Preprint: To Appear in MICRO 2007.

FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators

Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil,
William Reinhart, D. Eric Johnson, Jebediah Keefe and Hari Angepat
The University of Texas at Austin
{derek,sunwoo,turo,npatil,wreinhar,dejohnso,jkeefe,angepat}@ece.utexas.edu

Abstract

This paper describes FAST, a novel simulation methodology that can produce simulators that (i) are orders of magnitude faster than comparable simulators, (ii) are cycle-accurate, (iii) model the entire system running unmodified applications and operating systems, (iv) provide visibility with minimal simulation performance impact and (v) are capable of running current instruction sets such as x86. It achieves its capabilities by partitioning simulators into a speculative *functional model component that simulates the instruction set architecture and a timing model component that predicts performance. The speculative functional model enables the simulator to be parallelized, implementing the timing model in FPGA hardware for speed and the functional model using a modified full-system simulators. We currently achieve an average simulation speed of 1.2MIPS running x86 applications on x86 Linux and Windows XP and expect to achieve 10MIPS over time. Such simulators are useful to virtually all computer system simulator users ranging from architects, through RTL designers and verifiers to software developers. Sharing a common simulation/design infrastructure could foster better communication between these groups, potentially resulting in better system designs.*

1. Introduction

The ability to accurately, quickly and easily predict properties of computer systems is useful for computer architects, designers, software developers and users. Simulators provide a window into the inner workings of the computer that helps promote understanding and enable the accurate evaluation of ideas and theories. Because simulators are not subject to the same constraints as a real implementation, they are easier to create, modify and observe.

Good simulators are (i) *fast* (ii) *accurate*, accurately predicting whatever metrics are being measured (in this paper, we focus on predicting performance), (iii) *complete*, modeling the entire system and able to run unmodified applications and operating systems, (iv) *transparent*, providing visibility into the simulated system, (v) *inexpensive*, (vi) *current*, running current ISAs such as x86 and modeling current mainstream microarchitectures and (vii) *easy-to-use*. Many of these properties, however, conflict with each other, necessitating simulators specialized for a specific application. For example, architectural simulators often trade speed for cycle-accuracy while full-system simulators often trade cycle-accuracy for speed.

Industrial and academic architects traditionally use software-based cycle-accurate simulators to evaluate next generation processor and system architectures[28, 21, 16, 7, 34, 2, 26, 5, 19, 27, 31, 32]. Such simulators are *transparent*, *easy-to-use* and can be *cycle-accurate* but are generally not *fast* or *complete* and often not *current*. Intel's[15] and AMD's[4] fastest true cycle-accurate x86 simulators run at 1KHz to 10KHz which translates to two minutes of simulated time in approximately one to ten years of simulation time. At such speeds, it is impractical to use real program runs to explore, evaluate and refine microarchitectures.

Benchmarking and sampling[29, 33, 14] reduce the number of executed instructions to speed up simulation runs. Though such techniques can be effective, they rely on simplifying assumptions that, if incorrect, produce incorrect results. As (i) complex interactions between applications and the operating system, (ii) the number of external events, (iii) parallelism and (iv) the potential performance impact of a rare event all increase over time, what might initially appear to be a reasonable simplifying assumption might result in significant prediction inaccuracies.

A fast and accurate simulator could run a real software stack on real data, thus avoiding the potential unseen inaccuracies of benchmarking/sampling. In this paper, we explore a novel way to create fast and accurate simulators.

Since modern computer systems use extensive parallelism and high clock frequencies to achieve high perfor-

mance we argue that parallelism is required to significantly improve cycle-accurate computer simulator performance. Simulators, however, have traditionally resisted parallelization. There are, for example, no commercial parallel Verilog simulators. Recent studies of parallelizing processor simulators[24, 12] have yielded only modest speedups, in the two to six range, probably due to the tight interdependence of parallel activity in modern microprocessors.

Because hardware inherently exploits the fine-grain parallelism that pervades cycle-accurate simulators, a hardware host (we use the term *host* to mean the system that runs the simulator and the term *target* to mean the system being simulated) could support a significant amount of parallelism and, therefore, high simulation speeds. Hardware is, however, difficult and time consuming to develop. To make hardware-based simulation viable, it is necessary to simplify the hardware development process. One potential simplification is to implement part of the simulator in software and part of the simulator in hardware. Incorrect partitioning, however, could result in lower performance than a pure software simulator. An Intel experiment that moved the Simplescalar sim-outorder L1 data cache into a field-programmable gate-array (FPGA) sitting on the front-side bus of the host Pentium III processor (as close as an FPGA can get to a standard processor) produced lower performance than the original, unmodified Simplescalar[30].

This paper describes FPGA-Accelerated Simulation Technologies (FAST)[8, 9, 10] simulators, a new class of simulators that use FPGAs to generate very *fast*, *complete*, *cycle-accurate* and *current* simulators that are relatively *inexpensive*, *transparent* and *easy-to-use*. FAST simulators are based on a variant of the well-known functional model/timing model partitioning that is used in many pure software simulators including FastSim[28], TimingFirst[20], Asim[16], an IBM Power simulator[21] and certain versions of Simplescalar[2] and M5[5]. FAST simulators are different than traditional functional/timing partitioned simulators because they leverage two novel realizations: (i) the communication between the functional and timing partitions can be made latency-tolerant, allowing the functional model to run efficiently in parallel with the timing model and (ii) that the timing model is very parallel and very silicon-efficient compared to a full implementation of the target architecture, enabling most timing models to be implemented in a single FPGA.

The next section gives an overview of the architecture of FAST simulators. Section 3 then gives important details about how FAST simulators run in parallel and how the full system, including interrupts and exceptions, is modeled. Section 4 gives status and performance of our prototype cycle-accurate simulator that models an out-of-order, branch-predicted processor, executes the x86 ISA and runs unmodified applications on top of unmodified Linux and Windows XP. Though the FAST methodology also supports multiprocessor simulation, due to space constraints this paper addresses only uniprocessor simulation. Section 5 compares and contrasts FAST to closely related work. Section 6 gives some future work and conclusions.

2. FAST Partitioning

There is a wide range of functional/timing partitioned simulators, each with different tradeoffs. In this section, we first give a brief overview of the partitioning and then describe the specific partitioning used in FAST.

The functional model (i) simulates the computer at the functional level including the instruction set architecture (ISA) and peripherals, and (ii) executes application, operating system and BIOS code. The timing model simulates only the microarchitectural structures that affect the desired metrics. For example, to predict performance, we need to model such structures as pipeline registers, arbiters and associativity. Because data values are often not required to predict performance, data path components such as ALUs, data register values and cache values are generally not included in the timing model. Even much of the control structure, such as decoding, can be simplified using information from the functional model. The orthogonality between functional and timing models simplifies each significantly.

