The final version of this work can be found at: http://dx.doi.org/10.1109/ICCD.2012.6378650

Interface Design for Synthesized Structural Hybrid Microarchitectural Simulators

Zhuo Ruan and David A. Penry
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA
zruan@et.byu.edu, dpenry@ee.byu.edu

***Abstract*—Computer designers rely upon near-cycle-accurate microarchitectural simulators to explore the design space of new systems. Hybrid simulators which offload simulation work onto FPGAs overcome the speed limitations of software-only simulators as systems become more complex, however, such simulators must be automatically synthesized or the time to design them becomes prohibitive. The performance of a hybrid simulator is significantly affected by how the interface between software and hardware is constructed. We characterize the design space of interfaces for synthesized structural hybrid microarchitectural simulators, provide implementations for several such interfaces, and determine the tradeoffs involved in choosing an efficient design candidate.**

I. Introduction

Computer designers reply upon detailed microarchitectural simulations for near-cycle-accurate performance predictions when they evaluate a proposed system. However, the trend towards multicore designs and increased levels of system integration has resulted in simulators which are too slow to permit extensive exploration of complex future multicore systems [1].

Researchers have proposed to use FPGAs to accelerate microarchitectural simulation, a technique known as *FPGA Architecture Model Execution (FAME)* [1]. A promising class of FAME simulators implements only a portion of the simulator in FPGAs [2], [3], [4]; such *hybrid FAME* simulators allow trade offs to be made between simulator speed, ease of implementation, and FPGA resource utilization.

Three styles of hybrid FAME simulators have been advocated. In a *functional-timing* simulator [2], the functional behavior of instructions is performed in software (SW) while the time it takes to execute an instruction is calculated in hardware (HW). In a *transplanting* simulator [3], most behavior of the simulator is implemented in HW, but when a difficult-to-implement operation such as simulated I/O device behavior must be performed, the HW calls upon the SW to complete the operation. Finally, in a *structural* simulator [4], a collection of processes from a structural simulation model (a model which is specified in terms of concurrent processes and signals) is implemented in HW.

Development of any of these hybrid FAME simulators has generally required extensive design effort. Therefore, we proposed the Simulator Partitioning Research Infrastructure (SPRI) in [5] to synthesize structural hybrid FAME simulators from an existing SW-only simulator written in a structural simulation framework such as SystemC [6].[1]

A hybrid simulator must have some interface between the SW and HW portions of the simulator. This work is the first effort to explore the interface design space for structural hybrid FAME simulators and show that the interface design greatly impacts the simulator performance and the FPGA utilization. Our contributions are:

1) identifying the design space of interfaces for synthesized structural hybrid FAME simulators.
2) providing synthesis methodologies for several such interfaces in the design space.
3) determining the trade offs between simulator performance and FPGA utilization which must be considered when choosing an interface design.

We demonstrate these contributions by implementing the synthesis of several such interfaces within the SPRI infrastructure. We demonstrate that the key elements to improve performance are increased concurrency and internalized communication between HW processes, however, at the cost of HW resources.

II. Interface Design Space

The SW/HW interface of a hybrid FAME structural simulator must coordinate the production of new signal values and the update of state between the SW and HW processes. There are two dimensions along which the design space of SW/HW interfaces can be characterized: concurrency and composition.

A. Concurrency

The first dimension is the amount of concurrency provided by the interface. This concurrency may exist between HW elements and between HW and SW. In general, as concurrency increases, we would expect higher simulator performance. At one extreme, concurrency may be non-existent. When either SW or HW requires an operation of the other, a request is made through the interface and it blocks execution until the request has been handled [4], [3]. At the other extreme, SW

This work was supported by National Science Foundation grant CCF-1017004.

[1]The use of a structural simulation framework greatly simplifies specification of the SW-based simulation model [7] and has previously been shown to allow easier partitioning of HW and SW [4].

