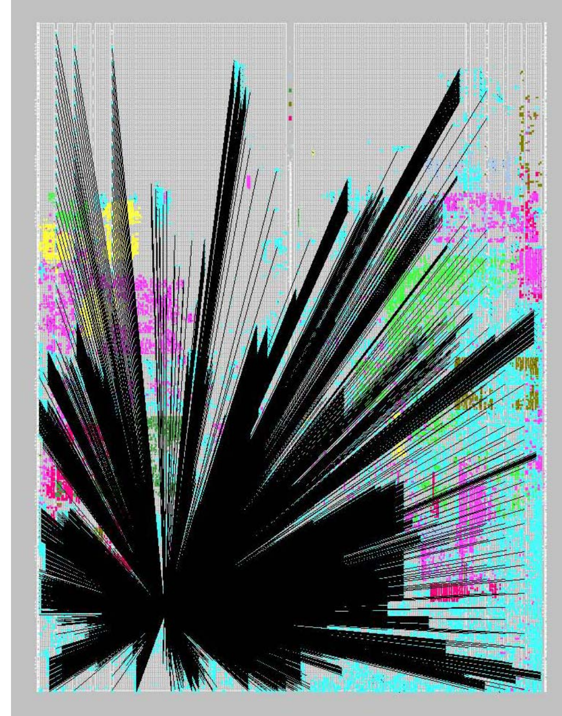

Fig. 1. Rationale for Multiple Target Cycle Caches

C. x86 is a LOT of Work

The FAST partitioning enables us to use software-based functional models. Because of the dominance of the x86 ISA and the fact that many software x86 simulators exist, we decided to support the x86 ISA from the beginning of the project. Though we believe it has been the right long-term decision, the effort to get x86 to work has been tremendous. We believe we could have saved at least one and a half years and maybe two years had we not done x86. Variable length instructions make the Fetch stage extremely difficult. For example, an accurate branch predictor has to guess the boundaries between instructions and then needs to make sure that the instructions being fed from the functional model are actually on those boundaries (one might mispredict the boundaries for example). Since x86 allows access to bits [15:8] of a 32-bit register as a separate 8-bit register, a more complex reservation station structure is necessary. Writing microcode for each x86 instruction is a task that could be tremendously large and occupies large teams of some of the best engineers within x86 companies. Supporting the REP prefix is a hassle since such modified instructions can generate a huge number of micro-ops and can be interrupted in the middle.

1) Full Operating Systems: Supporting full operating systems introduces significant complexities that complicate our simulator. For example, handling interrupts and exceptions require our system to continue *beyond* where the functional model would normally handle them, since the functional model would naturally handle an exception immediately but in a real system, the exception would be handled when the instruction is the oldest

instruction.

D. Building on top of Unreliable Platforms

The DRC platform was initially unstable, both due to our bugs as well as some issues within the DRC system. By introducing a very simple XOR-based checksum between the functional model and the timing model, we were very quickly able to identify problem causes, whether they were in the functional model, the DRC interface or the timing model. Even though the DRC box runs reliably today, we still pay the performance penalty of keeping the checksum in the system to watch for future problems. We use a simple XOR checksum. We were originally concerned that it would not be sufficient, so we tried a CRC32-based checksum (costing a 20% functional model speed penalty) but, so far, XOR has caught all of the bugs we have encountered.

E. FPGA Observability

As one might imagine, FPGA observability is extremely poor. Chipscope permits the observation of 256 signals, but we have experienced both probe effects as well as timing issues when Chipscope has been inserted. We generally fall back to Bluesim/Verilog sim to find bugs.

F. Differences in Bluesim/RTL/Hardware

We have seen differences between Bluesim simulation (Bluespec supports simulation in software that is significantly faster than the Verilog event-driven simulation), RTL simulation and hardware. Our earliest example was traced down to uninitialized block RAMs that Bluesim initializes to 0, but hardware does not. We solved that particular inconsistency by including a state machine that initialized block RAMs to 0 whenever reset is asserted.

III. Acknowledgments

This work was partially supported by a Department of Energy Early Career Principal Investigator Award (ER25686), the National Science Foundation (0615352, 0541416), SRC, grants and equipment donations from Intel, equipment (including one DRC Computer system and the loan of another) and software donations from Xilinx, Faculty Awards from IBM, and a gift from Freescale. We would like to thank all of them for their generous support.

We would also like to thank Joel Emer of Intel for very helpful discussions and strong support.

References

[1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[2] Derek Chiou. FAST: FPGA-based Acceleration of Simulator Timing models. In *Proceedings of the first Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-11, San Francisco, CA*, February 2005.

[3] Derek Chiou, Huzefa Sanjeliwala, Dam Sunwoo, John Zheng Xu, and Nikhil Patil. FPGA-based Fast, Cycle-Accurate, Full-System Simulators. In *Proceedings of the second Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-12, Austin, TX*, February 2006.

[4] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil Patil, William Reinhart, Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of MICRO*, December 2007.

[5] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William H. Reinhart, D. Eric Johnson, and Zheng Xu. The FAST Methodology for High-Speed SoC/Computer Simulation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, November 2007.

[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.

[7] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, October 1998.