FPGA-based Fast, Cycle-Accurate, Full-System Simulators

Derek Chiou, Huzefa Sunjeliwala, Dam Sunwoo, John Xu and Nikhil Patil
University of Texas at Austin, Electrical and Computer Engineering
{derek,sanjeliw,sunwoo,zxu,npatil}@ece.utexas.edu

***Abstract*— An ideal computer simulator is (i) *fast*, (ii) *accurate* to cycle level resolution, (iii) *complete*, modeling the entire system and running unmodified applications and operating systems, (iv) *transparent* providing visibility into all aspects of the system with minimum impact to simulation performance, (iv) *inexpensive* and (v) *easy* to create, extend and modify. Conventional wisdom says that no simulator can simultaneously have all these properties[1] and none currently does. Instead, simulators are specialized, emphasizing some desired properties over others. For example, architectural simulators traditionally trade speed for accuracy while full-system simulators traditionally trade accuracy for speed.**

This paper describes an approach to simulation that potentially has all of the characteristics of an ideal simulator listed above. It achieves its capabilities by partitioning a simulator into a software component and a hardware component implemented in FPGAs. The resulting simulators are capable of 1M to 100M cycles per second, full cycle-accuracy, the ability to run unmodified applications and operating systems and full visibility at a reasonable price. Such a simulator could potentially result in simulator convergence, where different groups can use the same simulation infrastructure, resulting in more coherent architectures, implementations and software.

I. Introduction

Being able to accurately and quickly predict properties of computer systems is useful for architects, designers, software developers and users of computers. Simulators provide a window into the inner workings of the computer that help foster understanding and enable the accurate evaluation of ideas and theories. Because simulators often do not have the same constraints as a real implementation, they can be made easier to probe, examine and modify.

A myriad of simulators exists. Architects traditionally use software-based cycle-accurate simulators to evaluate next generation processor and system architectures. There are many such simulators in academia and industry[10], [4], [1], [18], [2], [15], [19] and simulator builders[23]. Such simulators are *transparent* and can be *accurate* but are generally slow, simulating a single complex processor at around 10K cycles per second. Thus, such simulators are too slow to run real applications on real data sets. Benchmarking[22] and sampling[20], [9] can reduce the number of instructions while attempting to maintain accurate performance prediction capabilities but are still simplifications that can miss complex interactions between the application and the operating system, external events and parallelism.

The only real solution is a cycle-accurate simulator fast enough to run real applications. Even with perfect speedup, however, a software-based cycle-accurate simulator would require tens to hundreds of thousands of processors which is currently infeasible and would be very costly. It is clear that faster cycle-accurate simulators require hardware support.

Several companies such as Cadence (Quickturn and Palladium), Axis, IKOS and Tharas sell field-programmable gate array (FPGA) based accelerators or emulators to improve cycle-accurate simulator performance. In such systems, register transfer logic (RTL) code is compiled for an accelerator/emulator box or card, sometimes in conjunction with components executing in software. Such systems can be *fast* (up to 100M cycles per second), *accurate*, *complete* and *transparent*, but are so expensive that even large companies can only afford a few copies and often not *easy-to-use*. In addition, these systems require a complete version of the RTL, further increasing the cost and time-to-simulation.

There are several projects, many that were presented at the first WARFP, that implement some components of a computer system, such as the processor, memory controller[8], [14] or both[7], [12], [5], [16], in FPGAs and the rest in real hardware. Such systems are *fast* and generally *complete*, but are not *accurate* unless all components are implemented in FPGAs which makes such systems difficult to initially develop and often difficult to modify.

II. University of Texas FPGA-Accelerated Simulation Technologies (UT FAST)

A cycle-accurate computer simulator can be partitioned into (i) a *timing model* that simulates only the micro-architectural structures that affect timing and resource arbitration and (ii) a *functional model* that simulates only the instruction set architecture (ISA). The timing model implements only the *control path* and not the datapath, making the model itself extremely small, simple and easy to build up out of composable functional units. For example, the timing model only implements cache tags and not the data itself. ALUs become pipeline stages. Likewise, the functional model knows nothing about timing issues, also making it small and easy to extend.

