Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs

Michael Pellauer[†] Muralidaran Vijayaraghavan[†] Michael Adler[‡] Arvind[†] Joel Emer[†‡]

[†]Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Computation Structures Group
{pellauer, vmurali, arvind}@csail.mit.edu

[‡]Intel Corporation
VSSAD Group
{Michael.Adler, Joel.Emer}@intel.com

Abstract – **In this paper we explore microprocessor performance models implemented on FPGAs. While FPGAs can help with simulation speed, the increased implementation complexity can degrade model development time. We assess whether a simulator split into closely-coupled timing and functional partitions can address this by easing the development of timing models while retaining fine-grained parallelism. We give the semantics of our simulator partitioning, and discuss the architecture of its implementation on an FPGA. We describe how three timing models of vastly different target processors can use the same functional partition, and assess their performance.**

Index Terms – **Simulation, Performance Modeling, FPGAs**

I. INTRODUCTION

Microprocessor architects rely on cycle-accurate *performance models* to make decisions about next-generation systems. These simulators must be available early in the design process so that they can guide major design decisions before the costly step of Register-Transfer Level (RTL) hardware description. In order to be successful, a performance model must meet two criteria:

- Be accurate enough to use as a basis for major design decisions and feasibility studies.
- Total time of modeling (model development time and total model benchmark execution time) must be short enough to keep the architects at the head of the design cycle.

Recently, performance models have begun to fall behind in the area of total model benchmark execution time. Industry models with high levels of detail typically report simulation speeds in the 1 to 100s KIPS (Thousands of Simulated Instructions Per Second) range [4].

Although parallelizing the simulator can help improve performance, architects expect that the increasing popularity of multicore architectures will actually widen the gap between simulator speed and target speed. This is because of a variety of factors. First, simulating four cores is fundamentally four times the work of simulating one core, but running the simulator on a four-core machine does not in practice result in a 4x speedup due to communication overheads. Second, next-generation multicores typically increase the number of cores, so that architects may find themselves simulating six- or eight-core target machines on a four-core host. Third, the on-chip core interconnect network (the so-called *uncore*) grows in complexity as the number of cores increase and it becomes necessary to simulate more-complicated communication topologies. Fourth, the age-old problem of increasing cache sizes of next-generation cores is expected to continue, meaning longer-running benchmarks become necessary to fully exercise the machine.

Some kind of sea-change is necessary if performance models are going to maintain a high-enough level of speed or accuracy to stay relevant. Currently there is interest in the performance modeling community in exploring FPGAs as an execution platform for performance models. Contemporary efforts include Penry et al.'s accelerator for the Liberty simulator [17], Chiou's UT-FAST hardware-software hybrid simulator [3, 4], and HAsim—our ongoing effort to create an FPGA variant of the Asim simulation environment [6, 7, 16]. Performance models are also a target application of the RAMP FPGA platform [1, 9, 20]. The reasoning behind these projects is that performance models have been shown to have a degree of parallelism in the hundreds [19], yet these parallel tasks are typically quite small. The hope is that FPGAs will be better able to exploit this extremely fine-grained level of parallelism.

Although FPGAs can improve simulator execution speed, the process of designing a performance model for an FPGA is more complex than designing a simulator in software. FPGAs are configured with hardware description languages, and are not integrated into most modern debugging environments. There is a danger that performance modeling on FPGAs will fail not because of execution time, but because of increased model development time.

In this paper we explore whether this issue can be addressed through the use of timing-directed simulation. In this scheme the simulator is divided into separate functional and timing partitions which interact in order to form a complete simulation. The goal is that a single functional partition can be created, verified, and optimized for FPGAs, and then reused across several timing models. In contrast with earlier hardware/software hybrid efforts [17, 3, 4] HAsim places both partitions on the FPGA in order to minimize communication latency and take advantage of the closely-coupled simulation scheme presented here.

The main contribution of this paper is to describe an efficient architecture for such a simulator on an FPGA. We give the semantics for our functional partition, and argue that it is sufficient to model any microprocessor pipeline. We demonstrate the creation of three timing models, and use benchmarks to analyze interesting properties of the system.

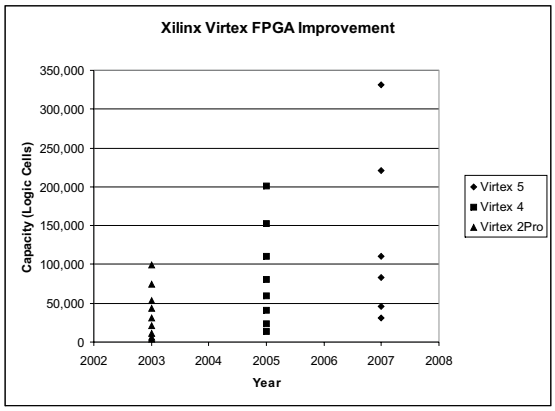


Figure 1: Increase in the number of resources for recent Xilinx FPGAs. Source: [22]. Virtex 2Pro FPGAs were available between capacities of 3,000 and 99,000, whereas the Virtex 5 ranges from 30,000 to 300,000. The Virtex 5 also contains several architectural improvements over previous generations.

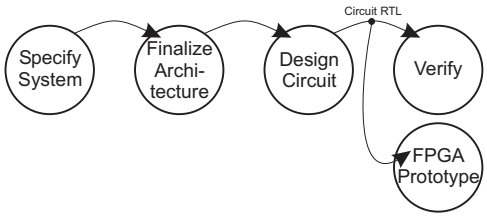


Figure 2: FPGAs are traditionally used late in the design flow as a platform to verify prototypes. The RTL of the target design is used to configure the FPGA,

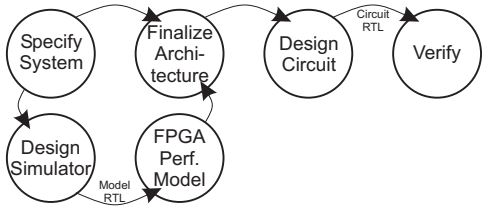


Figure 3: A performance model on an FPGA is constructed early in the design process. The FPGA is configured into a simulator, which may bear little or no resemblance to the final system. This simulator can use FPGA-efficient structures exclusively.

II. PERFORMANCE MODELS ON FPGAS

A. The Promise of FPGAs

Field-Programmable Gate Arrays (FPGAs) are a mature platform for reconfigurable computing. Recent advances in FPGA architectures have greatly increased the attractiveness of the platform. Figure 1 shows the increase in capacity for the last three generations of Xilinx FPGAs. In academia, many active research projects give hope that FPGAs will continue to improve in the immediate future [8].