The functional model sequentially executes the program, generating a *functional path* instruction trace, and pipes that stream to the timing model. It is often the case that the functional path is equivalent to the right path where branches are always correctly predicted. Each instruction entry in the trace includes everything needed by the timing model that the functional model can conveniently provide, such as a fixed-length opcode, instruction size, source, destination and condition code architectural 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 or data, can also be passed in the trace to further simplify the timing model at the expense of a larger trace.

In the example shown in Figure 1, the functional model executes and outputs eight instructions to the timing model via the *trace buffer* (TB). Each logical TB entry contains information used by multiple stages in the timing model and is thus not deallocated until the instruction is fully committed. Note that the instructions are not necessarily produced by the functional model in lockstep with the timing model. The target is a single issue machine with three functional units, ALU (+), Load/Store-DataCache ($) and Branch (B), and the ability to write up to three instructions (one per functional unit) to the ROB per cycle. The timing model first "fetches" from the TB, then cycle-by-cycle "processes" each instruction by arbitrating for and consuming the re-

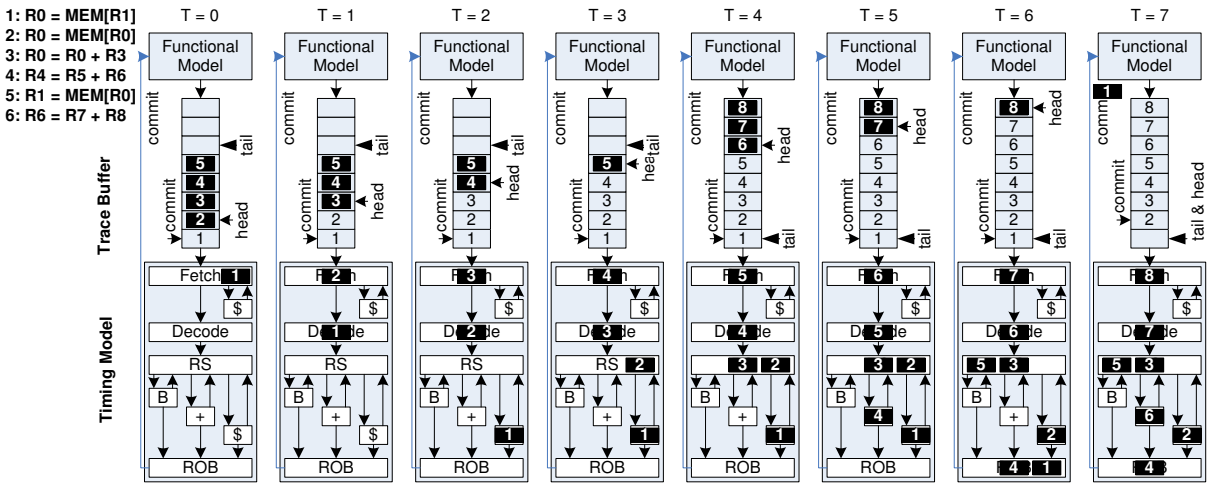

Figure 1. An Example of a FAST Simulator

quired resources in the correct order, thus accurately predicting what would happen in the target microarchitecture.

At $T = 3$, I_2 waits in the reservation station, blocked by a dependency on I_1 and a functional unit hazard. At $T = 4$, I_3 waits in the reservation station for I_2. At $T = 5$, I_4 goes directly to the ALU since it has no dependencies. At $T = 6$, I_1 and I_4 complete and go to the ROB, while I_2 accepts I_1's bypassed data and goes to the DataCache unit. At $T = 7$, I_1 is committed by the ROB that informs the TB to deallocate the TB entry by advancing the commit pointer and the FM for roll back management (see Section 3.2.)

The simulation is very accurate because it models all register-to-register transitions in a cycle-by-cycle fashion. It is, however, very computationally-intensive since every instruction is moved through every pipeline stage, each of which might perform multiple associative lookups, arbitrations, etc. Generally, much more computation is done in the timing model than in the functional model.

2.1. Target and Host Speculation

We call the dynamic instruction stream that would be fetched by the target microarchitecture the *target path* or *correct path*. The functional path is sometimes different than the target path, such as when a branch is mis-speculated causing wrong path instructions to be fetched and executed until the mis-speculation is resolved. The functional model must somehow determine when the two paths diverge and change its own path to conform to the target path. Instructions on the functional path that are not on the target path are called *incorrect path instructions*.

FAST simulators simulate the branch predictor to determine when mis-speculation and branch resolution occurs. Since most branch predictors depend on timing information, the branch predictor must be implemented in the timing model, though a branch predictor predictor can be implemented in the functional model to keep the functional path as similar as possible to the target path. On a branch misprediction, the timing model notifies the functional model to produce the correct wrong path instructions. On a branch resolution, the timing model notifies the functional model to produce the correct right path instructions.

To facilitate communication, every dynamic instruction passed from the functional model to the timing model is assigned an instruction number (IN). The functional model supports a `set_pc` command that takes two arguments, an IN and a program counter (PC). Calling `set_pc` rolls back the functional model to that IN, removing the effects of that instruction, changing to the new PC and then executing from that PC on. `set_pc` can be used to correct path divergences, including those caused by branch mis-speculations and resolutions. The more accurate the *target* speculation, implying a faster target, the less the paths diverge and the faster a FAST simulator simulates that target.

The timing model does not need to be rolled back due to mis-prediction. Since branch prediction is performed at the head of the pipeline, the timing model ignores incorrect path instructions and stalls until correct path instructions arrive.

Figure 2 shows an example of how mis-speculation is handled in a FAST simulator. I_2 (I is an instruction pointer, not the dynamic instruction number IN) is a branch that is mis-speculated. At time $T = 1$, the functional model is notified that I_2 is mis-speculated and to execute I_{4*} (we use either a "*" or a dark border to indicate a mis-speculated instruction) next. The timing model stalls until the wrong path instructions arrive. At $T = 1 + m$ (the "1" is the target cycle and the "m" indicates a later host time) the functional model has written two mis-speculated instructions, I_{4*} and