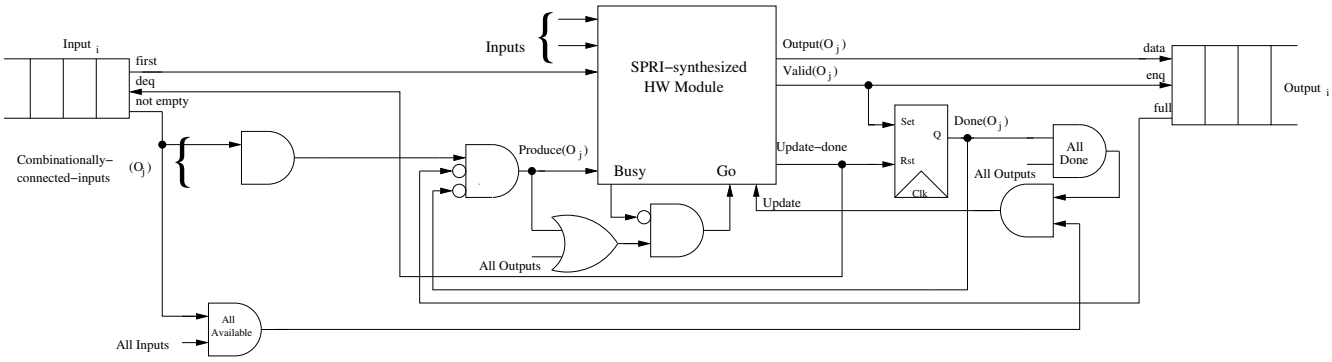

Fig. 1. SPRI-synthesized LI-BDN Wrapper [8]

and HW can be fully concurrent: either SW or HW may initiate multiple requests to each other and continue on its own work without waiting for the completion of their requests [2].

B. Composition

Composition is the direct interconnection of a set of HW processes. Signals internal to the set do not need to be communicated through the interface. In general, as the amount of communication decreases, we should expect higher simulator performance. We consider three degrees of composition:

- **No composition:** All communication between HW processes takes place via SW.
- **Single-FPGA-cycle composition:** Only single-FPGA-cycle HW processes are composed.
- **Multi-FPGA-cycle composition:** All HW processes are composed.

The distinction between single-FPGA-cycle and multi-FPGA-cycle composition is an important one. SystemC processes execute instantaneously with respect to simulated time. However, efficient FPGA implementations of those processes may require several FPGA cycles to compute the outputs and next state of one simulated cycle. We call FPGA process implementations which require only a single cycle to compute *single-FPGA-cycle HW processes* and those that require multiple cycles *multi-FPGA-cycle* HW processes.

Single-FPGA-cycle HW processes may be composed directly without affecting their correctness because the processes can directly implement the state machine being modeled. However, as described in [8], multi-FPGA-cycle HW processes cannot be directly composed because the processes actually implement a different state machine which requires multiple FPGA cycles to *simulate* the modeled state machine. Thus single-FPGA-cycle composition is fundamentally different from multi-FPGA-cycle composition.

Multi-FPGA-cycle composition is possible if the HW processes are designed to be latency-insensitive. Latency insensitive processes could be designed by the user, however, in order to *synthesize* an arbitrary structural process into a latency-sensitive version, some formal methodology for creating latency-insensitive processes is required. This methodology can be provided by Latency-Insensitive Bounded Dataflow Networks (LI-BDNs), which are formalized in [9]. HW processes can be wrapped to form LI-BDN processes using the interface and control logic shown in Figure 1. After LI-BDN transformation, LI-BDN processes are connected through FIFOs and simulated time is interpreted as enqueue and dequeue operations on the FIFOs: in one simulated clock cycle, each FIFO is enqueued and dequeued once.[2] LI-BDN processes execute autonomously: when their inputs become available, they produce outputs and update state.

Designation	Concurrency	Composition	Notes
BL-NONE	Blocking	None	
BL-SC	Blocking	Single-FPGA-cycle	
BL-MC	Blocking	Multi-FPGA-cycle	Nonsensical
NB-NONE	Non-blocking	None	
NB-SC	Non-blocking	Single-FPGA-cycle	
NB-MC	Non-blocking	Multi-FPGA-cycle	

TABLE I
THE INTERFACE DESIGN SPACE

C. Combining the dimensions: the interface design space

Table I lists each of the points in the interface design space as well as a name for each interface design style. Note that one design choice – BL-MC – is nonsensical; because LI-BDNs execute autonomously, there can be no notion of SW making a request for the HW to compute and blocking to wait until HW finishes.

The non-blocking interfaces should have better performance than the blocking interfaces due to their higher concurrency. Furthermore, interfaces with more composition should perform better than interfaces with less composition due to reduced communication overhead. However, multi-FPGA-cycle composition could have higher FPGA utilization caused by the LI-BDN transformation. Note that it is somewhat obvious that BL-NONE should perform much worse than the other interfaces. However, we still describe and evaluate BL-NONE because it corresponds to the natural "co-processor" model of using hardware accelerators: simply request the hardware to do some computation and then wait for it. Furthermore, previous

[2]Not all models can be composed in this fashion, as cycles of data dependence are not allowed. A detailed description of the LI-BDN implementation of HW processes is given in [8].