Yet despite these advances in capacity, FPGA clock rates typically range between 50 and 300 MHZ. How can a performance model running on an FPGA clocked at these speeds beat a software simulator running on 3GHz multicore host processors? The answer is fine-grained parallelism. Results from several technology generations ago established the typical amount of concurrency available in distributed logic simulation to be in the hundreds [19], and we expect that this number has increased since this research was published. This level of parallelism overwhelms today's multicore computers. Additionally, each of these parallel tasks is quite small, representing only the simulation of a few gates: much smaller than the general-purpose host they run on. Finally, there is a high degree of communication between these parallel tasks as results propagate throughout the system. For software simulators running on multicore hosts these communications are sequentialized through the bottleneck of main memory. Given these properties FPGAs, which feature numerous small lookup tables connected by a fast interconnect, should be a better platform for exploiting this fine-grained level of parallelism.

Traditionally, FPGAs have enjoyed success in the industrial digital design flow as a platform to verify circuit *prototypes*. As shown in Figure 2, FPGAs enter the picture late in the design flow, after the complete or near-complete RTL for a circuit is available. This RTL is used to configure the FPGA directly. As a consequence, if the circuit contains any structures that do not synthesize into a small number of FPGA lookup tables, then the resulting FPGA configuration will likely take a large number of resources and not achieve a good clock rate. Regardless, the result (or *waveform*) observed from the FPGA will still be cycle-accurate, albeit running at a slower clock rate.

Early efforts at performance modeling on FPGAs such as those by Ray [18] and Wunderlich [21] distinguished themselves from prototyping more in intent than technique. Their goal was to make a model much earlier in the design process where it could guide architectural decisions, however their approach was more like prototyping in that the FPGA clock-rate could be slow if the target circuit was not amenable to lookup-table configuration.

More recent efforts have taken a different approach: configuring the FPGA into a *simulator* rather than a prototype, as shown in Figure 3. By this we mean that the RTL used to configure the FPGA is not a description of the target circuit. Instead it is a description of the timing and behavior of the target circuit. This simulator may take any number of FPGA clock cycles to simulate one *model clock cycle*. Hence the result of simulation may no longer be obtained by directly observing the FPGA waveform, but must be reconstructed in some simulator-specific way [16].

The advantage of this approach is that the simulator RTL can make use of FPGA-optimized structures exclusively, and thus can be expected to use a small number of resources and achieve a good clock rate. The disadvantage of this approach is that this clock rate by itself no longer gives an indication of simulation rate: if the simulator takes too many FPGA clock cycles to compute a model cycle, then simulator performance will suffer. Another potential criticism of this approach is that the simulator RTL is not useful when the design team must actually implement the target circuit. However we do not consider this to be a major disadvantage as the source code for contemporary performance models written in C is similarly useless.

In order to calculate the simulation rate of an FPGA performance model we must take into account the number of FPGA cycles required to simulate a model cycle, known as the FPGA-cycle to Model-cycle Ratio (FMR) [16]. FMR is similar to the microprocessor performance metric Cycles Per Instruction (CPI) in that one can observe the FMR of a run, a region, or

a particular class of instructions in order to gain insight into simulator performance. The FMR of a simulator combined with its FPGA clock rate gives us simulation rate:

$$frequency_{simulator} = \frac{frequency_{fpga}}{FMR_{overall}}$$

Often it is useful to evaluate simulators based on their simulated Instructions Per Second (IPS). For a software simulator this is calculated as:

$$IPS_{simulator} = \frac{frequency_{simulator}}{CPI_{model}}$$

Plugging in our above equation gives us the means to calculate the IPS of an FPGA performance model:

$$IPS_{simulator} = \frac{frequency_{fpga}}{CPI_{model} \times FMR_{overall}}$$

B. Potential Drawbacks of the FPGA Approach

Although FPGAs show great potential for increasing the simulation rate of microprocessor performance models, the largest potential drawback is the increase in model development time. FPGAs are configured using hardware description languages such as VHDL or Verilog. The resulting RTL is passed through industrial synthesis and place-and-route tools which can have execution times in the hours. If the resulting configuration is faulty, it can be difficult to ascertain the cause of the error, particularly if the bug interferes with the FPGA's input-output capabilities.

In software development these problems have been addressed via two main mechanisms: code reuse and abstraction. Common functions are factored into operating systems or libraries, which are verified and trusted. Projects are written in machine-independent higher-level languages such as C and the source code is integrated into future projects via standardized interfacing semantics and calling conventions.

Ultimately, we believe that the problems of FPGA development will be solved via a similar set of solutions. The emergence of high-level synthesis languages such as Bluespec [2] and HandelC [14] help raise the level of abstraction of hardware development. Similarly, ongoing efforts to develop "middleware" for FPGAs [10, 15] will help lower the barrier of entry.

For performance models on FPGAs, the ability to reuse code will become a critical factor in reducing model development time. Rather than relying on ad-hoc code reuse, we propose to use a technique that has worked successfully in software performance models: timing-directed simulation.

III. TIMING-DIRECTED SIMULATION

A. Technique Overview

Performance models which are constructed in an ad-hoc manner often have limited potential for code reuse. One reason for this is the difficulty of separating source code dealing with timing (how long operations take) from functionality (what operations do). In microprocessors, exploring a future generation is often mostly about exploring when things happen (branch predictors, cache strategies, pipeline depths), and only a limited amount of genuinely new functionality.

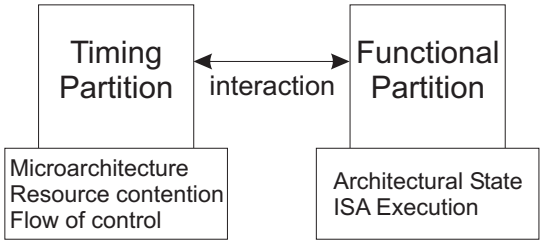


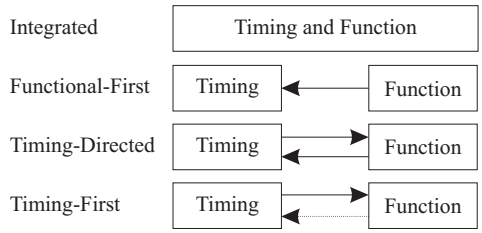
Figure 4: A partitioned simulator divides responsibilities, reusing the same functional partition across many different timing models.