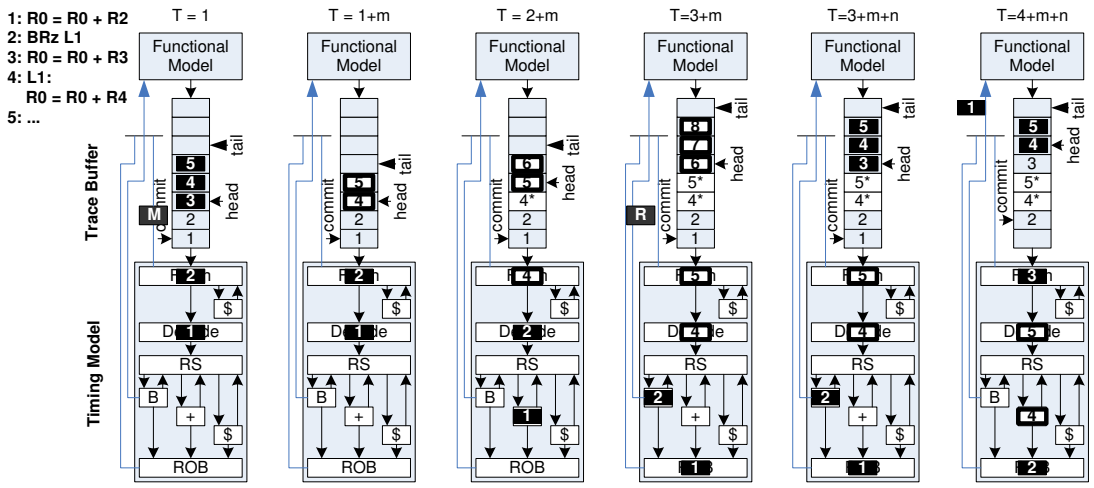

Figure 2. Handling Mis-Speculation

I_{5*} to the trace buffer, overwriting the incorrect path instructions I_3, I_4 and I_5. During the next two cycles, the timing model fetches I_{4*} and I_{5*} respectively and feeds them to the pipeline. At time $T = 3 + m$, the timing model resolves the branch and notifies the functional model that then produces correct path instructions overwriting the incorrect path instructions by time $T = 3 + m + n$. The next cycle, the timing model fetches the next instruction, pushes it to Fetch and commits I_1 by advancing the commit pointer. The timing model notifies the functional model of commits so that the functional model can release rollback resources.

Out-of-order targets are simulated in the same way as in-order targets. Since out-of-order targets fetch in-order, the functional path is still likely to closely match the target path. OOO processors provide the illusion that instructions are executed in-order even if they execute out-of-order, meaning the in-order execution of the functional model will still likely be functionally correct. Consequently, out-of-order execution alone does not cause roll back[1].

3. Making FAST Fast and Complete

The main contribution of FAST is how the functional/timing partitioning is leveraged to efficiently parallelize simulators at two levels: (i) within the timing model and (ii) between the timing model and the functional model. Timing models are very lightweight, requiring few resources, but require very frequent, low-latency communication between timing model modules making hardware an ideal host platform. To the best of our knowledge, we are the first to propose implementing such timing models in hardware/FPGAs. We believe that the FAST prototype contains the first implementation of an FPGA-based timing model. The timing model can also be implemented in pure software, at lower performance.

Since the functional model can roll back and thereby clean up any incorrect instructions, it can speculatively proceed, executing each instruction to completion and moving to the next, without immediate feedback from the timing model. The functional model can efficiently run in parallel with the timing model as long as the timing model does not need to resteer the functional model often. To the best of our knowledge, we are the first to propose parallelizing on the functional/timing boundary, leveraging functional model speculation to further relax dependency on communication. We also believe the FAST prototype is the first such simulator.

Statistics gathering and processing can be implemented in hardware, thus avoiding simulation slowdown given sufficient hardware resources. More complex queries that are normally unaffordable in software simulators are also enabled. For example, run-time queries, such as "when does the number of active functional units drop below 1?", can continuously run in hardware at full speed.

3.1. Analytical Model of Simulator Performance

To demonstrate why parallelizing between the functional/timing boundary works well, we present a very simple analytical model of parallel simulator performance based on Amdahl's Law. Partition the simulator into two components, A and B, that run in parallel. Each component takes T_A and T_B seconds per target cycle, including all one-way communication, that is, nothing needs to be imme-

[1] An important exception is parallel access to shared memory, but that is beyond the scope of this paper.

diately returned for the component producing the communication to continue making progress.

Round-trip communication, where one component cannot make progress until a response to a request is received from the other component, occurs at a fraction F of the total number of cycles. The communication latency of that round-trip is L_{rt}.

There may be additional extra work required to handle round-trip communication. For example, in a FAST simulator, rolling back over incorrect instructions is extra work caused by a round-trip communication. The time for that extra work, which is part of the total latency of the round-trip, is $\alpha_{A_A}/\alpha_{A_B}$ for the extra overhead on A for an A/B initiated request and $\alpha_{B_A}/\alpha_{B_B}$ for the extra overhead on B for a A/B initiated request.

C_A (cycles/sec) for A, is:

$$C_A = \frac{1}{T_A + F \times (L_{rt} + \alpha_{A_A} + \alpha_{B_A})}$$

The simulator cycles/sec is the minimum of C_A and C_B.

T, F, L_{rt} and α are all functions of the partitioning, making an efficient partitioning extremely important. A good partitioning can lead to T_A and T_B each being significantly less than the unpartitioned simulator, since partitioning may enable mapping one or more components onto a different, faster execution platform or each component may be better optimized for the current execution platform.

This equation shows why it is difficult to parallelize, on module boundaries, a simulator that combines timing and functionality in each simulator module. In most cases, either (i) one or both modules requires a round-trip almost every cycle, making F large and putting communication costs on the critical path or (ii) T_A and/or T_B are not significantly reduced, limiting the amount of performance improvement. For example, add an infinitely fast FPGA-based L1 iCache (T_B = 0) to a software simulator that runs at 10MIPS (T_A = 100ns) without memory hierarchy simulation. Assuming a target IPC of 1 and L_{rt} = 469ns, the parallelized performance is $\frac{1}{100ns+469ns}$ = 1.8MIPS, or less than one fifth of the original performance. Even if the original simulator was infinitely fast, performance could not exceed 2.1MIPS because of the necessity of a round-trip communication to the FPGA for every instruction. Also, the latency of the FPGA-based cache model (α_{B_A}) must be added to the round-trip latency.

In a FAST simulator, round-trip communication is only required for every branch mis-speculation and resolution. Thus, starting with the same assumptions as above, a 92% branch predictor and a 20% dynamic branch instruction ratio, $F = 0.08 \times .2 \times 2 = 0.032$, resulting in a simulation speed of $\frac{1}{100ns+.032\times469ns}$ = 8.7MIPS. The factor of two accounts for the round-trip for branch mis-predict and the round-trip for branch resolution. If $\alpha_{B_A} = 1000ns$ (approximately five instructions per basic block plus a roll back re-execution of five instructions), then the simulation performance becomes $\frac{1}{100ns+.032\times(469ns+1000ns)}$ = 6.8MIPS. Additionally, the latency of the FPGA-based cache model is not additive, since it runs in parallel with the functional model. Thus, the timing model could model the entire system microarchitecture and not just the cache and still achieve this performance.

