Preprint: To Appear in ICCAD 2007.

The FAST Methodology for High-Speed SoC/Computer Simulation

Derek Chiou[1], Dam Sunwoo[1], Joonsoo Kim[1], Nikhil Patil[1], William H. Reinhart[1], D. Eric Johnson[1] and Zheng Xu[1,2]
[1]The University of Texas at Austin, [2]Freescale Semiconductor, Inc.
{derek,sunwoo,turo,npatil,wreinhar,dejohnso,zxu}@ece.utexas.edu

Abstract— **This paper describes the FAST methodology that enables a single FPGA to accelerate the performance of cycle-accurate computer system simulators modeling modern, realistic SoCs, embedded systems and standard desktop/laptop/server computer systems. The methodology partitions a simulator into (i) a *functional model* that simulates the functionality of the computer system and (ii) a *predictive model* that predicts performance and other metrics. The partitioning is crafted to map most of the parallel work onto the hardware-based predictive model, eliminating much of the complexity and difficulty of simulating parallel constructs on a sequential platform.**

FAST conventions and libraries have been designed to make creating, modifying, using and measuring such simulators straightforward. We describe a prototype FAST system: a full-system, RTL-level cycle-accurate-capable computer system simulator that executes the x86 ISA, boots unmodified Linux and executes unmodified x86 applications. The prototype runs two to three orders of magnitude faster than RTL-level cycle-accurate x86 software-based simulators and about six to seven times faster than RTL simulation.

I. Introduction

Predicting behavior of an engineered system is an essential component of engineering. Traditionally, behavior prediction has been performed in a variety of ways ranging from physical modeling to mathematics. Today, running a simulator on a computer is one of the most widely used methods to predict behavior. Computer simulation has been applied to virtually all scientific and engineering fields.

Though one could imagine a day when simulators are considered fast enough, that day appears to be a long way off. Over time, however, most simulators have steadily improved in performance due to host[1] performance improvements and efficiency improvements in the simulators themselves. This paper focuses on improving the host performance by leveraging a specific partitioning to parallelize the simulator. Thus, in this paper we do not consider other simulator efficiency improvements, though such improvements are possible within our methodology as well.

Improved simulation performance requires additional exploited host parallelism that can be gotten from the space domain (multiple processors and/or other host hardware), the time domain (faster clocked processors and/or other host hardware) or a combination of the two. To date, parallelizing simulators of tightly coupled targets across multiple processors has been difficult[1], [2]. We are not aware of any highly scalable solution. Thus, most host performance improvements for such simulators come from the steady (until recently) performance improvements in uniprocessor computers.

Simulating a next generation computer is, however, one area where faster host computers do not significantly improve simulation performance. The reason is simple; the complexity growth in computers is generally faster than uniprocessor performance improvements. Thus, simulating a computer system on a uniprocessor system will generally not improve in performance if you are always simulating next generation computers; in fact, it is likely to get slower over time unless there are simulator efficiency improvements. Since parallelism is essential to improve speeds of simulating computer systems, and success has been limited when trying to map simulators onto multiprocessors, we believe that hardware is the only available solution to dramatically improve simulation performance.

[1]We use the term *target* to refer to the device being simulated and *host* to refer to the device that executes the simulation.

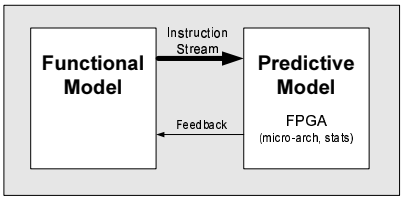

Fig. 1. A High-Level View of a FAST simulator

A general hardware simulation platform must be reconfigurable to accomodate multiple targets. Field programmable gate arrays (FPGAs) are one such reconfigurable hardware platform and the hardware platform we assume in this paper. FPGAs today generally include large amounts of dual-ported memory as well. The largest support 300K+ logic cells and 200K+ registers along with 10Mb of dual-ported BlockRAM. They are, of course, very parallel. Since FPGAs are made from the same silicon as processors, they grow at the same rate and thus could potentially maintain a high simulator performance.

This paper describes a methodology that enables SoC/computer system simulation to be efficiently accelerated using an FPGA. We first describe our approach and compare it to other current approaches to FPGA-acceleration of simulators. We then describe in detail how our approach works. Afterwards, we describe an example of such a simulator and results from that simulator. We then compare the simulator to software simulators and conclude.

II. The FAST Methodology

Many current simulators either run slowly or are highly optimized for performance. Performance optimizations often increase complexity, making simulators hard to verify. Ideally, simulators could be written in a way that is easy to understand and verify, yet provides high performance. As we argue in the introduction, we believe that hardware is needed for maximum simulation performance. Thus, the goal of this paper is to develop a methodology that enables fast and resource-efficient simulators to be easily created and modified.

The FAST[3], [4] simulation methodology attacks the speed/complexity issue by first partitioning the simulator into (i) a *functional model* that simulates the functionality of the target system including the instructions being executed on the processor or processors in the system and peripherals but not the timing and (ii) a *predictive model* that simulates the microarchitectural structures that affect performance but not the functionality of the target machine.