One way to address this is to divide the simulator into *functional* and *timing* partitions, as shown in Figure 4. The functional partition is responsible for correct ISA-level execution. The timing partition (or *timing model*) is responsible for driving the functional partition in such a way as to simulate a particular microarchitecture. Example responsibilities of the functional partition include decoding instructions, updating simulator memory, or guaranteeing that floating point operations conform to standards. Example responsibilities of the timing partition include deciding what instruction to issue next, tracking branch mispredictions, and recording that floating-point multiply instructions take 5 clock cycles to execute.

The goal of this partitioning is to speed development time. The functional partition might be complex to implement, optimize, and verify, but once it is complete it can be reused across many different timing models. The timing models themselves are significantly simpler to implement than simulators written from scratch: they do not need to worry about ISA functional correctness, but only track microarchitecture-specific timing details. Often structures can be excluded from the timing model partially or completely, as their behavior is handled by the functional partition. A common example of this is a timing model of a cache that needs to track tags and status bits but does not need to store the instructions or data – the goal being to decide whether a particular load hits or misses, but not actually track the data associated with it.

Most importantly, a large amount of code reuse is available between timing model generations, as only those portions of the microarchitecture that change from one generation to the next need to be reimplemented. Practice with the Asim simulator environment [7] has shown models can be decomposed in such a way as to reuse branch predictors, cache hierarchies, or communication networks with no changes whatsoever. Given these properties the HAsim project explores implementing a partitioned simulator on an FPGA.

Within partitioned simulation, there are many potential ways for these functional/timing partitions to interact. Mauer, Hill and Wood [13] categorized such simulators as *functional-first* (traditionally called *trace-driven*), *timing-directed*, and *timing-first*, as shown in Figure 5. In the functional-first scheme a functional model is used to generate an execution trace that is fed into a timing model, which adds microprocessor-specific timing information to the trace. A timing-directed simulator, in contrast, is an execution-driven simulator where the timing model invokes operations on the functional model at the right time. In the timing-first style timing is first calculated,



Arrows indicate inter-simulator interactions per simulated instruction.

Figure 5: Mauer, Hill, and Wood's categorization of partitioned simulators based on their interaction. Source: [13]

and then a functional model invoked to verify the results. Contemporary partitioned software simulators include Asim [7] and MASE [11].

Chiou's UT-FAST [3, 4] is a hybrid simulator where the functional model runs in software and produces a trace which is fed into a timing model on an FPGA. In this respect it resembles a trace-driven simulator, however capability is added so that the timing model can roll back the trace-generator if it goes down a different path from that of the simulated machine, ensuring proper simulation results. Given the long communication latencies between the software and the FPGA, the trace-generator uses speculation to continue to execute instructions in the absence of timing model input, such as at a branch in control flow. Capability is added to the trace generator to roll back to a previous point in the execution stream when the timing model detects a mis-speculation, ensuring accuracy of results. We term the UT-FAST approach to be a loosely-coupled timing-directed approach, because it aims to minimize the communication between the two partitions. This has been demonstrated to work well for simulating single-core systems [4], where speculative points occur only at branches. Extending this scheme to simulate multicores, where timing interactions in the cache coherence and the memory model are more common, is ongoing work.

HAsim uses an alternative approach which we term a *tightly-coupled* timing-directed scheme. In this scheme communication between the partitions is frequent, and thus both the functional and timing partitions are placed on the FPGA. The reasoning behind this decision is as follows:

- The fine-grained parallelism of the FPGA can benefit both the timing and functional partitions, which both have high degrees of parallelism.
- Because updates of functional state occur within a few FPGA cycles of the timing partition's requests, the functional partition does not need to speculate on future timing partition requests, which simplifies the architecture. (Note that this is different than *simulating* speculative processors, which we do support and describe below.)
- Rare events which are difficult to place on the FPGA, such as system call instructions, can be farmed out to software regardless of whether they occur in the functional or timing partition, similar to Chung's migration scheme [5].

In the following sections we give the semantics of our particular timing-directed scheme, give an architecture to implement it on an FPGA, and evaluate the efficiency of our scheme.

B. *Semantics of the HAsim Functional Partition*

At a high level, the job of the functional partition is straightforward: given a machine in a certain state and an instruction, calculate the new state of the machine after executing the instruction. This coarse granularity is sufficient for modeling simple in-order pipelines, but is not a high-enough level of detail to capture the behavior of today's microprocessors which include features like out-of-order execution and speculative execution.

In order to be able to precisely capture control speculation, data speculation, and the timings of interactions between threads, we identified seven operations for our functional partition, shown in Figure 6. These operations roughly correspond to stages in a traditional microprocessor pipeline, with additional support for controlling the precise timing of store operations in order to simulate thread communication. A description of the effects of these operations are given in Figure 7.

The order in which the timing partition invokes these operations determines the state of the machine at any given moment. Figure 8 demonstrates how the same functional partition can be three reused across three different timing models to simulate different microarchitectures.

For a single in-flight instruction, the operations are typically invoked in the order they are given in Figure 7 (operations which do not apply may by skipped). This corresponds to instructions flowing through pipeline stages in a real computer: the instruction is fetched (`getInstruction`) before it is decoded (`getDependencies`), takes place before register read (`getOperands`), and so on. The order in which the timing model invokes these operations on separate in-flight instructions determines the state of the machine. We can conceive of a timing model which fetches ten instructions before decoding one, for example.

As the functional partition executes each operation, it changes the architectural state of the simulator, and thus the result of subsequent operations. For example, executing `getResult()` on an instruction which writes register R5 will mean that a subsequent `getOperands()` call will see that value of R5. If an instruction is executed in some way which is not consistent with program order, the `abort()` operation undoes its effects and allows it to be retried. All operations are speculative and may be aborted until the `commit()` operation is called, at which point they become permanent.

The distinction between local writes and global writes allows for accurate control of inter-thread communication. The fact that the timing model uses these operations to control the exact timing, in modeled time, of the visibility of data allows for precise control of the timing of inter-thread communication. This is a key attribute of a closely-coupled functional partition.

Using these constructs a timing partition can accurately model advanced processor features such as out-of-order issue, or speculative execution, as shown in Figure 9. Data speculation can be supported by having the timing model provide results of the operations themselves (telling the functional partition that the result of `getOperands` should be zero, for instance), though this is not yet supported.