3.2. FAST Functional Model

Since the timing model is implemented in an FPGA, it might naturally follow that the functional model should be implemented in the FPGA as well[22]. Though we eventually plan to do so in an attempt to further improve speed, writing a full system simulator that boots unmodified Windows and Linux on top of the x86 instruction set is a non-trivial task. Thus, existing software-based full system simulators are an attractive starting point for a functional model. Full-system simulators are highly tuned for performance and microprocessors are the fastest known hardware structures to execute instruction sets. For these reasons, our current prototype FAST simulator uses a software-based full-system simulator as the functional model. Thus, we support a full system, including network, disk, video, etc.

FAST functional models generate an instruction trace and support a `set_pc` operation. Generating a trace is, in principle, straightforward, since one can generate a trace by augmenting each instruction execution to also dump that instruction to the instruction trace. Performance is impacted due to the additional operations and memory bandwidth required. Some of the performance impact of trace generation can be reduced by compression techniques such as mirroring translation caches (pass just a basic block number and addresses rather than all of the instructions in the basic block) and/or TLBs to remove the need to send physical addresses, compacting opcodes and so on.

We currently support `set_pc` using periodic software checkpoints of architectural state along with memory and I/O logging. At least two checkpoints that leapfrog each other are maintained to ensure that the functional model can rollback to any non-committed instruction. As commits return from the timing model, checkpoints are released and others are taken. Though these modifications are substantial, they only need to be done once.

3.3. Mapping to Hardware

FAST is subject to hardware/FPGA constraints that are, in some cases, more restrictive than in software. For example, only two-ported memory structures are available in current FPGAs. Consequently, a twenty-ported memory needed to implement a particular microarchitecture cannot

be directly implemented in an FPGA. One important point that is often overlooked, however, is that hardware is much easier to implement if performance is a secondary concern. Modern FPGAs run in the 100MHz-200MHz+ range for reasonable designs. Depending on the desired target cycle time, multi-cycle operations can be affordable. For example, a twenty-ported memory can be simulated by cycling a dual-ported memory ten times, resulting in 20MHz performance (in fact, Xilinx block RAMs can be clocked at up to about 550MHz meaning that, with some effort, even better performance can be obtained.) Of course, care must be taken to balance the speed of the timing model with the speed of the functional model to maximize performance.

Using multiple host cycles can dramatically simplify hardware. For example, highly associative caches and renaming multiple instructions per target cycle is trivial if multiple host cycles are used to model a single target cycle. A similar process, performing an entire operation in the first target cycle and then delaying the results to correctly model a multi-cycle target, is commonly done in software simulators to both simplify the implementation and to ease verifying correctness while maintaining accuracy.

3.4. Full System Capabilities

In addition to microprocessors, computer systems include components and peripherals such as memory, disk, network and video. FAST simulators can also simulate such components. The functional model simulates the correct functionality while the timing model predicts component timing. Like FAST processor models, component timing models simulate all the delays, resource arbitrations and queuing components necessary to build an accurate performance model of these structures. For example, accurate disk modeling can be achieved by tracking rotational speed, head position, buffers, and whether the disk is accelerating or decelerating. Thus, FAST simulators are capable of system cycle-accuracy and not just processor cycle-accuracy.

Accurately handling interrupts and exceptions is critical to both full system simulation and true cycle-accurate modeling. The timing model generates interrupts for reproducibility and passes those interrupts to the functional model. Exceptions could either be produced by the timing model (e.g., TLB misses handled in software) or by the functional model (e.g., arithmetic exceptions.) If the functional model discovers an exception, it indicates that in the instruction trace. It is, however, the responsibility of the timing model to signal *when* an interrupt/exception occurs. When the timing model detects an interrupt/exception at the appropriate place, such as when an instruction that will cause an arithmetic exception reaches the ALU, it freezes, notifies the functional model to start generating the interrupt/exception handler instructions and waits until those instructions arrive in the trace buffer.

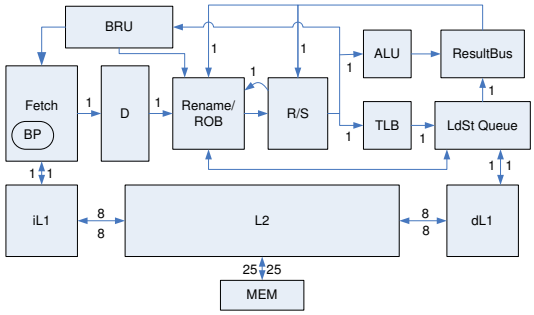

Figure 3. Target μArch With Default Delays

4. FAST Prototype

We are in the process of implementing a prototype FAST system. We have not yet started aggressive performance or area tuning. Also, due to the current communication interface that will improve over time, we are paying a round-trip communication cost every two basic blocks rather than twice per mis-predicted branch. Thus, the numbers presented here should be seen as a lower bound on the potential of the FAST approach.

Our current prototype executes the x86 ISA, boots unmodified Linux kernel versions 2.4 and 2.6 and unmodified Windows XP. Though any application that would run on those operating systems should run on the prototype, we have tested SPECINT2000, MySQL running some test cases and the Department of Energy's Sweep3D benchmark on Linux.

The prototype's target microarchitecture is shown in Figure 3. The major stages include Fetch (with a gshare branch predictor, an iTLB and an L1 instruction cache), Decode, Rename/ROB, r Reservation Stations, n general-purpose ALUs, b Branch Units and a simple delay model of memory. We can support a configurable number of nested branches. Our target is configured as a two-issue single core with eight-way 32KB L1 instruction and data caches, an eight-way 256KB shared L2 cache, 64 ROB entries, 16 shared reservation stations, 16 load/store queue entries, a 4-way and 8K BTB gshare branch predictor, multiple branch units, one load/store unit, eight general-purpose ALUs and up to four nested branches. The pipeline is between eight and ten stages deep, not including accesses beyond the L1 caches.

The functional model is based on QEMU[3], an open source full-system simulator capable of executing x86 as well as x86-64, PowerPC, Sparc and ARM ISAs. We heavily modified QEMU to support instruction trace and roll back, including across I/O operations. We have compressed opcodes to 11bits and instructions down to an average of

about four 32bit words per x86 instruction.

The timing model executes on an FPGA. It is constructed from configurable hierarchical Modules. The base Modules consist of structures such as CAMs, FIFOs, memories, registers and arbiters (currently LRU and round-robin) from which are built caches and load/store queues from which are built branch predictors (currently perfect, 2b saturating and gshare), from which are built our top-level modules, Fetch, Decode, Rename, Reservation Stations, ALUs, BranchUnit, Load/StoreUnit and ROB. Like many other simulators such as Flexus[32], modules of the same type are interchangeable, enabling quick configuration changes.

Modules are connected by Connectors[10] which are FIFOs that enforce timing and throughput constraints. Connectors can be configured for input throughput, output throughput, minimum latency and maximum transactions and will also provide statistics gathering and logging capabilities. By specifying parameters to a Connector, one can do such things as reconfigure a target from a single issue machine to a multi-issue machine (increasing throughput and maximum transactions on each Connector), change the latency or change the number of outstanding transactions allowed. Using such a scheme, one can quickly and easily explore a wide range of microarchitectures.