Figure 1 is a high-level view of a FAST simulator. The functional model pipes an instruction stream (we use "instruction" to mean instructions that a processor would execute as well as memory reads and writes and other operations from peripherals that implement and affect system functionality) to the FPGA-based predictive model.

The predictive model only models the microarchitectural structures that affect the behavior that is being predicted. To predict performance, arbitration, queuing and associative memories are some of the structures that should be modeled. Many structures can be dramatically simplified or even eliminated from the predictive model with no loss of accuracy. For example, ALUs are pipeline stages and do no real computation, caches only store tags and not data and so on. Our predictive models are quickly assembled from simple modules, such as FIFOs, associative memories and arbiters, that provide almost all of the models needed for most computer systems. Because of its simple and modular nature, a predictive model is easy to create, modify and extend.

This partitioning has significant benefits. The simulator is far more modular, simpler and more reusable than an integrated simulator. The functional model does not need to know about the microarchitecture of the target system. Likewise, the predictive model does not need to know the functionality of the target. Thus, if the functionality of a system remains the same, but it is re-architected to improve on some metric (performance, power, etc.) a previous functional model can be used as is and only the predictive model needs to change.

The partitioning alone does not improve simulation performance over a monolithic design when run on a sequential host, since what needs to be simulated remains fundamentally the same. An important observation, however, is that the predictive model consumes the vast majority of the computation cycles since modern targets are inherently very parallel while the functional model is inherently sequential. Thus, implementing at least the predictive model on an FPGA is a logical approach to improving performance. The issue then becomes one of tolerating the round-trip latency between the functional model and the predictive model.

Because the predictive model is implemented in hardware, we take a very simple approach to modeling timing. Rather than using an event queue to eliminate computation when it is unnecessary, our current predictive models advance every component every target cycle. Even when moving a bubble through one pipeline register to another, some work is performed. Implementing a software-based simulator in such a way would ensure low performance, but because we are using parallel hardware, such cycle-by-cycle evaluation can be done very quickly since it is done in parallel. In addition, the partitioning eliminates most wide datapaths and thus makes such an approach tractable in a single FPGA.

A. Existing FPGA simulators/emulators

Several companies such as Cadence (Quickturn/Palladium), Axis, Mentor/IKOS/VirtualWires and Tharas sell FPGA-based accelerators or emulators that map RTL to multiple FPGAs to build a cycle-accurate simulator. Such systems can be fast (up to 100M cycles per second according to their respective web sites) and accurate and provide the ability to take performance measurements with little degradation in performance. Such systems, however, often run in the millions to tens of millions of dollars.

The FAST system is very different than these approaches. Because FAST separates the functional model from the predictive model, it is able to model very complex systems with very few FPGA resources. For example, we can model an Intel Pentium M-like system, with a 2MB cache in a *single* Virtex 4 FPGA. A Quickturn-like approach would likely require hundreds of FPGAs. A FAST predictive model does not need to implement the functionality, thus saving a tremendous amount of FPGA resources. In addition, a FAST predictive model can be easily constructed from simple, pre-existing modules with a small amount of custom code to specify specific arbitration policies. The existing FPGA approaches require a complete and running version of the RTL making such boxes inappropriate for early architectural studies, when such simulation might be very helpful.

There are several projects that implement some components of a computer system, such as the processor or memory controller[5], [6] or both[7], [8], in FPGAs and the rest in real hardware. RAMP[9] reference systems are entirely implemented in FPGAs. Such systems are fast but are not accurate unless all components are accurately implemented in FPGAs which would require a tremendous number of FPGAs (as with the Quickturn solution) if the simulated machine is complex. All other prior hybrid FPGA/software simulators[10] that we are aware of are partitioned on target system module boundaries such as a cache, a floating point unit or even a processor core.

B. Functional/Predictive Consistency

On its own, the functional model produces the *right path* (which we also call the *functional path*) instruction stream while the predictive model needs a *target path* (which we also call the *correct path*) instruction stream. There are two primary reasons for the functional path to be different than the target path: branch prediction and parallel memory accesses.

Our general solution is to make the functional model speculative[**?**]. The functional model generates what it expects the instruction stream to be (the right path) and then modifies that instruction stream based on feedback from the predictive model (when the target path is different than the right path). Thus, the functional model has the ability to rollback to a previously executed instruction and then change its control flow.

1) Branch Prediction: When a branch-predicted processor mis-speculates a branch, it executes *wrong path* instructions until the mis-speculation is discovered and corrected. The functional model, however, naturally executes only right path instructions and thus will not normally produce wrong path instructions. To get the target path instructions, the predictive model issues commands to the functional model whenever the functional path is not equal to the target path.

To determine when the functional path is different than the target path, the predictive model models the branch predictor. It notifies the functional model when a branch has been mispredicted so that the functional model can then issue instructions from the wrong path to drive the predictive model. Nested mispredicted branches are possible. Once the predictive model resolves a mispredicted branch, it instructs the functional model to return to the right path. Each notification requires communication from the predictive model to the functional model. That path introduces a critical round-trip loop that directly impacts performance. The shorter the latency from the predictive model indicating a branch misprediction/resolution to the functional model rolling back, the better the simulator performance. The round-trip time is dependent on the size of the messages and the latency and bandwidth of the communication channel.