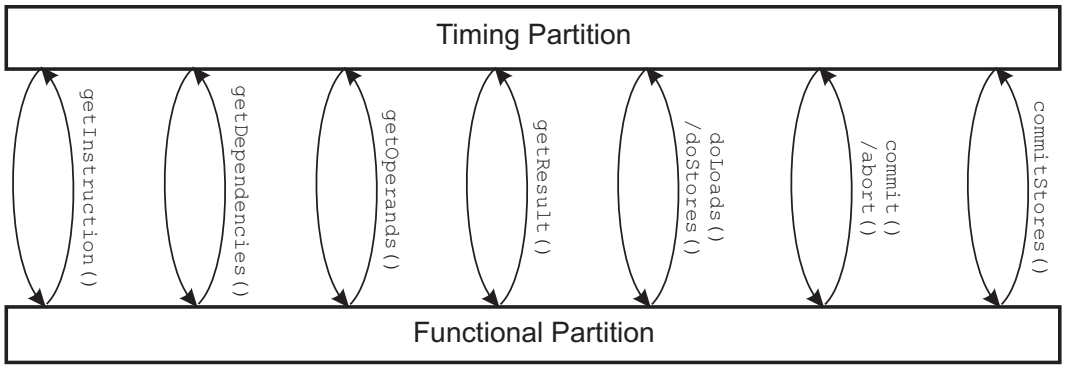


Figure 6: Overview of our timing-driven simulator semantics. The functional partition provides several operations which the timing model uses to produce a cycle-accurate simulation.

Operation	Parameters	Return Value	Effect
getInstruction	Addr	Inst	Fetch the instruction at this address and place it in flight.
getDependencies	Inst	Deps	Get the dependencies of this instruction relative to other in-flight instructions.
getOperands	Inst	Srcs	Read the register file and prepare the instruction for execution.
getResults	Inst	Result	Execute the instruction and return the result, including branch information. For loads and stores, do effective address calculation.
doLoads	Inst	Value	Perform and memory reads associated with the instruction.
doSpeculativeStores	Inst	-	Make any memory writes visible to local loads.
commit	Inst	-	Commit the instruction's local changes and remove it from being in-flight.
abort	Inst	-	Abort the instruction's local changes and remove it from being in-flight.
commitStores	Inst	-	Make any memory writes globally visible.

Figure 7: Summary of functional partition operations.

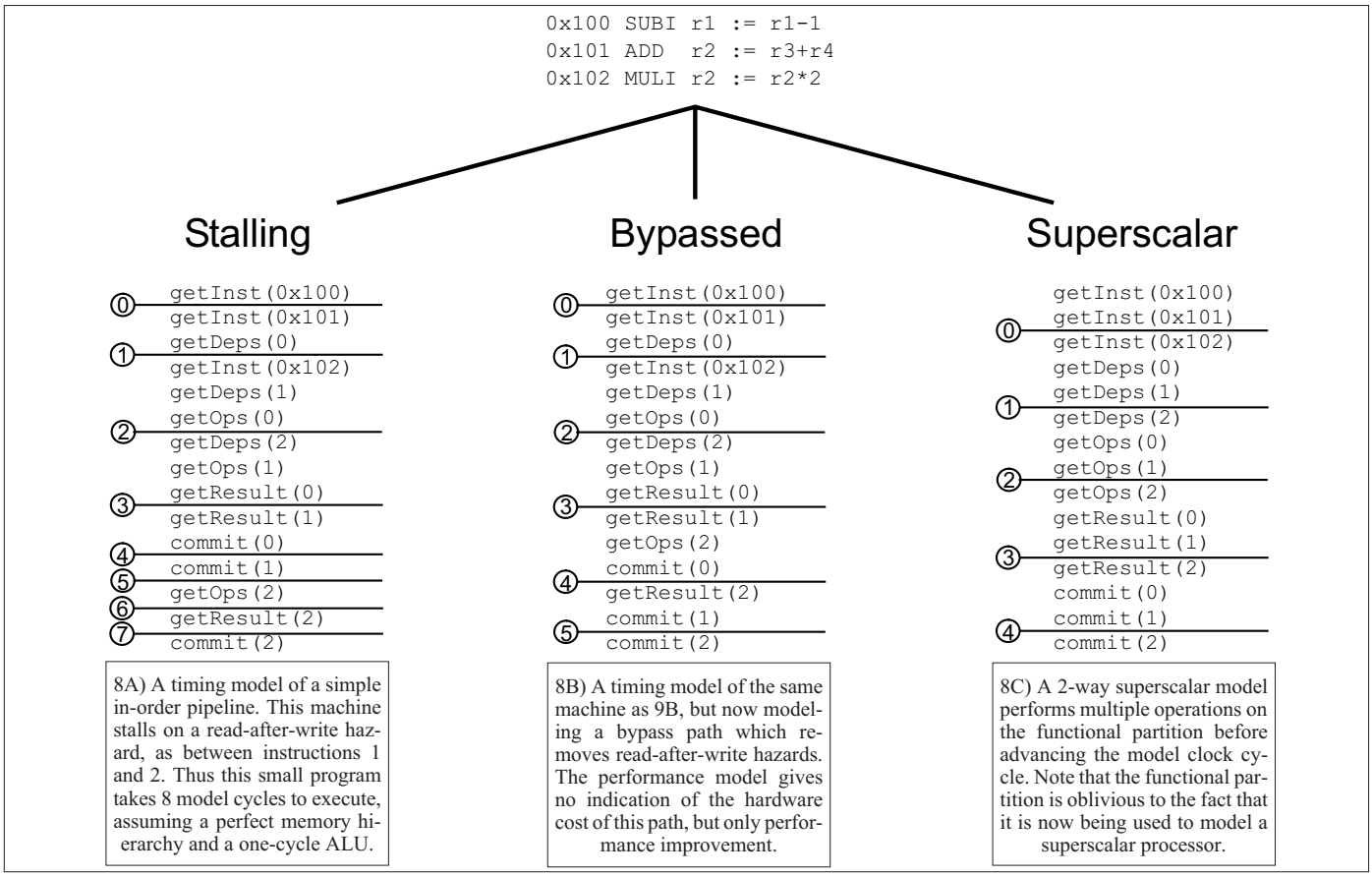


Figure 8: 3 different timing models operating on the same instruction stream. Each simulator must do the same fundamental amount of work. The only change is how this work relates to model time.

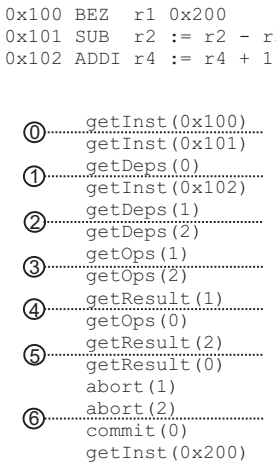


Figure 9: Timing model demonstrating out-of-order issue and speculative execution. Here the branch is stalled on a dependence, and the timing model predicts branch not-taken and issues past it. Upon branch resolution the abort mechanism is used to rollback speculative operations.