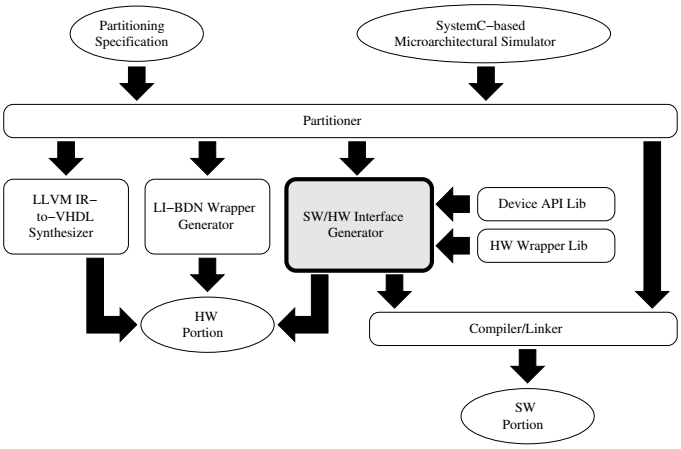

Fig. 2. SPRI Block Diagram

structural hybrid simulators [4], [10] have, in fact, used this mechanism.

III. Interface synthesis mechanisms

SPRI [11] automates the synthesis of structural hybrid FAME simulators from SystemC models. It is based on the LLVM compiler framework [12] and contains a partitioner, an interface generator, and a VHDL synthesizer. The partitioner takes the LLVM representation of a compiled SystemC model as input and partitions it into a SW portion and a HW portion. The interface generator creates HW and SW implementing the interface between the two portions of the simulator. The VHDL synthesizer produces VHDL components which implement the SystemC processes in the HW partition. The block diagram of SPRI is shown in Figure 2; this paper will focus on the implementation of the SW/HW interface generator.

The SPRI SW/HW interface generator performs three tasks. First, it produces a corresponding HW wrapper for each interface style, which bridges the communication channel of the target FPGA accelerator platform and the synthesized HW processes. Second, it creates new SystemC processes which communicate with the HW using accelerator platform API calls and device driver calls. Third, it generates code and constructs events and processes to "switch over" from SW simulation to hybrid simulation at runtime and "fix up" runtime information required by different interfaces (e.g signal sensitivity). The new processes and events are carefully constructed to ensure that interface code runs at the desired points within the main SystemC simulation loop without modifying the code of the simulation loop.

IV. Interface designs

A. Blocking, no composition (BL-NONE)

This interface blocks after making requests of the HW and allows no direct communication between HW processes. The SW side of the interface simply replaces the SW process corresponding to each HW-implemented process with a *proxy*

```
proxy() { // sensitive to input signals or clock edge
   foreach input signal s
      WriteDataToHW(s.read());
   // HW begins execution and updates state
   WaitForHWFinish();

   foreach output signal s
      s.write(ReadDataFromHW());
}
```

Fig. 3. BL-NONE Interface: SW Side

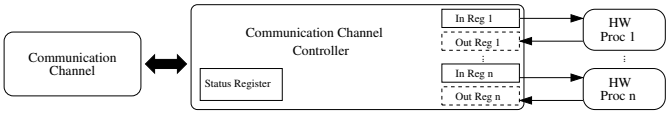

Fig. 4. BL-NONE Interface: HW Side

process sensitive to the same set of signals.[3] As shown in Figure 3, each proxy process first reads input signals from SystemC and sends them to the HW. It then waits for the HW to finish execution, reads the values of the output signals from HW, and writes them to the corresponding SystemC signals. The HW automatically begins execution when the last input signal is written; if the process is sensitive to the clock, the HW updates state as part of its execution.

On the SW side, better performance is achieved by staging the signal values through a memory buffer, transferring the entire buffer at once rather than each signal value individually. We speculatively read the HW outputs at the same time we poll the status register; in the common case of HW finishing before the poll, the poll and data transfer occur with only a single device driver call.

The HW side of the interface (Figure 4) has a communication channel controller (CCC) which manages the interface. Signal values are stored in distributed registers which are memory-mapped into the communication channel's address space. These registers are connected directly to the HW processes. There is also a memory-mapped completion status register. As a size optimization, output signals driven by single-FPGA-cycle HW processes do not have output registers, because such processes produce their outputs continuously. The CCC triggers the execution of a HW process when the HW process's last input signal is written; the CCC sets a completion bit in the status register when it receives a "done" signal from a HW process. The SW must read the status register to determine data availability before it reads and the CCC must set the completion bit only after all the output signal registers for a process are valid. The status register is mapped to address 0 so that a sequential read of addresses will produce the correct ordering.