However, because branches are generally predicted correctly, the cost of the round-trip communication can be amortized over a large number of instructions. Interestingly enough, the better the target branch predictor, the faster the FAST simulator modeling that system. More complex processors tend to have better branch predictors

since their branch misprediction penalties tend to be higher. Thus, as long as there are sufficient hardware resources, more complex microarchitectures will tend to simulate *faster* than simpler ones.

2) Read/Write Misordering Caused by Parallel Targets: Virtually all computer systems have multiple components, such as a processor and an Ethernet card, that can read and write memory. In such cases, the functional model may execute a read and a write to the same location in a different order than the predictive model predicts.

One fact that dramatically reduces the complexity of the problem is that the predictive model back-pressures its corresponding functional model to keep all of the functional components accessing the same resources at almost exactly the right time relative to the other functional components; therefore, memory operations are almost always executed in the correct relative order. In such a system, it is impossible for a reader to read a value that should have been written some n cycles before but has not yet been written because of timing skew. Thus, we only need to worry about the ordering of memory operations that are happening within a time window of n cycles.

There are two problems to solve: detection and correction. There are many solutions to the problems. The solution we present here addresses both of these problems. It assumes that the functional model is running on a sequential host. In that case, we have a single, sequential stream of instructions representing all of the functional activity of the target system, including both processor and peripheral activity, that clearly defines the ordering of memory accesses.

Each functional read is tagged with the functional write that produced the data it read. Each functional write is also tagged with its functional write order. In our planned implementation, that tagging is done by FPGA hardware as the instruction stream is fed into the predictive model. Likewise, the predictive model does the same thing, associating reads with the writes that produce their data. The predictive model confirms that the functional reads/writes were performed in the correct order. If not, the functional models are rolled back and executed in the predictive tag order until the mis-ordered instruction. In general, all functional components that depend on the mis-read data would have to be re-executed as well. As the system re-executes, the addresses of loads and stores waiting within the load/store queue in the predictive model might need to be updated if those addresses depended on the mis-read data.

Note that the predictive model does not need to roll back because control flow is determined by branch prediction and thus does not depend on the actual values of data read and because memory addresses that depend on mis-read data will not have been issued to the cache because of the dependency on the mis-read data. The pending memory addresses that depend on the mis-read data must have their addresses updated by the functional path trace as it is regenerated based on the correct memory ordering.

C. Making FPGA-based Predictive Models Easy and Economical

Writing reasonable, synthesizable RTL has traditionally been harder than writing software. There are many reasons for this difference in difficulty. Hardware is traditionally optimized for performance while software is often not. Meeting timing in microprocessor design requires much more time than first pass functionality. Synthesizable RTL requires knowledge of the underlying hardware to best use and manage the limited resources while software runs on top of an operating system and hardware that present a clean, abstract view of the underlying system.

Certain design decisions can greatly alleviate and even eliminate these differences between RTL development and software development. The first is accepting multiple host cycles to implement a single target cycle, resulting in lower target performance than the native FPGA clock frequency. Cleanly designed logic runs well over 100MHz in modern FPGAs[2] If one is willing to take ten host cycles for every target cycle, for example, the predictive model will run at 10MHz-20MHz, still much faster than virtually any other cycle-accurate simulator of complex systems. The functional model is generally the bottleneck anyways, making the predictive model performance moot.

In addition, multiple host cycles per target cycle allow us to trade hardware resources for performance, making FPGA implementation much more tractable and easier to express. Take, for example, a 16-way associative cache. A standard implementation would require 8 BlockRAMs and 16 comparators along with an appropriate tree to summarize the results and additional structures to perform updates. Using multiple host cycles per target cycle, that cache could be implemented in a single memory and a pair of comparators by reading two values from a single memory (Xilinx and Altera BlockRAMs are dual-ported) and comparing them to the desired tag every cycle for eight cycles. We have traded a factor of 8 in resources for a factor of 8 in time. If the value is found early, the multi-cycle approach can exit early and thus increase its average bandwidth.

Thus, if one is willing to sacrifice some performance, hardware development can be made much easier and less resource-expensive. A factor of two in cycle time is a giant factor for a shipping product, but generally does not matter in a FAST simulator. Predictive models do not need to meet hard timing constraints.

III. FAST Modules and Connectors

One way to write the predictive model is to deconstruct it into two classes of objects: modules and uni-directional connectors. Modules are only attached to connectors while connectors are only attached to modules.

In the FAST methodology, modules model behavior and not time while connectors model time and not behavior. Passing through a module incurs no target time; all time is counted by the connectors attached to the module. We describe connectors and modules in more detail in the rest of the section.

A. Connectors

At the core of the connector is a FIFO buffer. In addition to transferring data in a FIFO manner, FAST connectors (i) delay data and back-pressure information by zero or more target-cycles, (ii) use a local synchronization scheme between the producer and consumer modules, and (iii) include the ability to profile and record trace information.