It should be noted that the lack of ordering restrictions is loose compared to the requirements of most modern processors. Using these operations one could construct timing models which commit instructions out of program order, or fetch instructions non-sequentially. We specifically chose this level of granularity because it allows timing partitions the flexibility to model all types of speculation. In most cases it makes sense for the functional partition to check that committed instructions follow program order, raising a simulator exception if dependencies are violated.

IV. FPGA IMPLEMENTATION

A. Architecture Overview

We implemented our simulator using the Bluespec SystemVerilog [2] high-level synthesis language. Our FPGA implementation concentrates on making good use of port-limited BlockRAMs while maintaining a high-degree of parallelism. As shown in Figure 10, we use BlockRAMs to track the register state and memory state of the machine, as well as information about in-flight instructions. In-flight instructions are tracked using *tokens*, pointers which allow the timing partition to refer to specific instructions without passing large amounts of data back and forth.

The number of bits used to represent the token determines the number of instructions which may be in-flight simultaneously. This size is set by a static compile-time parameter, allowing it to be increased if a particular value proves insufficient (though doing so will increase the size of all the tables). It is often necessary to compare two in-flight instructions to see which is older (for example, see the store buffer below). For efficiency we wish to do this comparison using the tokens of the instructions. In order for this comparison to function properly we must add the restriction that in-flight instructions are retired (or aborted) in token order. This represents a restriction over the general semantics of our functional partition, but is consistent with the semantics of the architectures we are modeling.

The functional partition operations described previously in Section 3 are implemented as pipelines which read and write the token tables. For example, the `getInstruction` operation writes the address and instruction tables, which are later read by the `getResult` operation. An extra operation, `getToken`, was added to allocate a new in-flight instruction. Furthermore, the `getOperands` and `getResult` operations were merged for efficiency – none of the models we explored here utilized these separately, and by merging them we were able to eliminate intermediate state. A detailed look at the implementation of the operations is given in Figure 11.

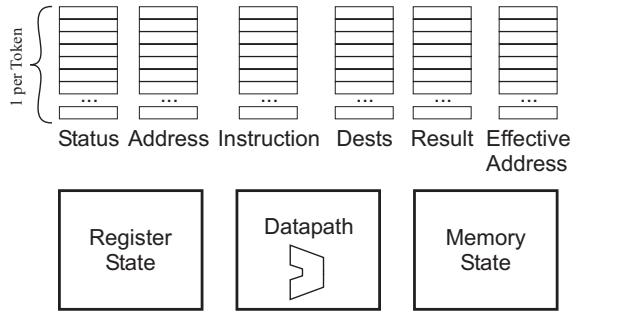


Figure 10: Overview of our functional partition FPGA architecture. BlockRAM tables track information about each in-flight instruction. The register state includes a physical register file, maptable, freelist, and snapshot/rollback mechanism to handle aborts. The memory state includes an on-FPGA cache and a store buffer which determines the youngest store to a particular address.

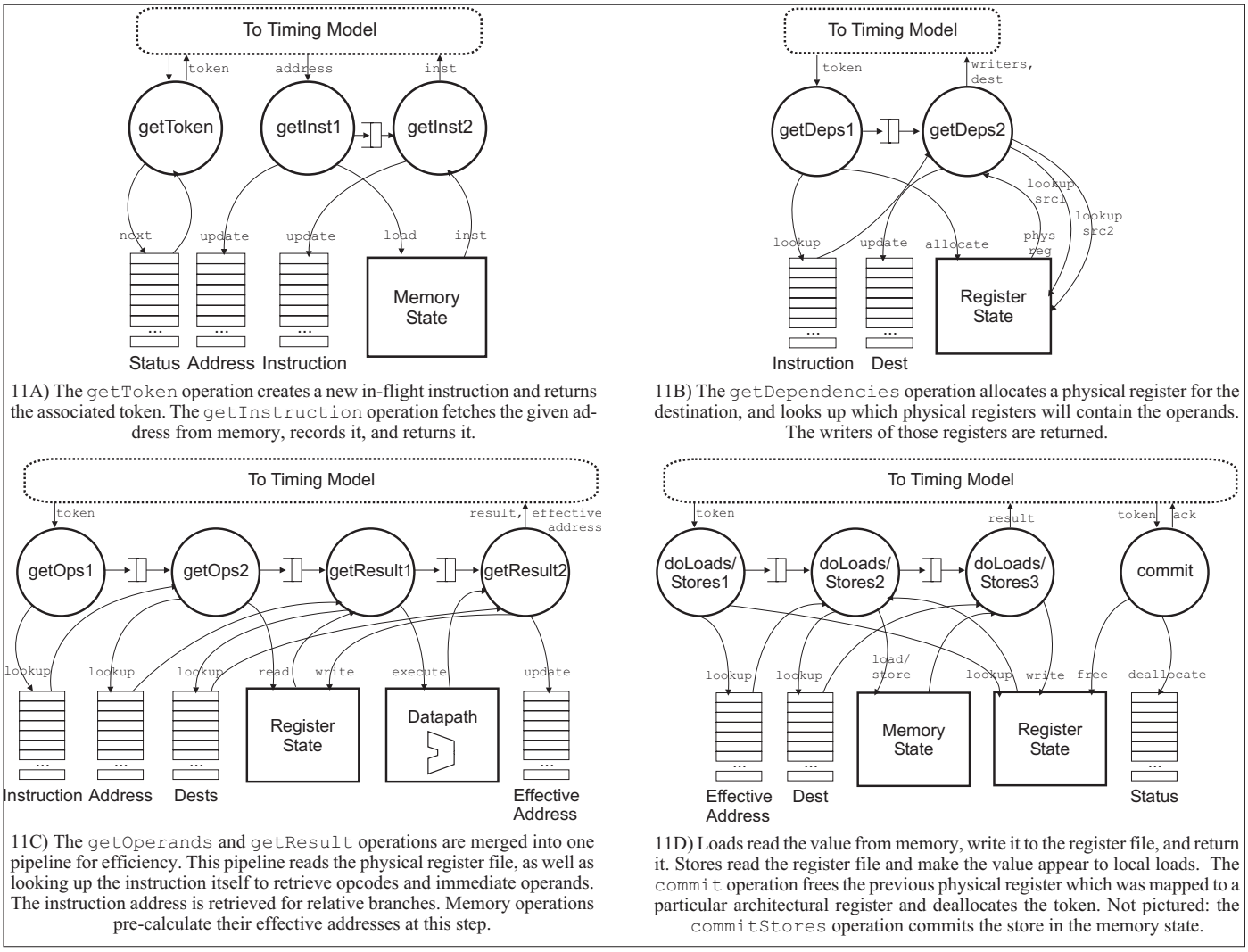
To implement rollbacks the register state was implemented as a physical register file with a maptable and a freelist, as would be found in many out-of-order processors. Rollbacks of stores are supported via a store buffer in the Memory State. In order to perform rollbacks quickly, snapshots of the maptable are made for every branch instruction. This allows the timing model to rewind the state of the machine to a previous branch in 2 FPGA cycles if there is a snapshot. Rewinds to non-snapshotted tokens are much slower as they are done by rewinding to the youngest previous snapshot and sequentially walking the in-flight tokens to recreate the maptable.

