© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Improving the Interface Performance of Synthesized Structural FAME Simulators through Scheduling

David A. Penry
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA
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 (also known as FAME simulators) can overcome the speed limitations of software-only simulators. However such simulators must be automatically synthesized or the time to design them becomes prohibitive. Previous work has shown that synthesized simulators should use a latency-insensitive design style in the hardware and a concurrent interface with the software.**

We show that the performance of the interface in such a simulator can be improved significantly by scheduling all communication between hardware and software. Scheduling reduces the amount of hardware/software communication and reduces software overhead. Scheduling is made possible by exploiting the properties of the latency-insensitive design technique recommended in previous work. We observe speedups of up to 1.54 versus the former interface for a multi-core simulator.

I. Introduction

Computer designers rely 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.

Development of any of these hybrid FAME simulators has generally required extensive design effort. One powerful means to reduce this design effort would be to synthesize hybrid FAME simulators automatically from an existing software-only simulator [5]. Simulators written in a structural simulation framework such as SystemC [6] are particularly amenable to synthesis [7], [8].

The performance of a synthesized structural hybrid FAME simulator depends critically upon the design of the interface between software and hardware. Ruan and Penry [9] showed that the interface must permit both concurrent execution of hardware and software processes and the composition, or internal connection, of arbitrary processes within the hardware. A concurrent interface combined with latency-insensitive design of the hardware processes [10] meets these conditions.

When a concurrent interface is used with latency-insensitive design, the primary performance-limiting factor becomes the time spent in communication between hardware and software. A secondary factor is the overhead of certain software changes required to allow software to interact with latency-insensitive processes. In this paper we show that the properties of Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) – the latency-insensitive design technique used by Ruan and Penry [9] – are consistent with those of a particular software model of computation. This model of computation permits us to generate a static schedule of communication. This schedule can then be used to reduce the number of individual communications and to eliminate the aforementioned software changes, leading to significant performance improvement.

Our contributions are:

1) an algorithm for computing a communications schedule based upon the properties of LI-BDNs.
2) a synthesis methodology for a scheduled interface.
3) measurements of the effectiveness of scheduled communication in a structural hybrid FAME simulator.

We demonstrate these contributions by implementing communication scheduling within the structural hybrid FAME simulator synthesis infrastructure of [9] and comparing the performance of scheduled and non-scheduled communication.

II. Interfaces for Structural Hybrid FAME Simulators

This section reviews the design space of the hardware/ software interface for structural hybrid FAME simulators, describes the recommended interface, and discusses interface factors affecting simulator performance.

A. Interface Design Space

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

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

```
launch():                    // sensitive to clock edge
   transferEvent.notify(SC_ZERO_TIME)
   completionEvent.notify(SC_ZERO_TIME)

transfer():     // sensitive to transferEvent
                // and all software-to-hardware signals
   foreach software-to-hardware signal s:
      if s.available:
         copytoBuffer(s.read())
      set/clear available flag for s in buffer
   if any signal is available:
      WriteDataToHardware(data_from_buffer)

completion():                // sensitive to completionEvent
   foreach hardware-to-software signal s:
      if s.availableInHardware():
         s.write(ReadDataFromHardware())
         s.available = true
   if any signal is not available in hardware:
      completionEvent.notify(SC_ZERO_TIME)
```

Fig. 1. NBMC Interface: Software side [9]

the amount of concurrency provided by the interface between hardware elements and between hardware and software. The second dimension is the amount of composition, or direct interconnection, permitted between sets of hardware processes.

Ruan and Penry [9] explored this design space. As might be expected, the interface design with the highest concurrency and composition performed best. What was unexpected was the magnitude of the performance difference: up to 34x between this design and a blocking, non-composing one. This result is even more startling when one considers that FPGA accelerators are designed with a blocking, non-composing coprocessor-style interface in mind and that previous work [3], [4], [7] used a coprocessor-style of interface. Thus, they concluded that a structural hybrid FAME simulator should use a concurrent interface permitting multi-cycle composition. This interface is known as the *NBMC* interface.

B. The NBMC Interface

The NBMC interface permits concurrent hardware and software execution as well as composition of multi-FPGA-cycle hardware processes. Multi-FPGA-cycle processes are those processes that require multiple FPGA cycles to execute. Single-FPGA-cycle hardware processes may be composed directly without affecting their correctness because the processes can directly implement the state machine being modeled. However, multi-FPGA-cycle hardware processes cannot be directly composed because the processes actually implement a state machine which requires multiple FPGA cycles to *simulate* the modeled state machine [10].

Multi-FPGA-cycle composition is possible if the hardware processes are synthesized to be latency-insensitive. Such synthesis depends upon a formal latency-insensitive design methodology; such a methodology is provided by Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) [11]. 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. LI-BDN processes execute autonomously: when their inputs become *available*, they produce outputs and update state. No request from the interface is necessary, providing concurrency between hardware and software. Thus

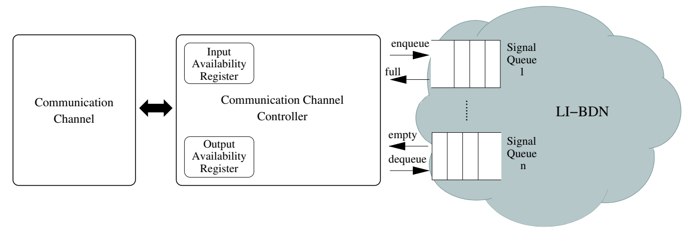

Fig. 2. NBMC Interface: Hardware side [9]

it is necessary to keep track of when signals are available, and this information must be communicated over the interface.