The timing model is written in Bluespec[6], a high-level hardware description language. Bluespec provides powerful features, including parameterized types, that enable components to be written with parameterized interfaces that are specified during instantiation.

We have successfully incorporated the FAST prototype into Intel's Architect's WorkBench (AWB)[17], the Asim infrastructure. By doing so, we have the potential to use Asim configuration and statistics viewing capabilities and have also enabled easy integration with other Asim components.

4.1. Prototype Limitations

The current prototype has some limitations. Caches are currently blocking. Resolving mis-predictions currently require flushing the pipeline through the ROB before right-path instructions can enter the pipeline. Our simple gshare branch predictor has fairly low branch prediction accuracies. The reservation station implementation limits us to 16 entries. We are working on solving these limitations.

We currently do not model peripherals and DRAM, beyond a fixed delay. We also do not model interrupts and exceptions accurately (though they are handled functionally correctly.) The prototype does not pass in or store data values in the timing model beyond those used by the control path such as software-TLB entries. Thus, we currently do not handle microarchitectures whose performance is data-dependent. The current prototype also does not handle certain architectural mechanisms such as data speculation.

However, FAST can simulate such mechanisms by providing the timing model with a way to detect that the functional model has not correctly mis-speculated on the data and then forcing the functional model down the mis-speculated path with the wrong data. One way to provide detection capabilities is to save some data (or an abbreviation of the data like a checksum) in the timing model to be compared with the data that the functional model used to compute the instruction. Another approach for data-dependent ALU performance is to have the functional model do that computation.

4.2. Execution Platforms

Our primary execution platform is a DRC Computer development platform[13]. This machine contains a dual-socket motherboard, where one socket contains an AMD Opteron 275 (2.2GHz) and the other socket contains a Xilinx Virtex4 LX200 (4VLX200) FPGA. The Opteron communicates to the FPGA via HyperTransport. In our prototype, the functional model runs on the Opteron and the timing model runs on the FPGA. DRC provides libraries to read and write from the FPGA.

We also run on a Xilinx Virtex2Pro FPGA (2VP30) in the low-cost Xilinx University Platform board that contains a pair of embedded PowerPC 405 processors running at 300MHz with the functional model running on the embedded PowerPC processor and the timing model running within the FPGA fabric. Because the serial interface to the embedded processors is less convenient than the DRC box, we currently do all development on the DRC platform and thus do not present numbers from the Xilinx boards.

4.3. Microcode Generation

As is well known, the x86 ISA presents a greater implementation challenge than many other modern ISAs. x86 instructions are variable-length (1B to 15B) CISC instructions that can potentially specify a loop that performs hundreds or thousands of operations. Virtually all modern implementations of the x86 ISA *crack* each x86 instruction into smaller, RISC-like instructions call micro-ops. Our prototype models instruction cracking. The micro-ops for each x86 instruction are stored in a microcode table that is automatically generated by a compiler we have developed. The compiler takes C code that specifies the functionality of each instruction (currently taken from a instruction set simulator) and compiles it into fairly optimized microcode for that instruction on the specified microarchitecture. The compiler was developed to ease the process of (i) porting new ISAs, (ii) generating new instructions and (iii) porting to new microarchitectures with different microcode. Cur-

App	Fraction	μOps/inst
Linux-2.4	95.94%	1.15
164.gzip	99.98%	1.34
175.vpr	84.62%	1.19
176.gcc	99.90%	1.30
181.mcf	99.93%	1.17
186.crafty	98.96%	1.15
197.parser	99.74%	1.27
252.eon	52.32%	1.24
253.perlbmk	98.64%	1.29
254.gap	99.80%	1.31
255.vortex	99.91%	1.21
256.bzip2	99.98%	1.29
300.twolf	95.20%	1.25
Linux-2.6	98.02%	1.45
Sweep3D	44.05%	1.19
MySQL	99.15%	1.51

Table 1. Fraction of Dynamic Instructions Translated to μOps

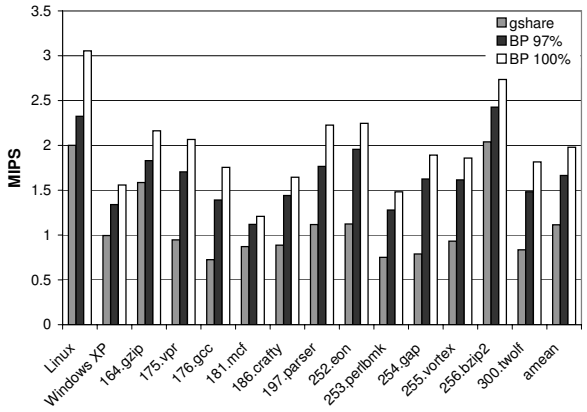

Figure 4. Simulator Performance

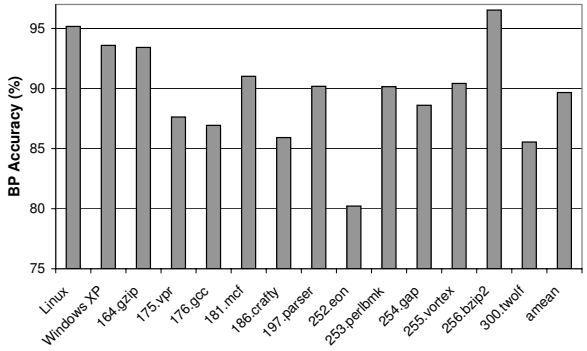

Figure 5. Branch Prediction Accuracy

rently, we generate about 1.27 micro-ops (dynamic average) per x86 instruction. The microcode table is, to first order, a lookup table and can be easily populated with x86-to-microcode mappings from other sources if desired.

Table 1 gives the fraction of the total dynamic instructions executed that have valid microcode. Our microcode compiler generated almost all of the microcode automatically. We currently support only about 25% of the dynamic floating point instructions executed by the SPEC2000 FP benchmarks. `eon` and `vpr` both have a significant number of floating point operations, lowering the number of instructions covered. Although it is not difficult to support these instructions, we have been focusing on the integer benchmarks. Instructions that we do not yet have automatic translation for are either inserted into the table by hand or are replaced with a NOP.

4.4. Prototype Performance

Figure 4 shows the simulation performance of several benchmarks running on the current FAST prototype executing on the DRC platform. The statistics are target path MIPS and thus include requested wrong path instructions, but not incorrect instructions. Figure 5 shows the branch prediction accuracy (including all branches). The FPGA cycle time is 100MHz. Since simulator performance is dependent on branch prediction accuracy, we ran with three branch predictor configurations: Perfect, 97%, and gshare with a four-way, 8K BTB.

Windows XP was quite difficult to get to work on the prototype because it uses a wider range of instructions and touches more devices than Linux does.