Connectors are universal and parameterizable. They are used to connect all modules within a predictive model and are customized at each instantiation using parameters. The input side is connected to a producer module and the output side is connected to a consumer module. The producer module enqueues data items of a configurable type into the input side. The consumer module dequeues data items of the same type from the output side.

In addition to the data type being passed, connectors are configured for a fixed minimum delay D, an input side throughput T_i, an output side throughput T_o, and a limit on the number of in-flight transactions $TransLimit$. The delay specifies the minimum number of target cycles between the enqueuing and subsequent dequeuing of a data item. Throughput indicates the maximum number of items that can be enqueued/dequeued in a single target cycle. The transaction limit

[2]One of the authors' professional experience indicates that well designed RTL code easily synthesizes to over 200MHz in Xilinx Virtex 4 parts.

specifies the overall capacity, or depth, of the connector. Connectors used for collecting statistical information or tracing data also accept as instantiation arguments the modules to conduct the statistical compilation or data tracing.

The interface of the connector is very similar to that of a FIFO interface. Standard FIFO interface methods, *Enq*, *Deq*, *First*, are mirrored in the connector but are augmented with methods necessary to maintain timing synchronization, *pDone*, *pCommit*, *getPtime*, *cDone*, *cCommit*, *getCtime* as well as those for managing statistics and data tracing, *rtnStats*, *activateTrace*, *triggerDump*.

1) Buffer Design: The connector buffer is implemented in the FPGA using either BlockRAM or Distributed RAM. For a given cycle, the producer module writes to the buffer assuming that there is no back-pressure. If the buffer is full, then the enqueue method will be blocked and the producer will not be able to progress. More advanced back-pressure (i.e. the required receipt of a token from another module), is implemented using a separate, dedicated connector for the back-pressure and internal producer module logic to control processing and progress.

2) Timing and Delay: The connector maintains local time counters and synchronizes the data movement from producer to consumer modules locally without communicating or synchronizing with other components in the system. Therefore, it is possible for different logical times to exist throughout the model.

a) Time: To delineate time, the connector logically maintains two target cycle counters, one for producer time, *pTime*, and one for consumer time, *cTime*. Although the processing of data may require multiple host cycles, target cycle time advances only when a module commits to the operations. The *pTime* counter is incremented upon receipt of a *pCommit* from the producer and *cTime* is incremented upon receipt of a *cCommit*. Although in a closed system the model is governed by the slowest module, the decoupling of *pTime* and *cTime* within each connector enables modules, as well as the model as a whole, to advance as soon as data is ready with no external synchronization.

b) Delay: The connector maintains a register to record the number of items that have been resident in the buffer for D delay cycles and blocks consumer access, *Deq* and *First*, when that number is zero. The register is decremented by the consumer's *Deq* and incremented by use of a small assist-FIFO, of depth D, that records the number of items enqueued each target cycle. After an initial $D-1$ cycle delay, the assist-FIFO is read once each target cycle to obtain the number of additional items available for dequeue for the next target cycle.

Connectors can be configured to model a zero target time delay. What is actually modeled is a register that is bypassed in the same target cycle if the downstream module can accept the output of the upstream module on the same target cycle. Thus, a module1-connector-module2 model with a zero delay connector is essentially a single module1-module2 device with a zero target time delay when module2 is not back-pressured.

Optionally, timestamp information can be stored in the buffer along with the data and delay can be accomplished by comparing *cTime* with the stored timestamp. Although this option is more straightforward than using the assist-FIFO, it requires increased storage capacity for every entry in the buffer.

3) Trace Recording and Statistics Gathering: Connectors can be configured to trace the data by accepting a trace module as an argument at instantiation time. Configuration of the tracing module includes (i) the specification of trace criteria, i.e., which entries to capture, (ii) the specification of trace packing, i.e., which items of each entry to store, (iii) the specification of the storage location and (iv) and whether to automatically dump the contents of the trace buffer when it becomes full. The connector interface includes a *triggerDump* method which enables pre-event trace capture. For example, by tracing data as it is dequeued into a consumer module and triggering a trace dump if the module stalls (i.e. if the module is unable to process the available data), we can capture the sequence of instructions that led to the stalling condition.

Similar to tracing, connectors through which we desire to capture statistics can accept as an argument at instantiation a configured statistics module. The statistics module is configured with a statistics function and returns its compiled count upon use of the connector method *rtnStats*.

4) Operation: The following section describes the interaction between the connector, producer, and consumer modules. Of particular interest is the assertion and communication of the *Done* and *Commit* signals between the connector and the modules in order to properly delineate target cycle boundaries.

Because the connector is the device responsible for tracking actions in time, it generates the *pDone* signal if the number of enqueued items equals the configured T_i. Similarly for the consumer end, it generates the *cDone* signal if either the number of dequeued items equals the configured T_o or if all the available items for the target cycle have been consumed. Additionally, either the producer and consumer can indicate to the connector when, due to internal module processes, a target cycle has completed by asserting their respective *Commit* signal.

Because connectors maintain local times and do not synchronize externally, modules must *Commit* to every one of their connectors each target cycle. This ensures that the target cycle times associated with the ends of the connectors attached to the module will be the same. Failure to do so can result in an inconsistent timing state with a single module existing in numerous logical times (as maintained in its attached connectors).