This partitioning is not novel. For example, Asim[10] and Simplescalar[1] are both partitioned in this fashion. However, since both components are running on the same processor in such simulators, no effort was made to minimize the communication between the two models. In such simulators, there is also much more processing done in the timing model due to the large number of parallel structures to simulate than

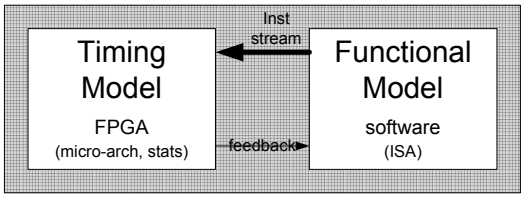

Fig. 1. A High-Level View of a FAST simulator

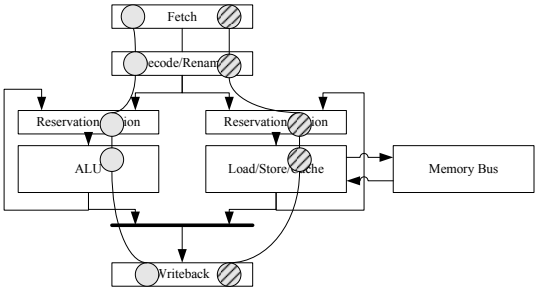

Fig. 2. A simple superscalar processor implemented in FAST. The solid white circles indicate the DGR of an `add` instruction while the lined circles indicate the DGR of a `load` instruction.

in the functional model. In addition, most statistics gathering is done within the timing model since it produces performance information, further slowing down the timing model.

FAST attacks the performance bottleneck inherent in software simulators by implementing the timing model in hardware. Hardware is inherently parallel and thus can efficiently implement the parallel structures found in the timing model. The separation between functional model and timing model eliminates the need to support the data path, dramatically reducing the hardware resources required. Thus, very complex processors can fit in a relatively small amount of hardware. In addition, hardware can be applied to gather statistics, enabling full speed statistics gathering and processing.

Unlike existing software-only simulators, FAST simulators are carefully optimized to minimize communication between the functional model and timing model. By doing so, the two models can be more loosely coupled and thus be implemented in different technologies with little impact on performance.

Unlike other hybrid FPGA/software simulators that are partitioned on target system module boundaries such as a cache, a floating point unit or even a processor core, FAST is partitioned on the simulator boundary between the functionality and timing. This architecture pushes all of the hard-for-software-easy-for-hardware components to hardware and the hard-for-hardware-easy-for-software components to software, minimizing bottlenecks.

A. FAST Basics

An instruction set simulator capable of booting operating systems and running unmodified applications is used as the functional models. Existing simulators such as Simics[11], SimOS[18], Bochs[13], QEMU[17], M5[2], Mambo[4] and AMD's SimNow[21] can all serve as full-system functional models. Such functional simulators are capable of running in excess of 100M instructions per second, though not all do.

Figure 1 is a high-level view of a FAST simulator. The functional model implemented in software pipes an instruction stream to the timing model implemented in FPGAs[1]. The functional model's job is to decode and execute one instruction at a time and generate a "perfect" instruction stream with correctly resolved branches. It pipes the decoded instruction including a decoded opcode, source and destination register names, instruction and data virtual addresses and data written to special registers such as software-filled TLB entries. Additional and/or redundant information, such as physical addresses, can also be passed from the functional model to the timing model to further simplify the latter at the expense of a larger instruction stream.

Each decoded instruction is "executed" by the timing model by arbitrating for and consuming the required resources in the correct order as required by the instruction. The timing model transforms the decoded opcode, by some combination of table lookup and combinational logic, into a directed graph of resources (DGR) used to execute the instruction[2]. An interesting side-effect of the DGR method is that any ISA can be mapped to a timing model that is capable of supporting that ISA. For example, a Pentium M timing model could run a PowerPC ISA by changing only the opcode-to-DGR mapping.

Such a simulator is essentially the same as a classic trace-driven simulator with the software-based timing model replaced by an FPGA-based timing model. Such simulators are capable of simulating non-speculative, non-parallel thread target system micro-architectures. For example, a trace-driven simulator could simulate a micro-architecture based on a classic Tomasulo algorithm with no branch prediction even though it does exploit parallelism in a single instruction stream, but fundamentally does not change the instruction stream. For such systems, the communication latency between the functional model and timing model are not important since there is no feedback, making a hardware timing model an obvious performance win.

B. A Simple Example

Figure 2 shows an example of a very simple non-speculative, single-issue superscalar processor and the DGR for two instructions, `add` and `load`. Each instruction must first traverse the FETCH unit where the instruction TLB and L1 cache are checked and will stall upon a miss. The instruction then traverses the DECODE unit where it is determined if the read register values are available (the actual values

[1]Though initial implementations use a software-based functional model, hardware-accelerated and hardware implemented functional models[16] are possible. The entire FAST simulator can also be implemented in pure software if FPGA-board availability is an issue.