One interesting thing to notice is that branch prediction performance is not always correlated with simulator performance. `perlbmk` has reasonable branch prediction accuracy (90.2%) but still performs poorly. The reason is that `perlbmk` makes several calls to the `sleep` and `time` system calls that use the HALT instruction. The default QEMU behavior stops the processor until the timer interrupt fires when it reenables the `perlbmk` application (we are not running any other applications on the system.) During this time, the timing model gets no instructions, reducing MIPS.

`eon` is another example where branch prediction accuracy is below average but performance is about average. The reason is eon uses a large number of floating point operations that are not mapped to microcode. Thus, floating point dependencies within eon are not currently enforced, resulting in higher simulator performance.

4.5. Bottleneck Analysis

The current bottleneck is the timing model. We had not paid sufficient attention to the number of host cycles consumed, resulting in a larger number of host cycles per target cycle than the approximately twenty or so host cycles per target cycle we feel is reasonable. Also, because the target microarchitecture is currently less than ideal, especially in the branch predictor, IPCs range from 0.17 to 0.62. Low IPCs slow down the simulator because even bubbles consume some host cycles and if there are many bubbles, those host cycles add up and become a bottleneck. Improving performance requires both improving the target microarchitecture (e.g., non-blocking caches and better handling of branch mis-speculation) and going over each module to reduce the number of host cycles per target cycle being used. Any module improvements will automatically be incorporated into any future design, while an improved microarchitecture will form the basis of future designs and thus likely automatically incorporated as well.

Because the timing model is currently a bottleneck, we cannot accurately project future performance using numbers generated with hardware. Thus, we also ran QEMU in a variety of configurations and compared the performance with that of the total simulator. All runs, including ones that do not use the FPGA, were done on the DRC System. Unmodified QEMU runs at about 137MIPS (Linux boot). We turned off several optimizations, including block chaining, an assembly softMMU and the real timer interrupt, to make our modifications easier. The performance with those optimizations turned off is 45.8MIPS.

Performance with tracing and checkpointing support running with a software verification test rig emulating a DRC interface is 11.5MIPS. Performance adding a 97% count-based branch predictor that causes rollbacks is 8.6MIPS while a 95% BP is 5.9MIPS. With a software 2bit branch predictor, performance is 5.1MIPS with BP accuracies of 94.8%. If we replace the test rig with an immediate-commit FPGA-based dummy timing model which appears to have perfect branch prediction, performance is 5.4MIPS. With our real Fetch unit and a perfect branch predictor, performance is 4.6MIPS.

We also measured DRC latencies between the Opteron and the FPGA. The numbers are computed by looping on the same operation and taking the total time and dividing by the number of operations.

The minimum latencies are measured by doing operations to registers very close to the I/O pins of the FPGA connected to HyperTransport. A user direct read from the Opteron to the register takes 378ns, a write from the Opteron takes 287ns and a burst write from the Opteron of 20 32bit words takes 13.3ns/word. Reading from our own logic (more realistic) takes 469ns while a write takes 307ns and a burst write of 20 32bit words takes 20ns/word. Currently, the reads are blocking, a serious issue that eliminates the benefits of prefetching and transforms what should be a one-way communication, polling for a `set_pc` or a commit, into a round-trip communication.

Our functional model polls an FPGA queue (1 read for a commit, 2 reads for a mis-prediction) every other basic block, incurring 469ns of latency for each blocking read. Our basic blocks average about 5 instructions for Linux boot. Thus, every 10 instructions we issue a single read (assuming a 100% branch predictor). We use burst writes to write the instruction stream to the FPGA. We average about 20 32bit writes per basic block. Thus, for each pair of basic blocks, we would add 469ns for the polling reads and 800ns for the instruction trace writes. At 11.5MIPS (verification test rig, Linux boot), each instruction takes about 87ns. Thus, for each pair of basic blocks we take 10 * 87ns + 469ns + 800ns = 2139ns. Each instruction takes 2139ns/10 = 214ns, or 4.7MIPS, which is very close to the measured real Fetch, perfect BP run performance of 4.6MIPS.

While the current DRC platform supports uncached IO accesses, future systems are expected to allow for cache-coherent HyperTransport access from the Opteron. Using such an interface, Opteron writes can be buffered up in the cache and automatically written back via coherence when the FPGA reads those locations, reducing trace writes to the speeds of cached writes operations. Likewise, Opteron reads to a shared buffer will read from memory (75ns - 100ns) when there is a new FPGA write to the buffer and will hit in the cache when there are no new FPGA writes. Instruction commits and branch mis-speculates and resolutions will be written by the FPGA into a dedicated cache-line sized location within the Opteron memory. Commit writes will be aggregated, while branch mis-predicts and resolutions will be written immediately. Thus, the cost of a poll, currently 938ns per 14 instructions, will drop to (75ns * 2) + 19ns (estimate to do 19 cached reads) per 20 * 7 instructions = 1.2ns/instruction. Thus, we should achieve performance very similar to the soft timing model, 95% BP performance of 5.9MIPS, assuming a fast timing model.

QEMU can be much more optimized. For example, block chaining (jumping directly from one basic block to the next without returning to the block scheduler) was removed to allow us to implement polling the timing model in the block scheduler. Block chaining could be re-introduced by inserting the FPGA poll code (a load, a compare and a branch) in each block rather than in the block scheduler. By reintroducing optimizations that were turned off for expediency and removing debug code, we believe QEMU performance can be significantly increased.

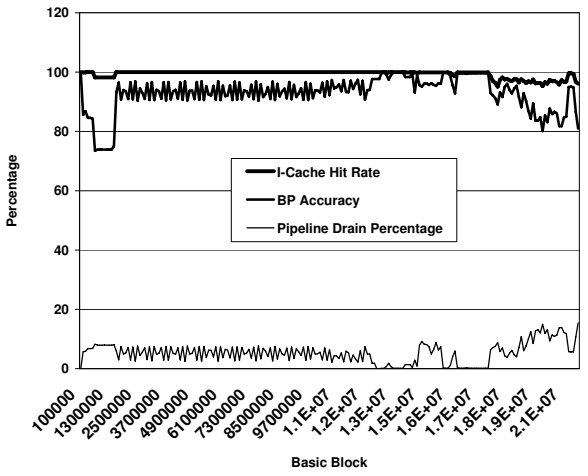

Figure 6. A Statistic Trace

4.6. Statistics Gathering

FAST simulators can gather statistics with little to no simulation performance degradation since hardware can be dedicated to gather and aggregate statistics. Figure 6 shows a combined graph of some of the counter-based statistics that our current prototype gathers. These graphs show three performance metrics while booting Linux: pipe drain cycles due to branch mis-prediction, iCache hits and branch prediction accuracy. The statistics are gathered every 100K basic blocks.