5) Connector Performance: Connectors consume one host cycle for internal calculations, determining *Enq* and *Deq* numbers for the next cycle, as well as for the final commit processing, either *pCommit* or *cCommit*. Additional host cycles required depend upon the configuration of the connector and resulting ability to conduct operations concurrent in a single host cycle. Table I displays the performance for several connector configurations for a producer and consumer with no back-pressure (i.e. the modules are always ready to produce and consume). With the 0-delay configurations, because *Deq*'s are dependent on *Enq*'s, the initial *Enq* consumes a host cycle alone while subsequent *Enq*'s can be conducted concurrently with the *Deq* of previous data. In general, 0-delay connectors will require $2 +$ 1 + the connector throughput host cycles to simulate. With the non-zero delay configurations, all *Deq* and *Enq* operations can be conducted concurrently resulting in $2 +$ the connector throughput

TABLE I
STEADY STATE PERFORMANCE OF THE CONNECTOR

Configuration (T_i, T_o, D)	host cycles per target cycle
$(1, 1, 0)$	4
$(4, 4, 0)$	7
$(1, 1, 1)$	3
$(4, 4, 1)$	6
$(1, 1, 10)$	3
$(4, 4, 10)$	6

```
module mkCAM#(ConnectSource#(Tuple2#(addr_t, data_t)) insertQ,
              ConnectSource#(data_t) matchQ,
              ConnectSink#(addr_t) outQ,
              addr_t size)
        (CAM#(addr_t, data_t));

    SPMem#(addr_t, data_t) mem <- mkSPMem(size);
    Reg#(addr_t)           addr <- mkReg(0);
    Reg#(State_t)          state <- mkReg(MATCH);

    rule match_incr (state == MATCH);
        if (mem.read(addr) == matchQ.first) begin
            matchQ.deq();
            outQ.enq(addr);
            addr <= 0;
        end
        else if (addr != size - 1)
            addr <= addr + 1;
        else begin
            matchQ.deq();
            addr <= 0;
        end
    endrule
    rule match_done (state == MATCH && (outQ.done || matchQ.done));
        matchQ.commit(True);
        outQ.commit(True);
        state <= INSERT;
    endrule
    rule insert (state == INSERT);
        match { .addr, .data } = insertQ.first;
        mem.write(addr, data);
        insertQ.deq();
    endrule
    rule insert_done (state == INSERT && insertQ.done);
        insertQ.commit(True);
        state <= MATCH;
    endrule
endmodule
```

Fig. 2. A simple CAM Module

host cycles to simulate.

B. Modules

Modules provide the functionality of the predictive model. For example, modules provide such functionality as CAMs, Caches, Branch Predictors, Fetch, Decode, Rename, and so on. Modules are hierarchical and are very configurable. For example, we provide a CAM module that is used by the Cache module that is, itself, used by the Branch Predictor module. Such hierarchy promotes code reuse and thus reduces implementation and verification effort.

Figure 2 shows a CAM module that takes an arbitrary number of insert and match commands. An insert command inserts into a specified location in the CAM, and a match command returns the lowest address in which the specified data word is found. This CAM can be configured to be of any arbitrary size and can have arbitrarily long data-words. It assumes that within a (target) clock cycle, match commands do not conflict with the insert commands.

Algorithmically, for each match command in the match command queue (matchQ), the CAM module iterates over each entry of a single-ported memory (mem), until it finds the first successful match, the address of which it enqueues into the output queue (outQ). It then inserts data words from the insert command queue (insertQ) into mem.

Since the above implementation uses only a single-ported memory, clearly it needs one cycle per memory access. Thus, an insert command takes one cycle, and a match command takes cycles equal to however many comparisons it needs to do; e.g., an unsuccessful match in a CAM of size 32 will take 32 cycles, but if the first entry in the CAM happens to match, it will only take one cycle. In addition to the comparison delay, there is an extra overhead due to the hand-shaking with the connector.

It is much easier to write a module in this fashion because:

1) As mentioned in Section II-C, modules are not as difficult as regular hardware design because they can take multiple host cycles to do the job of one target cycle.
2) The connector abstracts away much of the work in connecting two modules unlike traditional module-connection semantics that revolve around "exporting" a set of ports in the interface, where the instantiating module is expected to connect these ports appropriately.
3) The connector maintains time, thus preventing modules from having to do so to avoid getting out of sync.
4) The CAM does not assume anything about the number of ports into and out of the CAM module, nor does it know about the number of cycles the request will be processed in on the simulated machine. The connector enforces the right throughput at the right latency, by actively telling the module that it is done and providing only the data for the current target cycle.