[2]The DGR can be complex. For example, each node in the DGR can have multiple output arcs that can be conditionally traversed and multiple input arcs of which a configurable subset may fire the next node. There may be looping in a DGR as well to implement complex instructions.

are not stored, but a presence bit is) and appropriate register renaming is performed. Then the `add` instruction goes to the ALU reservation station where it waits for any pending arguments, if necessary, and then is passed to the ALU. Once the `add` instruction completes, it gets put on the bypass buses to the ALU and the LOAD/STORE unit and arbitrates for the WRITEBACK unit to be written back (set presence bit to present in the register file.)

After the DECODE unit, the `load` instruction goes to the LOAD/STORE unit where the effective address is computed, the data TLB and cache are checked and stalls upon a miss. Once the `load` completes in the memory hierarchy, the result gets put on the bypass buses and the result arbitrates for the WRITEBACK unit to be written back.

C. Simulating Speculative Processors

The FAST simulators described above are simple but incapable of simulating realistic processors that predict branches. The basic problem is that speculative processors will execute down an incorrect instruction path, fetching and issuing misspeculated instructions, until the misspeculation is discovered and corrected. The functional model, however, will naturally execute the correct path and thus will not normally produce misspeculated instructions.

FAST simulators resolve this issue by implementing the simulated branch predictor and forcing the functional model to mis-speculate when necessary. If the timing model implements the branch predictor, it notifies the functional model of mis-predicted branches so that the functional model can then issue instructions from the wrong path to drive the timing model. Mis-predicted branches are always resolved by the timing model. When a branch is resolved, the timing model must again notify the functional model of that fact so that the functional model can continue executing down the correct path. This notification requires a communication path from the timing model to the functional model. That path now introduces a loop where communication costs matter. The shorter the time between the timing model indicating a branch mis-prediction to the functional model issuing speculative instructions is critical for performance. That time is dependent on the size of the messages and the latency and bandwidth of the communication channel.

To mis-speculate and resume, the functional model supports rollback. The two rollbacks are signaled using a `set_pc` command from the timing model to the functional model that rolls back to a particular past instruction and forces its program counter to a specified value. FAST currently relies on software checkpoints of the functional model to implement rollbacks. Depending on the relative overhead of a checkpoint, the timing model can cache the correct path trace to minimize the functional model roll back on recovery. We are experimenting old value logging to make rollback faster. The overwritten values can either be saved by software or passed to the timing model as part of the instruction stream. In the latter case, the timing model can, upon rollback, return only the most recent overwritten value per modified register or memory location. Doing so dramatically reduces the cost of rollback at the cost of bandwidth between the functional model and timing model.

There are ways to dramatically reduce the need for rollback as well. For example, our initial implementation models simple branch predictors in the functional model. It is not precisely accurate due to timing variations in updating the branch prediction structures and thus is a branch predictor predictor. The functional model assumes that its branch predictor is correct and either executes a conservative number of wrong-path instructions or executes until it is told to stop by the timing model. In addition, the timing model continues to model the real branch predictor with timing and is capable of notifying the functional model when its branch predictor is incorrect. The accuracy of the branch predictor predictor, however, is very high and thus almost eliminates rollbacks.

One may wonder why the timing model does not need to be rolled back as well. Since branch prediction is performed at the head of the pipeline, the timing model can avoid being polluted by wrong instructions being issued from the functional model by stalling and waiting for the right (but incorrect) instructions to be passed to it. Interestingly, microprocessor architecture and optimizations are generally beneficial to FAST simulators as well.

D. Functional/Timing Model Interaction

FAST performance hinges on the interface between the functional model and the timing model. Care must be taken to maximize the performance of the communication protocol, its implementation and the physical link. Most of the communication flows from the functional model to the timing model as instructions written to an instruction buffer. There is a set of command queues from the timing model to the functional model (i) to commit instructions and thus free up instruction buffer slots and (ii) to force instruction execution down wrong path or to return instruction execution back to the right path necessary for simulation speculation or parallelism.

The size of each instruction in the trace is important and can be aggressively compressed. For example, an uncompressed representation of x86 instructions takes about 32B. Such an instruction encodes virtually everything from a flattened opcode to instruction and data virtual and physical addresses to source and destination registers, regardless of whether they are used. Compression techniques such as mirroring simulation structures like TLBs and translated instruction buffers and address compression[6] can dramatically reduce the average number of bytes per instruction, perhaps by a factor of eight or more, at the cost of increased complexity in the functional model. In addition, techniques used in high-performance parallel systems to improve the communication can also be used to improve communication between the functional model and the timing model.