The phases of the Linux boot can be easily seen in the trace. The beginning of the trace is running through the BIOS that is comprised of many branches that are executed only once explaining the large number of branch mispredictions. Since there is a low limit on the number of outstanding branches and a small number of instructions between branches, however, the percentage of time spent in pipe drains are bounded in that region of the code. The relatively flat iCache and BP period of time following is the Linux kernel being decompressed. Then the OS really starts running accounting for decreased BP and iCache hits and increased pipe drains.

Our long term plans include the logging/tracing statistics that will provide substantially more powerful and more configurable statistics gathering capabilities.

4.7. FPGA Issues and Lessons

The FPGA resources required for a complete superscalar processor are fairly minimal. The percentages listed in Table 2 are the percentage of the total number of resources in a Virtex4 LX200 that has 89,088 slices and 336 Block RAMs. We vary the issue width from one to eight throughout the entire pipeline. All configurations met the 100MHz target cycle time without optimization for time or space. In particular, we know the ubiquitous Connectors are under-optimized regarding area, especially in the block RAMs. Thus, we expect these numbers will improve over time.

Issue Width	1	2	4	8
User Logic	32.84%	32.76%	32.81%	32.87%
Block RAMs	50.0%	51.2%	51.2%	51.2%

Table 2. Fraction of a Virtex4 LX200 Consumed by Default FAST Timing Model

The timing model was written initially without significant optimization for FPGA routing and placement. While this allowed rapid development times, such a strategy is not without cost. In particular, while developing a unified statistics tracing fabric, a temporary mechanism was implemented in each Module to track relevant metrics. Collecting and piping this data out of the FPGA required significant global routing resources that limited the number of metrics tracked as well as impacted FPGA timing closure. We are developing a tree-based statistics network that will flow back through the Connectors, ensuring distributed and easy resource routing.

By encapsulating FPGA-specific optimizations in library components such as the statistics network, we are able to maintain rapid development time without sacrificing area/performance. As our simulation library expands through Modules and Connectors, we incorporate these lessons to ensure that future users will not run into similar problems.

A fresh build consisting of a compile (Bluespec $\rightarrow$ Verilog), synthesis (Verilog $\rightarrow$ Netlist) and place-and-route (Netlist $\rightarrow$ bit file) takes a total of about two hours. We are implementing support for incremental builds that should reduce time by only rebuilding what has changed rather than the entire design.

Debugging hardware is definitely more difficult than debugging software. In particular, observability into running hardware is far less than observability into running software. Thus, determining where performance is being lost in the hardware has been difficult. We will address the observability issue by introducing logging/tracing statistics support with triggering (start, stop and dump logs/traces based on user-specified criteria) and user-specified aggregation/compression into the Connectors in the near future.

5. Related Work

FAST is most similar to FastSim[28] that is partitioned across a functional model and a timing model and uses roll

back to return to the right path after executing down the wrong path as required by the target. FastSim's functional model calls the timing model branch predictor every branch to determine if the branch is mis-speculated, in which case the functional model goes down the wrong path immediately. Thus, a FastSim simulator never needs to rollback for a branch mis-speculation, only for a branch resolution. Though a FAST functional model could also either read branch predictor information every branch (round-trip communication) or implement its own branch predictor predictor to make the functional path much more similar to the target path, it is also capable of executing the right path beyond the mis-speculated branch for some time before the timing model corrects it.

FastSim uses instrumented binaries as its functional model, making full-system simulation very difficult to support whereas our FAST prototype uses a full-system simulator as a functional model. Also, FastSim uses simplifications that reduce accuracy for some microarchitectures. For example, when a load/store is selected by the load/store queue to be issued to the cache, a cache simulator is called to determine when that operation will complete. In many modern processors, subsequent load/stores may affect an earlier load/store in the next level of the memory hierarchy.

The biggest difference, however, is that FAST is architected to run in parallel. The authors of FastSim noted that performance of the partitioned simulator was similar to other approaches and thus introduced memoization to fast-forward over previously seen microarchitectural states.

Unlike FAST, Asim[16], Timing-First/Opal[20] and current M5[5] simulators are always driven by the timing model. In Opal/M5, an instruction is executed by the functional model exactly when the timing model indicates it will be executed. In Asim, the functional model is divided into seven components, each component performing its task only when the timing model tells it to. Thus, the functional model does not even fetch an instruction until instructed by the timing model Fetch module. In such a scheme, the functional model never needs to rollback, since each component of the instruction execution is performed at the correct time. However, such a scheme requires continuous communication between the functional model and the timing model. In order to operate, both components must run in essentially lock-step order with each other and generally must round-trip communicate every simulated cycle. It also requires a fairly complex, out-of-order, infinitely renamed functional model. Additionally, some studies, such as perfect branch predictor studies, cannot be done on Asim, since the functional model relies on the timing model for the address of the instruction to fetch, and the timing model cannot know which way a branch will go since it does not implement functionality.

Earlier versions of M5 and some versions of Simplescalar used a scheme that reserved all necessary microarchitectural structures at the time an instruction is fetched. Such a scheme is inherently inaccurate because a later instruction can never contend with an earlier one.

Simulator	ISA	μarch	Speed	OS
Intel	x86-64	Core 2	1-10KHz	Y
AMD	x86-64	Opteron	1-10KHz	Y
IBM	Power	Power5	200KIPS	Y
Freescale	PPC	e500	80KIPS	N
PTLSim	x86-64	Athlon	270KIPS	Y
sim-outorder	Alpha	21264	740KIPS	N
GEMS	Sparc	generic	69KIPS	Y
FAST	x86	generic	1.2MIPS	Y

Table 3. Software Simulator Performance

Though all functional/timing partitioned simulators benefit from the complexity reduction that the orthogonality buys, they are still inherently slow due to the large number of tasks in the timing model required to simulate complex and/or parallel processors. Since the functional models in such simulators consume a very small fraction of the total cycles, their performance is often ignored and under-optimized. The slowness of such simulators often make it impractical to provide full-system support.

We have surveyed and run software cycle-accurate or near cycle-accurate simulators from a variety of different industry and academic sources. We present performance numbers in Table 3. The OS column indicates whether the simulator simulates the full-system.

The first four simulators (Intel[15], AMD[4], IBM[35] and Freescale[18]) are from industry. The numbers reported are averages. PTLSim is an open-source, full-system, cycle-accurate x86-64 simulator that runs purely in software. The reported performance number[34] of 415KHz (270KIPS) is for one benchmark, rsync. Accuracy is claimed to be within 5%. PTLSim is extremely fast for a pure software simulator, but it appears to have already been highly tuned. sim-outorder is the detailed Simplescalar[2] simulator. GEMS[19] is a cycle-accurate simulator from the University of Wisconsin. The sim-outorder and GEMS runs were both done on the DRC platform.