V. EVALUATION

A. Timing Model Creation

In order to evaluate our functional partition we identified three target circuits, shown in Figure 12: An unpipelined processor (12A), a 5-stage in-order pipeline (12B), and a MIPS R10K-like 4-way superscalar out-of-order machine (12C). In order to emphasize the differences in the processor pipelines themselves the systems were assumed to be paired with a single-cycle "magic" memory rather than a realistic cache hierarchy. Because ISA research is not the focus of this project, we implemented a subset of the MIPS instruction set. For a detailed discussion of creating efficient timing models for FPGAs see [16].

As we noted earlier, the timing-directed simulation scheme means that the timing model does not need to implement all structures, as some of their functionality is handled by the functional partition. Figure 13A shows a summary of the various structures in the target systems, and the degree to which they were present in the performance models, including a summary of the number of lines of code required to implement each partition (Figures 13B and 13C).



11A) The `getToken` operation creates a new in-flight instruction and returns the associated token. The `getInstruction` operation fetches the given address from memory, records it, and returns it.

11B) The `getDependencies` operation allocates a physical register for the destination, and looks up which physical registers will contain the operands. The writers of those registers are returned.

11C) The `getOperands` and `getResult` operations are merged into one pipeline for efficiency. This pipeline reads the physical register file, as well as looking up the instruction itself to retrieve opcodes and immediate operands. The instruction address is retrieved for relative branches. Memory operations pre-calculate their effective addresses at this step.

11D) Loads read the value from memory, write it to the register file, and return it. Stores read the register file and make the value appear to local loads. The `commit` operation frees the previous physical register which was mapped to a particular architectural register and deallocates the token. Not pictured: the `commitStores` operation commits the store in the memory state.

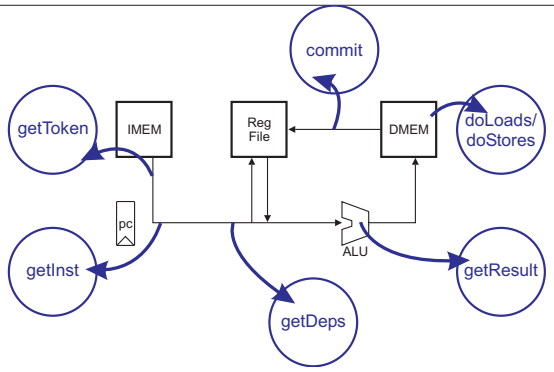
Figure 11: The functional partition operations are implemented as pipelines which read and write the BlockRAM tables, as well as interacting with system memory.

Another motivation for the timing-directed scheme was to encourage reuse. Figure 13D summarizes the code reuse which was possible between the five-stage and out-of-order timing models. First off, the entire functional partition was reused with no changes. Within the timing partition, the greatest possibility for reuse came in the branch predictor, which was reused verbatim. This matches our intuition, as the behavior of a branch predictor is mostly separate from the surrounding pipeline. Partial reuse was possible in the fetch, execute, and data memory modules, which were essentially taken from the 5-stage pipeline and extended to more general, superscalar versions. No such reuse was possible in the decode stage or issue stages, where the out-of-order machine's ROB was different enough from the 5-stage pipeline's simple scoreboard to necessitate full reimplementation.

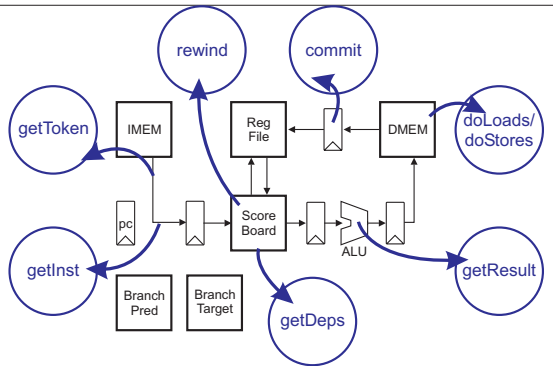
B. Performance Assessment

In order to assess our timing-directed simulation scheme we ran five simple benchmarks: numeric median and multiplication, quick sort, Towers of Hanoi, and vector-vector addition. While we recognize the limitation of trying to assess machines without a realistic memory hierarchy running toy benchmarks, these programs were nevertheless sufficient to demonstrate several interesting features of our simulator. The results of this assessment are given in Figure 14.

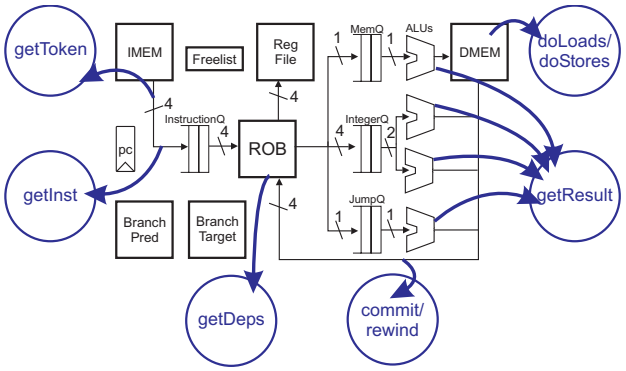
First let us examine the simulation results about the target designs. The performance of the unpipelined processor is unsurprising: it always takes one model cycle to execute every instruction, giving it a CPI of 1. The 5-stage pipeline target achieves an average CPI of 2.7 on our benchmarks. The out-of-order model is 1.5 to 3.6 times faster than the 5-stage, depending on the amount of instruction-level parallelism available. Ultimately these CPIs would be offset by physical factors. Presumably the pipelined processor would ultimately achieve a much higher clock rate than the unpipelined machine. Similarly the performance benefit of the out-of-order machine will ultimately cost increased circuit area. Performance models (whether implemented in an FPGA or in software) do not give insight into these physical aspects. System architects must assess which performance point is most likely to meet the needs of a particular project.