B. Blocking, single-FPGA-cycle composition (BL-SC)

This interface blocks after making requests of the HW but permits direct composition of single-FPGA-cycle HW

[3] A process sensitive to no signals will have a proxy which is never fired; this behavior is correct, as the process must be driving a constant value which will be driven during the hybrid simulator's initialization.

```
proxy() { // sensitive to input signals or clock edge
   foreach cross-boundary input signal s
      copytoBuffer(s.read());
   if (sensitive to clock edge)
      clockedTransferEv.notify();
   else
      nonclockedTransferEv.notify();
}

transfer() { // sensitive to a transfer event
   WriteDataToHW(data_from_buffer);
   // HW begins execution and updates state
   WaitForHWFinish();
   foreach cross-boundary output signal s
            of the process set
      s.write(ReadDataFromHW());
}
```

Fig. 5. BL-SC Interface: SW Side

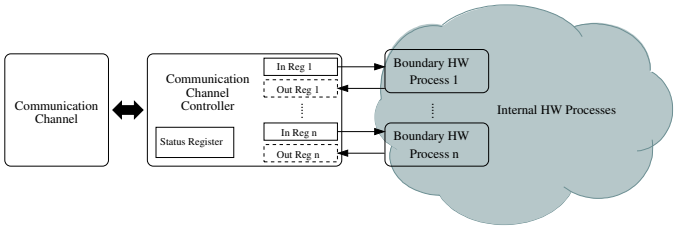

Fig. 6. BL-SC Interface: HW Side

processes. All single-FPGA-cycle processes are composed to internalize communication.

The SW side of the interface has a proxy process for each process that has input signals across the SW/HW interface. All single-FPGA-cycle processes are grouped into a single *process set*. Two *transfer* processes are required to transfer data and update values, one for the clock-sensitive processes in the process set and another for the non-clock sensitive processes in the process set. The two-step proxy/transfer flow is motivated by a desire to efficiently transfer data for many signals with a single device driver call. Figure 5 gives pseudo-code for these processes.

A proxy process copies its cross-boundary input signal data into a buffer and triggers a transfer process to run using a SystemC immediate notification. Each transfer process writes the data from the buffer to the HW and starts HW execution. After the HW finishes, the transfer process reads the cross-boundary output signals from HW and writes them to the corresponding SystemC signals. Output signals are polled speculatively as in BL-NONE.

The transfer processes for clock-sensitive and non-clock-sensitive processes are different. First, the clock-sensitive transfer process writes one additional word of data to the HW which the HW interprets as an UPDATE STATE command. Second, while the non-clock-sensitive process transfers all the cross-boundary outputs of non-clock-sensitive processes, the clock-sensitive process must transfer the cross-boundary outputs of all processes. This difference occurs because the state update in the HW may have changed signal values which are combinationally dependent upon the state and thus affected the outputs of non-clock-sensitive processes.

The structure of the HW side of the interface is shown in Figure 6. It is quite similar to that of BL-NONE, except that the CCC need only consider cross-boundary inputs and outputs. The CCC asserts an UPDATE STATE signal to all processes in a process set when a dummy address just beyond the last cross-boundary signal value's register for that set is written. This extra dummy address provides easy control over whether state updates occur. Only a single completion status bit is maintained for each process set.

C. Non-blocking, no composition (NB-NONE)

This interface does not block after making requests of the HW but allows no direct communication between HW processes. To achieve a non-blocking interface, the SW side of the interface must separate the requests it makes to HW from the checks it makes for the completion of those requests. It must also execute any runnable SW processes without waiting for HW completion. The SW side has a proxy process for every HW-implemented process, two transfer processes, and two *completion* processes. The pseudo-code for these processes is shown in Figure 7.

The proxy process is similar to that of BL-SC, but also sets a flag indicating that HW execution has been requested. As in BL-SC, the transfer processes write data to HW, however, they do not wait for the HW to complete. Instead, they schedule a completion process to run later by firing an event to which the completion process is sensitive. There is one transfer process for all clock-sensitive proxy processes and one transfer process for all non-clock-sensitive proxy processes; the clock-sensitive transfer process writes an additional word of data to indicate UPDATE STATE to the HW, as in the BL-SC interface.

There is a completion process for each transfer process. Each completion process begins by determining which HW processes are executing by reading a HW status register. The completion process then reads the output signals of each non-executing process whose proxy process has requested HW process execution and writes them to the corresponding SystemC signals, clearing the proxy process' request flag. Finally, if any of the corresponding HW processes are still executing, the completion process schedules itself to run again.