5.1. Hardware Simulation/Prototyping

HASim[11] uses Asim-like partitioning but implemented in fully in FPGAs. Because round-trip communication is required for every simulated target cycle, both the functional model (an infinitely renamed, superscalar processor) and the timing model must be implemented in closely coupled hardware to ensure the low latencies required for high performance.

If RTL is available, which is obviously not possible during architectural exploration, one could retarget it or part of it for one or more FPGAs or other programmable devices. Unfortunately, only the smallest current processors fit into a single FPGA; most take much more. For example, Intel was recently able to fit a single Pentium (3.1M transistors, circa 1993) into the largest Xilinx Virtex4 part (4VLX200). Though most ARM processors will fit into a Xilinx 2V8000 (about half the size of the 4VLX200), a CORTEX-A8 core barely fits into Xilinx's largest FPGA (5VLX330)[25]. FAST simulators, on the other hand, are able to fit a modern processor into the same part and do not require full RTL to run.

Several companies such as Cadence (Quickturn/Palladium), Axis, Mentor/IKOS/Virtual Machine Wires, Synopsys, Synplicity and EVE/Tharas sell FPGA-based accelerators, emulators or tools that take arbitrary RTL and map it to hardware to improve simulation performance. They are often difficult to map to and tend to be very expensive.

The RAMP collaboration[23] is building the necessary infrastructure to build 1000 core machines that run real software including OSs. RAMP systems, however, are cycle-accurate only if the real RTL of all components, including the processors, is used; otherwise, FAST-like timing model technology is required.

6. Future Work and Conclusions

We are continuing to improve our FAST simulator prototype. We are currently fixing our mis-speculation flush and improving the configurability of our generic timing model and will be adding additional timing model modules for system devices such as disk and network. We will then have a fast, full-system uniprocessor simulator that we intend to use to study a variety of topics, starting with the effects of real applications/real operating systems on architectural mechanisms. The long term plan of getting a timing model calibrated to real hardware is to help industry build a model of a modern processor. The speed, accuracy and observability of the simulator should provide us with the ability to observe new phenomena that were difficult to discover in the past. We plan to study applications in order to improve their performance both by changing code and modifying the microarchitecture. We will also evaluate various forms of hardware support for performance observation and tuning.

We plan to add support for multiprocessor targets such as CMPs. We plan to use RAMP-White[1] as a parallel host for the functional model and thus eliminate the slowdown generally associated with simulating parallel targets. The roll back capability of the functional model described in this paper along with other capabilities not described in this paper will be used to reorder memory operations to target order when necessary.

We have started the process of incorporating power estimation into the timing model. The initial goal is not to perfectly estimate power, but to provide relative power estimates that will permit architects to compare different architectures. Such a simulator can also be used by application writers to optimize power algorithms and to better write code that trades off power for performance.

In conclusion, we have developed a simulation methodology and built a simulator using that methodology that has more desirable properties than are normally found in a single simulator. A version that boots Windows XP and Linux is running today. Though implementing the first version of such a simulator has been more difficult than a more conventional software simulator, we are achieving performance that exceeds any similarly-accurate simulator that we are aware of. In addition, most of the implementation difficulties, from incorporating tracing and roll back capabilities into a full-system simulator to reducing the number of host cycles consumed by timing model Modules, only need to be solved once and can then be easily reused in future designs. We are developing tools, such as the microcode compiler, to further reduce the effort to build and modify such simulators. We expect that FAST simulators, with their combination of speed, accuracy and observability, will provide much insight into the inner workings of computer systems.

7. Acknowledgments

This work was partially supported by a Department of Energy Early Career Principal Investigator Award (ER25686), the National Science Foundation (0615352, 0541416), 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 like to thank Joel Emer of Intel for helpful discussions and assistance in integrating FAST into AWB, Paul Hartke of Xilinx for his help with everything FPGAs and Michael Monkang Chu of DRC Computer for his diligent support of the DRC product.

References

[1] H. Angepat, D. Sunwoo, and D. Chiou. RAMP-White: An FPGA-Based Coherent Shared Memory Parallel Computer Emulator. In *8th Annual Austin CAS Conference*, Mar. 2007.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] R. Bhargava, L. Barnes, and B. Sander. AMD. personal email communication.

[5] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commerical Workloads (CAECW)*, Feb. 2003.

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

[7] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: A Full System Simulator for the PowerPC Architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[8] D. 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*, Feb. 2005.

[9] D. Chiou, H. Sanjeliwala, D. Sunwoo, J. Z. Xu, and N. 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*, Feb. 2006.

[10] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu. The FAST Methodology for High-Speed SoC/Computer Simulation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 2007.

[11] N. Dave, M. Pellauer, Arvind, and J. Emer. Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-12*, Feb. 2006.

[12] J. Donald and M. Martonosi. An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation. *Computer Architecture Letters*, 5, Aug 2006.

[13] DRC Computer. http://www.drccomputer.com/.

[14] L. Eeckhout, R. H. B. Jr., B. Stougie, K. D. Bosschere, and L. 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.

[15] J. Emer. HASim talk at RAMP Retreat, June 2007.

[16] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[17] J. Emer, C. Beckmann, and M. Pellauer. AWB: The Asim Architect's Workbench. In *Proceedings of MOBS 2007, San Diego, CA*, June 2007.

[18] J. Holt. Freescale. personal email communication, July 2007.

[19] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alamelden, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. submitted to Computer Architecture News.

[20] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM Press.

[21] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3):15–25, 1999.

[22] E. Nurvitadhi and J. Hoe. Full-System Architectural Exploration Sandbox. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, Feb. 2005.

[23] D. Patterson, Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, , M. Oskin, J. Rabaey, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors. In *Proceedings of Hot Chips 18, Palo Alto, CA*, Aug. 2006.

[24] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors. Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors. In *12th International Symposium on High-Performance Computer Architecture*, pages 27–38, Feb 2006.

[25] S. Ravet. ARM. personal email communication, Sept. 2007.

[26] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7(1):78–103, 1997.

[27] L. Schaelicke and M. Parker. ML-RSIM Reference Manual. Technical report, Department of Computer Science and Engineering, Notre Dame, 2002.

[28] E. Schnarr and J. 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, Oct. 1998.

[29] T. Sherwood, E. Perelman, G. Hamerly, and B. 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.

[30] T. Suh, H.-H. S. Lee, S.-L. Lu, and J. Shen. Initial Observations of Hardware/Software Co-Simulation using FPGA in Architectural Research. In *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-12*, Feb. 2006.

[31] M. Vachharajani, N. Vachharajani, and D. 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.

[32] T. F. Wenish, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer Architecture Simulation. *IEEE Micro*, 26(4):18–31, July/August 2006.

[33] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, 2003.

[34] M. T. Yourst. PTLSim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of ISPASS*, Jan. 2007.

[35] L. Zhang. IBM. personal email communication, Aug. 2007.