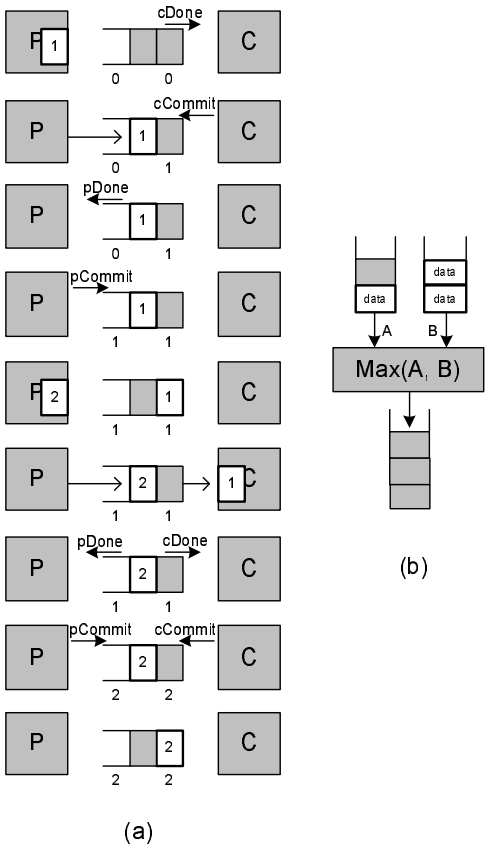

Fig. 3. A simple example of the connector

C. Simple Example

For an example, consider Figure 3(a) that has a Producer module connected to a Consumer module via a connector. Throughputs are configured to be 1 on both the input side of the connector and the output side of the connector and the delay is configured to be 1 cycle.

The producer attempts to enqueue into the connector. If there is space in the connector, the producer is allowed to enqueue. Since throughput is configured to 1, that is the only data item that can be enqueued and thus the connector generates a *pDone* signal. The producer acknowledges with a *pCommit* signal, thus incrementing the input producer target cycle by one in the connector. Similar communication is conducted at the connector output with the consumer.

Figure 3(b) shows an example of a module that has two source connectors (each with a T_o of 2) and one sink connector (with a T_i of 3). The module is a sorting module; it examines the head value of

each of its inputs, dequeues the larger value and forwards it to the output. The module must either see a *cDone* on both of its inputs or a *pDone* on its output before it completes its target cycle and asserts *cCommit* on both of its input connectors and *pCommit* on its output connector advancing the target cycle time for its end of each connector.

The left source connector has one data value enqueued while the right source connector has two data values enqueued. Neither source connector *cDone* signal is asserted. Because there is data on both inputs, the module examines both, determines the left source connector's head value is larger and dequeues it and passes it to the sink connector. After the left source connector is dequeued, it will assert the *cDone* signal since it has no more data for this target clock cycle.

IV. FAST Implementation Status and Performance

Our initial FAST simulator supports the x86 instruction set, boots Linux and runs unmodified application binaries compiled on real machines. Our functional model is QEMU[11], a full-system simulator that we have extensively modified to support instruction trace and rollback. It runs on both standard workstations as well as on the embedded processor (an in-order IBM PowerPC 405 running at 300MHz) within Xilinx Virtex 2 Pro FPGAs. Though there is no reason that the functional model cannot be implemented in hardware, there are significant benefits to starting with a software-based full-system functional model that already runs the x86 ISA and boots Linux and Windows.

We are finishing a processor-core predictive model (Figure 4, modules are blocks and connectors are FIFOs) that simulates a branch-predicted superscalar processor core with reservation stations, multiple functional units, virtual memory and a full memory hierarchy. We will then augment the model with a memory bus, DRAM and disks. Our predictive model is being assembled from composable/configurable modules that include models for CAMs, caches, arbiters, DRAMs, disks, branch predictors, FIFOs, memories and ALUs.

The entire predictive model is written in Bluespec[12]. Bluespec's ability to pass types permits us to make the module and connector interfaces very general. Each module has a defined type that is extendible but also contains additional "pass-through" and "pass-back" type fields that permit the user to provide additional information without having to modify the module. The pass-through type provides information that the consumer module or a future consumer module might need. The pass-back type provides information that would be passed back to the producer module when a reply is returned to the producer module.

Since each connector can be individually configured with T_i, T_o and delay and since each module is highly configurable, even starting from a pre-created predictive model, one can study a wide range of microarchitectures. For example, through the connectors attached to the ALU, one can set the latency of the ALU. One can vary the cache-sizes. The throughput settings on the connectors indicate how many instructions can be produced and consumed per target cycle for each connector. Thus, by making the appropriate throughput settings, one can generate an n-way superscalar processor. Throughputs do not have to be the same on both sides of a connector. For example, one could allow the Fetch to enqueue a maximum of four instructions per cycle, but only allow the Decode to dequeue a maximum of two instructions per cycle.

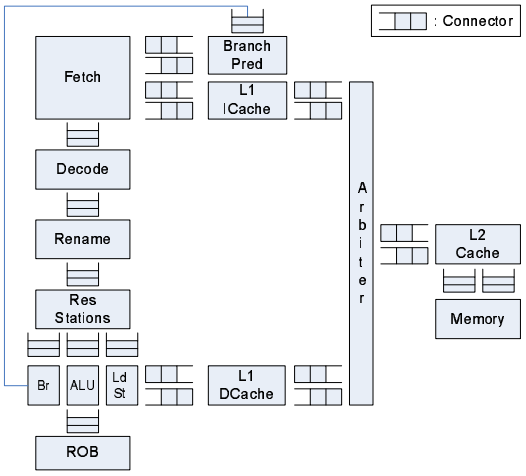

Fig. 4. A Predictive Model of a Superscalar processor

A. Development Platforms

Our first development platform was a stand-alone Xilinx board with one Virtex 2 Pro 30 FPGA. Depending on the number of connectors implemented in BlockRAMs, a predictive model for a current general-purpose out-of-order processor with a 2MB cache can fit in that FPGA which can also be found on a $600 Xilinx XUP development board. It is very likely that multiple predictive models could fit in a single modern FPGA.