12A) Unpipelined processor target. The implementation executes each functional partition operation before incrementing model time, thus simulating an unpipelined processor. The ALU, IMEM, and DMEM, are not present, as they are simulated entirely via functional partition operations. The register file is also not present, as the functional partition's register state is sufficient.



12B) 5-stage processor target. As with the unpipelined model, the IMEM, DMEM, ALU, and Register File are not present. The branch predictor structure is implemented entirely in the timing model, as it relates to nothing in the functional partition. On mispredicts a `rewind()` is issued to represent a pipeline flush, and simulated no-ops are passed through the pipeline.



12C) Out-of-order, 4-way superscalar target. Again the IMEM, DMEM, and Register File are not present. The branch predictors are reused from the 5-stage model. The superscalar behavior is simulated by making multiple calls to the functional partition before advancing model time. The simulated ROB is substantially simpler than a real ROB, as it does not need to implement the dependency tracking logic. Instead it uses the result of the `getDeps()` operation and then uses a sequential search to determine which instructions should be issued next. The ALUs are not present, as they are represented by multiple calls to the `getResult()` operation.

Figure 12: Target processors and their simulator implementation.

Now let us examine the performance of the simulators themselves. Here the story is quite different: the unpipelined model achieves the slowest simulation rate, whereas the 5-stage is the fastest, with the out-of-order in the middle.

It is not surprising that the unpipelined simulator is slow: the timing model executes every one of the seven functional partition operations for an instruction before even beginning to fetch the next one. Overall this results in 41 FPGA cycles to simulate one model cycle. Timing-directed simulation is not a good match for this target because the model is not able to take advantage of the parallelism available in the functional partition.

The simulator of the 5-stage pipeline, on the other hand, is strictly faster than the simulator of the unpipelined processor, requiring an average of 7.4 FPGA cycles to simulate one model cycle. This is because of two factors: first, the pipelined nature means that the model executes functional partition operations in parallel. Second, the fact that the target circuit stalls the pipeline for back-to-back dependent instructions actually increases simulation rate. Pipeline bubbles are fast to simulate as they do not necessitate invocations of the functional partition.

The superscalar out-of-order simulator is the most difficult to evaluate. Its FPGA-cycle-to-Model-cycle Ratio (FMR) is significantly slower than the 5-stage pipeline (15.6 versus 7.4). In this case, the limiting step of this simulator comes from the timing model itself. While the target machine would implement the out-of-order issue logic using combinational CAMs, these structures are extremely expensive to implement on FPGAs. In order to save resources this logic was simulated using sequential searches of FPGA RAM resources. In the worst case (when the 4 instructions to issue are found consecutively at the bottom of the table) the issue stage could take 32 FPGA cycles to decide which instructions to issue next – though this case never occurred in our benchmarks.

While these tradeoffs slow the rate of simulating clock cycles, the situation is different when we consider the rate of simulating instructions. By this metric the out-of-order simulator is nearly as fast as the 5-stage (5.1 versus 4.7 MIPS). This is because the superscalar nature of the target means that each timing stage executes multiple functional partition operations per clock cycle. Overall both simulators are equally good at keeping the functional partition busy, and while the out-of-order model takes longer to simulate a single cycle, the performance improvement of the model means that it will need to simulate significantly fewer such cycles to run a benchmark into completion.

Design	IMEM	Program Counter	Branch Predictor	Scoreboard/ ROB	Reg File	Maptable/ Freelist	ALU	DMEM	Store Buffer	Snapshots/ Rollback
Functional Partition	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Unpipelined	No	Yes	N/A	N/A	No	N/A	No	No	N/A	N/A
5-Stage	No	Yes	Yes	Partial	No	No	No	No	N/A	No
Out-of-Order	No	Yes	Yes	Partial	No	Partial	No	No	No	No

13A) Almost all large processor structures are implemented within the functional partition. The timing model is responsible for controlling simulation, and thus implements the program counter and branch prediction. The functional partition operations eliminate the need for structures such as the register file and ALU. The dependencies tracking significantly eases the implementation burden of the issue logic, with the timing model generally tracking details like the number of model

Datatypes	Token Tables	Scoreboard	ALU	Register State	Memory State	Store Buffer
691	896	303	487	136	187	433

13B) Lines of Bluespec code to implement the functional partition.

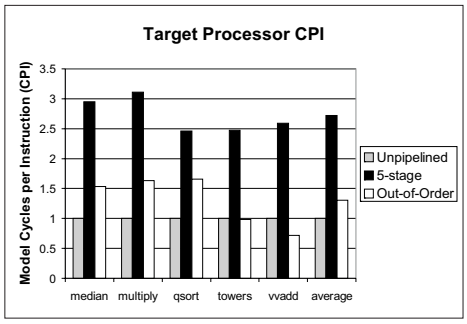
Model	Fetch	Decode/Issue	Execute	Memory Ops	Writeback
Unpipelined	405				
5-Stage	156	190	148	138	93
Out-of-Order	79	953	157	N/A	30

13C) Lines of Bluespec code to implement the various timing models. The out-of-order model Decode stage interacts with the branch predictor, which simplifies the Fetch stage despite its being 4-way superscalar. Similarly the complex ROB simplifies the out-of-order writeback. Memory ops are folded into the execute module.

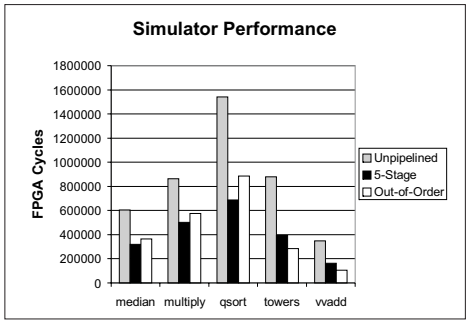
Functional Partition	Fetch	Branch Predictor	Decode	Issue	Execute	Memory Ops	Writeback
Full	Some	Full	None	None	Some	Some	Some

13D) Code reuse between the 5-stage pipeline and the out-of-order model. The entire functional partition was reused with no changes. Most pipeline stages were adapted from being single issue to being superscalar. However the complexity of the out-of-order issue logic meant that the Decode/Issue modules had to be implemented from scratch. The branch predictor was directly reusable with no adaptation. We expect a realistic cache hierarchy to be similarly portable.