III. FAST Implementation Status

Our inital set of FAST simulators supports the IA32 instruction set. We are currently using a heavily modified Bochs[13] that supports rollback via checkpoint as the functional model.

Thus, the simulators can boot unmodified operating systems and run unmodified applications. It runs on both standard Linux boxes as well as on the embedded processor within a Xilinx Virtex II Pro part. Our modified Bochs is running at about 3.86 MIPS on a 3.0GHz Pentium 4 and about 60K cycles per second on the 300MHz, in-order embedded PowerPC on the Xilinx part. The functional model is currently unoptimized giving us much headroom to improve. Since we do no yet have a PCI-Express FPGA board, we are running Bochs on the embedded PowerPC in the FPGA.

Our modified Bochs is being created by a script that automatically parses and modifies Bochs to include instruction tracing, checkpointing and rollback. The script permits us to quickly adapt to new Bochs releases.

We are on track to completing a timing model that simulates a simple out-of-order, branch-predicted processor core that supports reservation stations, multiple functional units, virtual memory, a full memory hierarchy, DRAM and disks by March of this year. It is designed to support CMP/SMP configurations as well. The model is being written in Verilog and Bluespec[3]. We have already completed a simple 80486-like architecture with separate L1 instruction and data caches and separate instruction and data TLBs that runs at 100MHz when running without a functional model.

Preliminary estimates indicate that a timing model for the largest current Pentium M with a 2MB cache fits in a single, medium/large FPGA (Virtex 4 FX60). It is very likely that multiple timing models could fit in a single FPGA.

IV. Conclusions

The FAST approach has strong potential to produce very fast, accurate, complete and transparent simulators that are inexpensive and easy-to-use. FAST simulators are capable of efficiently simulating almost all general-purpose speculative processors as well as multiprocessors. FAST simulators do so by carefully partitioning the simulation problem into a functional model responsible for simulating at the ISA and functional peripheral level and a timing model responsible for modeling micro-architectural structures that impact timing. By partitioning functionality and timing, each module within each model becomes significantly simpler than an integrated solution, making them easier to create, modify and maintain. The resulting simulators are faster than pure software simulators since they implement highly parallel micro-architectural constructs in hardware, but leave the nearly sequential functional computation to microprocessors more suitable for those tasks.

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] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commerical Workloads (CAECW)*, February 2003.

[3] Bluespec webpage. http://www.bluespec.com.

[4] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the powerpc architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[5] Jared Casper, Ronny Krashinsky, Christopher Batten, and Krste Asanović. A Parameterizable FPGA Prototype of a Vector-Thread Processor. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, February 2005.

[6] Daniel Citron and Larry Rudolph. Creating a wider bus using caching techniques. In *In Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 90–99, 1995.

[7] 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*, February 2005.

[8] John D. Davis, Lance Hammond, and Kunle Olukotun. A Flexible Architecture for Simulation and Testing (FAST) Multiprocessor Systems. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, February 2005.

[9] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2004.

[10] 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.

[11] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. In *IEEE Computer*, pages 50–58, February 2002.

[12] Christos Kozyrakis and Kunle Olukotun. ATLAS: A Scalable Emulator for Transactional Parallel Systems. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, February 2005.

[13] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es):7, 1996.

[14] Shih-Lien Lu, Eriko Nurvitadhi, Jumnit Hong, and Steen Larsen. Memory Subsystem Performance Evaluation with FPGA based Emulators. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, February 2005.

[15] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alamelden, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. submitted to Computer Architecture News.

[16] Eriko Nurvitadhi and James Hoe. Full-System Architeectural Exploration Sandbox. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, February 2005.

[17] QEMU webpage. http://fabrice.bellard.free.fr/qemu.

[18] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7(1):78–103, 1997.

[19] Lambert Schaelicke and Mike Parker. ML-RSIM Reference Manual. Technical report, Department of Computer Science and Engineering, Notre Dame, 2002.

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57. ACM Press, 2002.

[21] SimNow webpage. http://developer.amd.com/simnow.aspx.

[22] SPEC webpage. http://www.spec.org.

[23] Manish Vachharajani, Neil Vachharajani, and David I. August. The liberty structural specification language: a high-level modeling language for component reuse. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 195–206. ACM Press, 2004.