The HW side of the interface is very similar to that of the BL-NONE interface. Indeed, the same HW could be used. However, we optimize this HW slightly by using only two trigger signals, one for clock-sensitive processes and one for non-clock-sensitive processes. The memory mapping of the input signal registers is arranged such that the input signals of each of these two groups of processes are located at consecutive addresses and thus the last signal to be written to in each group may be detected. Reducing the number of trigger signals slightly reduces the FPGA resources required.

D. Non-blocking, single-FPGA-cycle composition (NB-SC)

This interface does not block after making requests of the HW and permits direct communication between HW processes which execute in a single FPGA cycle. Processes are grouped into process sets such that single-FPGA-cycle processes which

```
proxy() { // sensitive to input signals or clock edge
   this.request = true;
   foreach input signal s
      copytoBuffer(s.read());
   if sensitive to clock edge
      clockedTransferEv.notify();
   else
      nonclockedTransferEv.notify();
}

transfer() { // sensitive to a transfer event
   if !busy or this is the clocked transfer process
      WriteDataToHW(data_from_buffer);
      // HW begins execution and updates state
      busy = true;
      if this is the clocked transfer process
         clockedCompletionEvent.notify(SC_ZERO_TIME);
      else
         nonclockedCompletionEvent.notify(SC_ZERO_TIME);
   else
      mustRedo = true;
}

completion() { // sensitive to a completion event
   checkHWstatus();
   foreach proxy process p such that the
            corresponding HW process is not active
            and p.request is true
      foreach output signal s of p
         s.write(ReadDataFromHW());
      p.request = false;
   if no HW processes are active
      busy = false;
      if mustRedo
         mustRedo = false;
         nonclockedTransferEv.notify(SC_ZERO_TIME);
   else
      if this is the clocked completion process
         clockedCompletionEvent.notify(SC_ZERO_TIME);
      else
         nonclockedCompletionEvent.notify(SC_ZERO_TIME);
}
```

Fig. 7. NB-NONE Interface: SW Side

share an input or output signal are in the same set. The SW side of the interface is structured identically to the SW side of the NB-NONE interface, with four differences:

1) Processes whose input/output signals are completely contained within HW have no proxy processes.
2) There are two transfer and two completion processes for each process set. There is a pair of busy and redo flags for each process set.
3) Proxy, transfer, and completion processes only read, transfer, and write signals which are on the boundary between HW and SW. Transfer and completion processes only transfer and write those signals which are destined for or produced by their process set.
4) There are no individual request flags for the proxy processes. The non-clock-sensitive completion process writes all boundary signals produced by non-clock-sensitive processes in its process set. The clock-sensitive completion process writes all boundary signals produced by all processes in its process set.

The HW side of the interface is identical to that of the BL-SC interface.

```
launch() { // sensitive to clock edge
   transferEvent.notify(SC_ZERO_TIME);
   completionEvent.notify(SC_ZERO_TIME);
}

transfer() { // sensitive to transferEvent
             //   and all boundary input signals
   foreach boundary input-to-SW signal s
      if s.available
         copytoBuffer(s.read());
      set/clear available flag for s in buffer
   if any signal is available
      WriteDataToHW(data_from_buffer);
}

completion() { // sensitive to completionEvent
   foreach boundary output-to-HW signal s
      if s.availableInHW();
         s.write(ReadDataFromHW());
         s.available = true;
   if any boundary output signal is not available in HW
      completionEvent.notify(SC_ZERO_TIME);
}
```

Fig. 8. NB-MC: SW Side

*E. Non-blocking, multi-FPGA-cycle composition (*NB-MC*)*

The final interface does not block after making requests of the HW and permits composition of multi-FPGA-cycle HW processes. Such composition requires the use of a latency-insensitive HW implementation method such as LI-BDNs and a rather different interface. Two issues must be addressed. First, because time in LI-BDNs is measured in enqueue and dequeue operations, the interface must ensure that exactly one value per boundary input signal to the HW is enqueued per simulated cycle. Second, LI-BDNs do not have a centralized notion of time as the SW does; individual LI-BDN elements may "slip" in time from one another. Therefore, there must be a mechanism to allow the boundary LI-BDNs and SW to coordinate updates of simulation time. Furthermore, signals must be operated upon individually as their values become *available*, unlike the previous interfaces.

The SW side of the interface is shown in Figure 8. There is one launch process, one transfer process, and one completion process. The launch process is made sensitive to the clock edge and serves solely to schedule the completion and transfer processes to run later. Note that there is no distinction made between clocked processes and unclocked processes.