We have also ported our implementation to a DRC Computer prototyping system[13] that consists of an FPGA module, in our case a Xilinx Virtex 4 LX60, that plugs into one HyperTransport socket in a dual-socket AMD Opteron system. The other socket is populated with a standard Opteron 275. This system will eventually provide very low latencies between the Opteron and the FPGA (but does not currently, impacting performance).

FPGA utilization results using the DRC platform are presented in Table II. It is interesting to note that very low area penalty is incurred for the predictive model when the issue width of the target processor is increased. This is due to the fact that the critical connectors are built from BlockRAMs. When the connector size is increased, the BlockRAM utilization for each connector only increases when the memory requirement exceeds what is currently available. If the increased connector size does not require more memory than is available in the currently allocated BlockRAMs, then no additional resources are required. Additionally, the connector enables time multiplexing of modules, meaning that architectural changes that would typically result in larger modules only results in larger connectors.

B. Current Performance

Figure 5 shows the FAST simulator performance on the DRC platform booting unmodified Linux and running SPECINT2000 benchmarks[14] on top of the operating system. The speed of the simulator is over 1.2 million instructions per second (MIPS) on average and over 3 MIPS for some benchmarks.

Performance suffers for a variety of reasons. Perhaps the most important and easiest to fix is the fact that the DRC platform latency between the Opteron and the FPGA is quite long, about a factor of four longer than it should be. In addition, by using coherent cache accesses (not currently supported), average latencies will drop to approximately a cache hit.

TABLE II
UTILIZATION RESULTS FOR VARIOUS ISSUE-WIDTH FAST PREDICTIVE MODELS IN A XILINX XC4VLX60

1 issue		
slice count	8,135 / 26,624	30%
BlockRAMs	117 / 160	73%
4 issue		
slice count	8,077 / 26,624	30%
BlockRAMs	121 / 160	75%
8 issue		
slice count	8,393 / 26,624	30%
BlockRAMs	121 / 160	75%

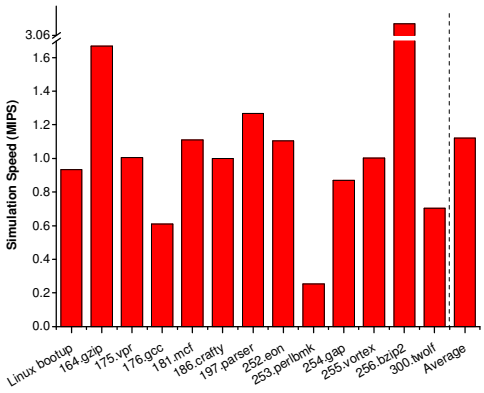

Fig. 5. Simulation Performance of FAST Simulators on a DRC Computer prototyping system

Since this is our initial prototype, both the functional model and predictive model need to be tuned for performance. We do know that there is a tremendous amount of debug code in our current implementation of the functional model and that the predictive model can also be extensively optimized.

Combining the performance benefits of both an improved interface between the processor and the FPGA with some simple optimizations, we have reason to believe that the simulator will be significantly faster than today, perhaps by an order of magnitude or more.

V. RELATED WORK

The functional/predictive partition itself is not novel. FastSim[15], Asim[16] and M5[17] are some of the software-only cycle-accurate or near cycle-accurate simulators that are partitioned in this fashion. Only FastSim, however, uses a speculative functional model that rolls back to handle mis-speculation. It uses binary instrumentation rather than simulation, however, requiring that the host processor supports the same ISA as the target processor and greatly complicates simulating OS instructions. We cannot directly compare against those simulators, however, since FastSim and M5 only support the Alpha ISA and Asim is not available outside of Intel.

Asim is the only functional/predictive partitoned simulator that runs the x86 ISA and is trusted within Intel as being truly cycle-acccurate. It has been our main comparison because it is the basic simulator used through much of Intel. In cycle-accurate mode, however, it runs speeds between 1KHz and 10KHz[18] and does not boot operating systems. Asim has been in development approximately as long as Simplescalar and has had the benefit of being actively used within DEC/Compaq/Intel to evaluate real products. The speed at which Asim runs as compared to other, less accurate academic simulators, is some evidence that achieving true cycle-accuracy for x86 is significantly more expensive than 5% level aggregate accuracy.

Simplescalar[19] is the most commonly used academic simulator. We ran Simplescalar's `sim-outorder`, the most accurate model from the standard 3.0 distribution, on the same DRC machine that we used to generate the FAST simulated numbers and achieved 500KIPS to 600KIPS. Simplescalar, however, is well known to be inaccurate. Even a calibrated version of Simplescalar, `sim-alpha`[20] is within 6.6% for SPECINT2000 and within 21% for SPECFP2000.