Figure 13: Assessment of the benefit of partitioning to model development time.



14A) Target processor performance matches our intuition. The 5-stage pipeline stalls enough that it loses performance. The out-of-order processor's performance is dependent on the amount of instruction-level parallelism available, and thus does best on the vector-vector add benchmark. Physical factors would offset these numbers – we expect the clock rate of the unpipelined processor to be slow, and the area of the out-of-order model to be high.



14B) The performance of the simulators themselves is more interesting. Despite the fact that 5-stage pipeline is a slower target, its simulator actually faster overall than those of the other models. The out-of-order simulator's performance is extremely benchmark dependent. The unpipelined model's lack of parallelism means it cannot exploit the functional partition effectively.

	Unpipelined/ Functional Partition	5-stage	Out of Order
FPGA Slices	6599 (20%)	9220 (28%)	22,873 (69%)
Block RAMs	18 (5%)	25 (7%)	25 (7%)
Clock Speed	98.8 MHZ	96.9 MHZ	95.0 MHZ

14C) Synthesis results for a Virtex 2Pro 70 FPGA platform. Results were obtained using Xilinx ISE 8.2i. The unpipelined timing model requires so few resources that its results can be considered equivalent to that of the functional partition alone.

	Unpipelined	5-stage	Out of Order
Average FPGA-cycle-to-Model-cycle Ratio	41.1	7.49	15.6
Simulation Rate	2.4 MHZ	14 MHZ	6 MHZ
Average Simulator IPS	2.4 MIPS	5.1 MIPS	4.7 MIPS

14C) Impact of FMR on simulation speed. The out-of-order model requires a long time to simulate clock cycles, but simulates instructions nearly as fast as the 5-stage pipeline, as the better target CPI means more instructions are being executed every cycle.

Figure 14: Evaluating the performance of the target circuits, and of the simulators of the targets.

VI. DISCUSSION

In this paper we explored the idea of using timing-directed simulation on FPGAs. Our motivation was to gain a significant increase in model performance while offsetting the increase in development time which comes from using hardware description languages. We implemented three timing models and demonstrated that our partitioning strategy reduced the amount of structures which needed to be implemented, and encouraged code reuse across different timing models. Although the three timing models represented vastly different target systems, we demonstrated that they all could successfully interact with the same functional partition and achieve good rates of simulation.

In the future we hope to implement a complete system simulator with support for multicore models on an FPGA. Additionally we plan to integrate our FPGA simulator with a software component running on a host processor. This software will be used to handle rare but difficult-to-simulate events such as system calls and simulated DMA transfers. While communication with the host processor will undoubtedly be slower than communication on-chip, related results have shown that if the transferred events are rare enough, then the impact on overall execution rate is small [5].

ACKNOWLEDGMENTS

This work was funded by a grant from Intel, and by NSF grant CCF-0541164. Collaboration with the RAMP project was made possible by NSF grant CNS-0551739. The authors would like to acknowledge the collaboration and feedback of Angshuman Parashar, Zhihong Yu, Tao Wang, and Guan-Yi Sun.

BIBLIOGRAPHY

[1] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical Report.

[2] Bluespec, Inc. Bluespec Language Reference Manual.

[3] 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. Proceedings of *International Conference on Computer-Aided Design (ICCAD)*, 2007.

[4] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, D. E. Johnson, J. Keefe and H. Angepat. FPGA-Accelerated Simulation Technologies FAST: Fast, Full-System, Cycle-Accurate Simulators. *Proceedings of MICRO*, 2007.

[5] E. Chung, E. Nurvitadhi, J. Hoe, K. Mai, and B. Falsafi. Accelerating Architectural-level, Full-System Multiprocessor Simulations using FPGAs. In *FPGA '08: Proceedings of the 2003 ACM SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, 2008.

[6] N. Dave, M. Pellauer, Arvind, and J. Emer. Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA. *2nd Workshop on Architecture Research using FPGA Platforms (WARFP)*, February 2006.

[7] 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, February, 2002.

[8] D. Fang, C. LaFrieda, S. Peng, and R. Manohar. A 3-Tier Asynchronous FPGA. In *Proceedings of the 23rd International VLSI/ULSI Multilevel Interconnection Conference*, September 2006.

[9] G. Gibeling, A. Schultz, and K. Asanovic. The RAMP Architecture & Description Language. Technical Report.

[10] S.H. Kim, W.H. Tranter, S.F. Midkiff, Middleware for a distributed reconfigurable simulator. In *Proceedings of the 35th Annual Simulation Symposium*, pp. 253-258, 2002.

[11] E. Larson, S. Chatterjee, and T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Nov. 2001.

[12] S.L. Lu, P. Yiannacouras, T. Suh, R. Kassa, and M. Konow. An FPGA-Based Pentium(R) in a Complete Desktop System. In *Proceedings of the Fifteenth ACM SIGDA International Symposium on Field Programmable Gate Arrays (ISFPGA)*, February 2007

[13] C.J. Mauer, M.D. Hill, and D.A. Wood. Full-System Timing-First Simulation. *ACM SIGMETRICS Performance Evaluation Review*. Volume 30.1, pp. 108-116, 2002

[14] Page, I. Constructing Hardware-Software Systems from a Single Description. In *Journal of VLSI Signal Processing*, Volume 12, pp. 87-107, 1996.

[15] A. Parashar, M. Adler, J. Emer. Facilitating Cross-FPGA Platform Designs using "Virtual" Platforms. The First Bluespec Workshop, August 2007. http://csg.csail.mit.edu/bluespec/

[16] M. Pellauer, M. Vijaraghavan, M. Adler, Arvind, and J. Emer. A-Ports: A Distributed, Efficient Abstraction for Performance Models on FPGAs. *International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, 2008.

[17] 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 *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006, pp. 27-38.

[18] J. Ray and J.C. Hoe. High-Level Modeling and FPGA Prototyping of Microprocessors. In *FPGA '03: Proceedings of the 2003 ACM SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 100-107, 2003.

[19] L. Soule and A. Gupta. Parallel Distributed-Time Logic Simulation. In *IEEE Design and Test*, November 1989, pp. 32-48.

[20] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: A Research Accelerator for Multiple Processors, IEEE Micro Mar/Apr 2007, pp. 46-57.

[21] R. Wunderlich and J. C. Hoe. *In-System FPGA Prototyping of an Itanium Microarchitecture. In Proceedings of International Conference on Computer Design (ICCD)*, October 2004.

[22] Xilinx Inc. Website. http://www.xilinx.com/