The transfer process is made sensitive to every boundary input signal to the HW as well as the event which triggers it at the start of the clock cycle. This process checks whether each signal is marked as available; if it is, its value is copied into a buffer and its availability flag in the buffer is marked. Then the buffer is written to HW, which will enqueue the available signals which have not yet been enqueued in this cycle.

The completion process reads the boundary output signals as well as their availability. If a signal is available from the HW and has not yet been made available to SystemC in this clock cycle, the signal is written to SystemC and set to be available. The completion process reschedules itself to run again if

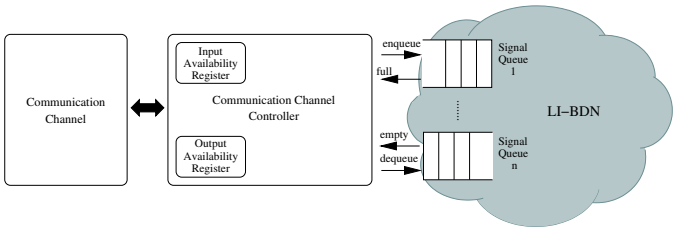

Fig. 9. NB-MC Interface: HW Side

any output signals are not yet available in the HW.[4] This rescheduling allows the interface to be non-blocking. As an optimization, all signal values and availability are speculatively read from the HW in one device driver call, which in many cases will reduce the number of transfers per cycle required.

The HW side of the interface is shown in Figure 9. The CCC is connected to the LI-BDN through FIFOs. The CCC contains input signal value registers but no output signal value registers. It also contains availability registers for both input and output signals. LI-BDNs fire autonomously as their inputs become available, thereby requiring no process control signals.

The CCC sets a bit in the output availability register when an output FIFO is not empty. The heads of the output FIFOs can be read without dequeing. The CCC also maintains a simple state machine for each boundary-crossing input signal; this state machine ensures that the signal is only enqueued once per simulated clock cycle.

The end of a simulated clock cycle is detected when all of the input FIFOs have been enqueued, all of the output signals are available, and the CCC has serviced a read of the output signals in which all signals were available. When this occurs, all input state machines are reset, the output signal availability register is reset, and all output FIFOs are dequeued. To prevent race conditions, SW must ensure that it writes the input signal availability after writing the values. Likewise, it must read the output signal values after reading the output availability. To make this easy to achieve in block writes and reads, the input signal availability register is memory mapped at an address just after the input signal values and the output signal availability register is mapped at address 0.

V. Evaluation

We evaluate the performance and HW resource utilization of the five explored interface designs for hybrid FAME simulators. The hybrid simulators are synthesized by SPRI from a microarchitectural model of a chip multiprocessor which contains a parameterizable number of simple, in-order PowerPC cores and a simplistic cache hierarchy. We chose a speculative-functional-first [13] simulator organization: all instruction-set functional behavior is separated from the timing behavior. SPRI keeps the functional behavior processes in SW and synthesizes the timing behavior processes into HW. This organization, partitioning, and simple model were chosen

[4]The SystemC processes which remain in SW require modifications to propagate signal availability. Details of how these modifications can be made within SPRI are beyond the scope of this paper.