IBM's Mambo has a cycle-accurate model of a Power4/Power5 processor that runs at about 200KIPS[21] while Freescale has a cycle-accurate PowerPC simulator that runs at approximately 80KHz simulating an E500 target which is an OOO, dual-issue processor with branch prediction[22]. These simulators are close to their peak performance, however, having been highly optimized. They are also executing the Power/PowerPC ISA which, though complicated, is still significantly less complicated than the x86.

PTLSim[23] is the simulator that is closest in performance and functionality to FAST. PTLSim is a software-only cycle-accurate simulator that targets the x86 ISA and can run full system software. The claimed aggregate accuracy is within 5% for the single benchmark that was calibrated and the speed is very high at around 415KHz or within a factor of three of the current version of FAST. However, we believe that PTLSim is already highly optimized while we have not yet started optimizing FAST. In addition, FAST collects statistics in FPGA hardware ensuring almost zero performance impact as long as there are sufficient hardware resources. Also, a single calibration point cannot accurately determine accuracy, since there are often large variations within a program and across different programs.

VaST Systems[24] claim in the tens of MHz cycle-accurate simulation performance but only model simple, in-order cores rather than out-of-order superscalar cores that FAST supports. In addition, their performance suffers as the number of statistics grows.

We believe we are the first to implement the predictive model in an FPGA. A recent Intel project, HASim[25], intends to produce a hardware Asim which is based on a predictive-model-driven functional model. The predictive model tells the functional model when to fetch, decode, rename, execute and retire each instruction. Thus functionality is performed at the appropriate time, eliminating the need for functional model rollback. HASim, however, requires extensive communication between the predictive model and functional model and thus the functional model must be implemented on the FPGA, a difficult task for complex ISAs like x86.

Emer's Asim[16] is earlier work in the software-only domain that provided inspiration for the connector described in this paper. Pellauer and Emer have started to extend that work to the FPGA domain as well.

VI. CONCLUSIONS

We have described the FAST simulation methodology that enables very high speed simulation of computer/SoC/embedded systems by using FPGAs as an accelerator. The methology depends on a partitioning of the simulator into a functional model and a predictive model. We also give details on how a predictive model is constructed using Connectors that model time and Modules that model predictive model behavior but not time. The successive decomposition simplifies the resulting simulator and enables it to be partially implemented in an FPGA, resulting in very high simulation performance. Our initial results show that a predictive model of eight-issue superscalar processor can be modeled in a single Xilinx Virtex 4 LX60.

VII. Acknowledgements

This research was partially funded by a Department of Energy Early Career Principal Investigator Award, the National Science Foundation, a Faculty Award from IBM, grants and equipment donations from Intel, equipment and software donations from Xilinx and a gift from Freescale.

We would like to acknowledge Dr. Joel Emer of Intel and Michael Pellauer of MIT for comments and technical discussions.

References

[1] T. Li, Y. Guo, and S.-K. Li, "Design and Implementation of a parallel Verilog simulator: PVSim," in *Proceedings of the 17th International Conference on VLSI Design*, 2004, pp. 329–334.

[2] 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*, Feb 2006, pp. 27–38.

[3] 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.

[4] 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.

[5] J. D. Davis, L. Hammond, and K. 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*, Feb. 2005. [Online]. Available: http://cag.csail.mit.edu/warfp2005/submissions/33-davis.pdf

[6] S.-L. Lu, E. Nurvitadhi, J. Hong, and S. Larsen, "Memory Subsystem Performance Evaluation with FPGA based Emulators," in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, Feb. 2005. [Online]. Available: http://cag.csail.mit.edu/warfp2005/submissions/16-lu.pdf

[7] C. Kozyrakis and K. Olukotun, "ATLAS: A Scalable Emulator for Transactional Parallel Systems," in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*, Feb. 2005. [Online]. Available: http://cag.csail.mit.edu/warfp2005/submissions/6-kozyrakis.pdf

[8] 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. [Online]. Available: http://cag.csail.mit.edu/warfp2005/submissions/18-nurvitadhi.pdf

[9] 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.

[10] 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. [Online]. Available: http://www.cag.csail.mit.edu/warfp2006/submissions/suh-git.pdf

[11] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[12] "Bluespec webpage," http://www.bluespec.com.

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

[14] "SPEC webpage," http://www.spec.org.

[15] 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*, Oct. 1998, pp. 283–294.

[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*, vol. 35, no. 2, pp. 68–76, 2002.

[17] 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.

[18] J. Emer, "HASim talk at RAMP Retreat, June 2007." [Online]. Available: http://ramp.eecs.berkeley.edu

[19] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[20] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, 2001, pp. 266–277. [Online]. Available: citeseer.ist.psu.edu/article/desikan01measuring.html

[21] L. Zhang, "personal email communication," 2007.

[22] J. Holt, "personal email communication," 2007.

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

[24] "VaST Systems," www.vastsystems.com.

[25] 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. [Online]. Available: http://www.cag.csail.mit.edu/warfp2006/submissions/dave-mit.pdf