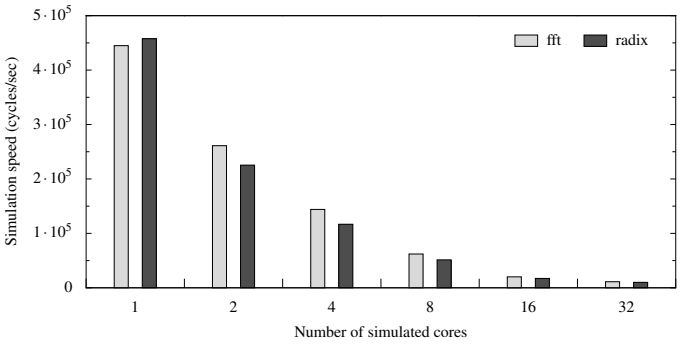

Fig. 10. SW-only Simulation Speed

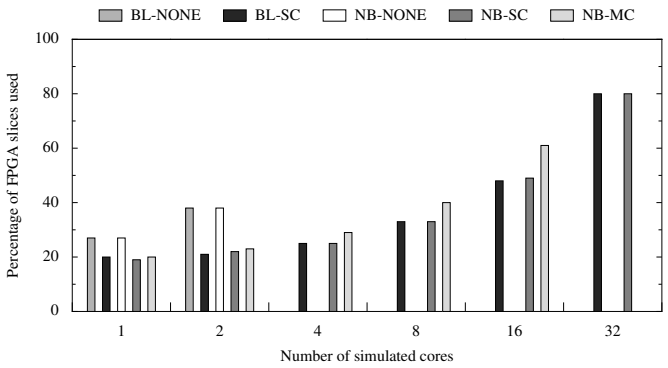

Fig. 11. FPGA Slice Utilization

to highlight the differences between the five interfaces and demonstrate their scalability. Each core model in HW has independent communication with the SW; the size of data to be communicated grows linearly with the size of the model. The cores are simple so that we may fit large numbers of them on the FPGA and so that interface overhead is more pronounced. Because the core models are so simple, little internal parallelism can be exploited in HW. As a result, we expect hybrid-simulator speedup to be mainly due to exposing the parallelism between cores. This parallelism is limited by the number of cores, so we expect speedups to not be particularly high, however, there will be significant differences between the interfaces.

Hybrid simulators with the five interfaces are synthesized for six different core counts. The experiments are carried out on a DRC 1000 system with a dual-core AMD Opteron-275 CPU running at 2.1 GHz with 2 GB of system memory and a Xilinx XC4VLX60-11 FPGA as a coprocessor, and the VHDL-to-bitstream synthesis was done by ISE 8.2. The simulator is run on the FFT and Radix kernels from the SPLASH-2 [14] benchmark suite. LLVM 2.9 was used for compilation. Figure 10 shows the simulation speed for the SW-only simulators using the reference implementation of SystemC 2.0.1.

Figure 11 displays the FPGA slice utilization for the synthesized hybrid PowerPC microarchitectural simulators. Missing bars correspond to hybrid simulators which could not be created due to FPGA capacity problems of two kinds. First, routing resources are limited for the non-composition

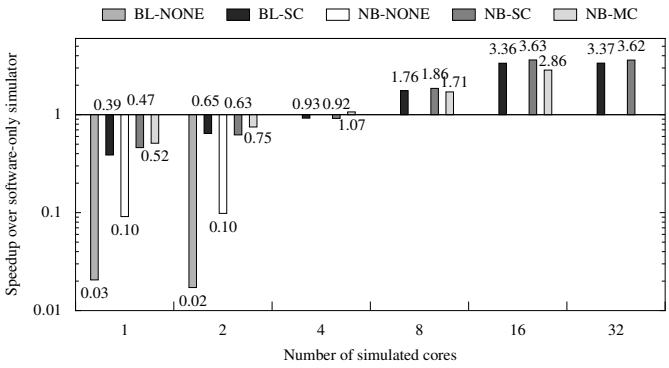

Fig. 12. Speedup of Hybrid over SW-only Simulation (FFT)

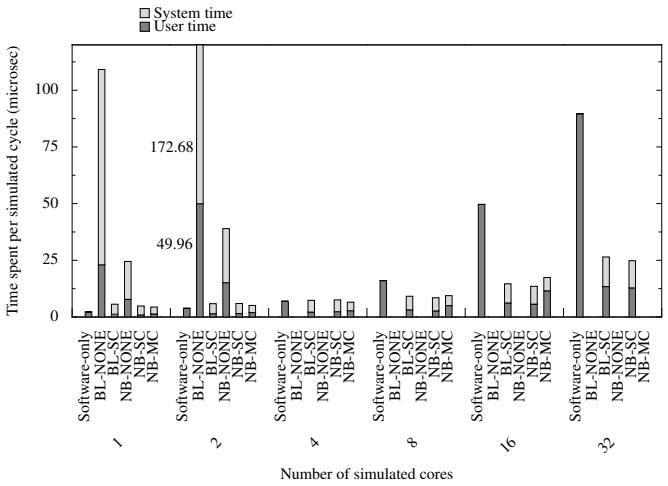

Fig. 13. Execution Time Breakdowns (FFT)

interfaces. More than two cores for BL-NONE and NB-NONE failed to route, because the large number of output signals cannot be muxed together within the CCC. Second, multi-FPGA-cycle composition requires more slice resources because of the FIFOs and control logic between processes.

Figure 12 shows the speedup over the baseline SW-only simulator achieved by the hybrid FAME simulators with the five interfaces. The FFT and Radix benchmarks have similar results, so only the results of FFT are shown. Missing bars indicate that the FPGA runs out of resources for the corresponding simulators. As expected, even the best speedups for this model are not large, due to the simplicity and lack of internal concurrency of this model. However, there is an enormous difference between interfaces: up to a 34x difference in speed. This difference is caused primarily by reducing the number of round-trip communications and reducing the amount of data to be transferred.

The minimum number of round-trip communications for the BL-NONE interface is equal to the number of HW processes. (There may be more round-trips if a HW process has not completed execution when it is polled.) The minimum number of round-trips for the BL-SC, NB-SC, and NB-NONE interfaces is equal to the maximum number of times a combinational path may cross from HW to SW plus one for the clock-sensitive processes. The partitioning and model used in this evaluation result in two round-trips. The minimum number of round-trips for the NB-MC interface is equal to the maximum number of times a combinational path may cross from HW to SW. No additional round-trip for clock-sensitive processes is necessary. For the partitioning and model we have used, the result is one round-trip. While NB-MC always has an advantage in terms of round-trips, it shows additional overhead in the user time; this overhead is spent in computing signal availability for SW processes.

The two non-composing interfaces transfer much more data than the other interfaces because all input and output signals of HW processes must cross the interface. Furthermore, increasing the number of simulated cores in HW causes more signals to cross between SW and HW for all interfaces with the partitioning we have used. The increasing amount of transferred data eventually saturates the CPU-to-FPGA communication bandwidth, which thereby becomes the performance bottleneck: when scaling to 32 simulated cores with the NB-SC and BL-SC interfaces, the speedups of simulating 32 cores are not larger than those of 16 cores.

Additionally, a certain amount of time is spent in the operating system, because the device driver called for communication spins while waiting for operations to complete. Figure 13 shows the amount of time spent per cycle in both system and user modes. The SW-only model bars indicate how the amount of work which the baseline simulator must do per cycle grows rapidly as the models increase in size. The two non-composing interfaces spend the vast majority of their time in communication, because of the large amounts of data to be transferred in both interfaces and the many round-trip communications in each cycle in the case of BL-NONE. In user mode, they both do 10 to 15 times more work which stems from copying signal data from place to place in the interface code. Thus both interfaces result in significant slowdowns. On the other hand, the amount of time spent in communication grows very slowly for the composing interfaces and the amount of time spent executing processes in SW increases far less rapidly than it does for the SW-only model. Thus, as the number of cores increases, the increasing amounts of offloaded concurrent work lead to higher speedups.

To summarize, the combination of module composition and non-blocking communication overcomes the speed limitations by permitting HW concurrency, reducing the communication overhead, and overlapping communication and computation. The two preferred interfaces are NB-SC and NB-MC. NB-SC should be used whenever single-FPGA-cycle composition is possible, as it is both faster and less resource-intensive. NB-MC should be used when there is a large number of multi-FPGA-cycle processes, in order to allow composition to reduce the amount of data to be transferred.

VI. Conclusion

This paper has explored the interface design space of structural hybrid FAME simulators and discussed the synthesis methodology of such interfaces. It is shown that the style

of SW/HW interface can greatly affect both simulator speed and FPGA resource usage. This best interface should be non-blocking and support composition of HW processes. As a result, synthesized hybrid FAME simulators promise faster performance without requiring excessive simulator design effort, thereby allowing computer architects to explore new systems more thoroughly.

REFERENCES

[1] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proceedings of the 37th International Symposium on Computer Architecture*, 2010, pp. 290–301.

[2] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST methodology for high-speed SoC/computer simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 295–302.

[3] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsfi, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 2, June 2009.

[4] 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*, February 2006, pp. 29–40.

[5] D. A. Penry, Z. Ruan, and K. Rehme, "An infrastructure for HW/SW partitioning and synthesis of architectural simulators," in *2nd Workshop on Architectural Research Prototyping*, 2007.

[6] *IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual*. IEEE, 2005.

[7] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The Liberty Simulation Environment: A deliberate approach to high-level system modeling," *ACM Transactions on Computer Systems*, vol. 24, no. 3, August 2006.

[8] T. S. Harris, Z. Ruan, and D. A. Penry, "Techniques for LI-BDN synthesis for hybrid microarchitectural simulation," in *Proceedings of the 2011 International Conference on Computer Design*, 2011, pp. 253–260.

[9] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proceedings of the 7th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, July 2009.

[10] Z. Ruan and D. A. Penry, "Partitioning and synthesis for hybrid architectural simulators," in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems*, 2010.

[11] Z. Ruan, K. Rehme, and D. A. Penry, "SPRI: Simulator partitioning research infrastructure," in *3rd Workshop on Architectural Research Prototyping*, 2008.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[13] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Computer Architecture Letters*, vol. 8, no. 2, July 2009.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 24–36.