The software side of the NBMC interface implemented in SystemC [6] is shown in Figure 1. The hardware side is shown in Figure 2. Full details of how this interface operates are given in [9]. Here we highlight the means by which the interface propagates the availability of signals.

The interface propagates signal availability from hardware to software in a straight-forward fashion. There is a set of memory-mapped output availability registers in the hardware. As LI-BDN output (hardware-to-software) signal FIFOs become "not-empty", their corresponding availability bits are set. The availability registers are read by the software side of the interface as it reads signal data; the availability bits it has read are used to determine which signals should be updated in the software. The availability bits are cleared (and output FIFOs are dequeued) when the end of a simulated clock cycle is detected; the end of a cycle occurs when all of the input FIFOs have been enqueued, all of the output signals are available, and the hardware has serviced a read of the output signals in which all output signals were available.

The interface propagates signal availability from software to hardware through a set of memory-mapped input availability registers. The software writes to these registers as it is writing signal data. When the availability bit for a signal becomes set in the hardware, the data for the signal is enqueued in the corresponding LI-BDN input signal FIFO. Input availability bits are cleared when the end of a simulated clock cycle is detected.

The SystemC processes which remain in software require modifications to propagate signal availability. These modifications introduce additional simulation overhead. There are two modifications: 1) a "wrapper" is created for each process which looks at input signal availability and sets output signal availability accordingly; an output signal is marked available if all the inputs upon which it depends are available. 2) the signal update methods are modified to notify processes which are sensitive to the signal to run whenever the signal becomes available; the normal update method would only notify if the signal had changed value.

C. Performance

The execution time of a software-only structural simulator consists of two components: the time taken within processes and simulation framework overhead. Time taken within processes is the actual time needed to perform the simulated behavior. Framework overhead in SystemC stems from the need to manipulate lists of processes and events, to double

Platform	1 word Write Latency	1 word Write BW	1 word Read Latency	1 word Read BW	4 KB Write Latency	4 KB Write BW	4 KB Read Latency	4 KB Read BW	Bus to FPGA	Internal freq (MHz)
DRC 1000	0.295	13.5	2.82	1.42	11.6	354	18.2	225	Hypertransport	200
Pico M-505 (interrupts)	6.18	0.647	6.13	0.652	6.71	610	6.41	639	PCIe x8	250
Pico M-505 (polling)	2.85	1.40	3.08	1.30	4.12	994	6.94	590	PCIe x8	250
XUPV5 (pio-mode)	0.161	24.7	1.68	2.3	166	24.7	1750	2.3	PCIe x1	100
XUPV5 (dma-mode)	0.140	28.6	7.63	0.524	73.4	55.8	27.3	150	PCIe x1	100

TABLE I. LATENCIES (μS) AND BANDWIDTHS (MB/S) OF FPGA COSIMULATION PLATFORMS

buffer signal values, and to compare old vs. new signal values when deciding whether to schedule a process.

In a hybrid structural FAME simulator, the primary means by which speedup occurs is by reducing the time taken within processes by executing processes in hardware, thus allowing them to execute concurrently and removing instruction processing overhead. Framework overhead is also reduced for all signals and processes moved to the hardware. However, additional execution time is introduced by the communication between software and hardware. Communication generally requires operating system calls to device drivers, which then need to set up the communication with the hardware.

The available FPGA cosimulation platforms which we have observed are all designed primarily for a co-processor model of execution: the software sends a command to the hardware and blocks waiting for the hardware to complete. This emphasis leads to hardware platform and device driver design decisions which make it difficult, if not impossible to overlap communications. It can also lead to high latencies. Table I shows the measured latencies and bandwidth of reads and writes of a single word and a 4 KB block of data to three different FPGA cosimulation platforms.[1] All three systems block on I/O. Note that it is difficult to characterize what a typical simulator's data requirements might be, as they depend upon the simulator's partitioning; the data requirements of the simulators used in Section V vary from one word to 1 KByte per simulated cycle.

Blocking I/O coupled with long latency is of particular concern, as hybrid structural FAME simulators use fine-grained communications with the FPGA. Typically, there is at least one write to the FPGA and one read from the FPGA per simulated cycle. In such a simulator, the sum of write and read latency becomes the limiting factor, limiting simulation throughput to the reciprocal of the sum of the latencies. As an example, consider the DRC platform used in [9]. For a simulator transferring 4 KB of data in each direction each cycle, the best achievable simulation speed is 33.6 simulation KHz. Of course, more communications per cycle reduce simulation throughput further.

Therefore, reducing the number of communications per cycle is very important. Unfortunately, the NBMC interface creates excess communication. This happens for three reasons:

1) Read commands for some platforms (e.g. the Pico platform) are actually a FPGA write operation followed by a read operation. This behavior occurs because the hardware pushes data to software through a bandwidth-efficient stream interface. As the NBMC interface may poll the hardware multiple times and the hardware does not know how many times it will be polled, the hardware cannot know how much data to push into a stream queue without receiving some sort of command from the software.
2) A read command for input-to-software data happens on *every* simulation delta cycle until all data is read. In general, data will not all become available in a single delta cycle, particularly if the data coming from the hardware depends upon outputs to the hardware. For example, if hardware contains a simulated combinational block whose inputs come from simulated registers, its outputs cannot become available before two delta cycles after the clock edge.
3) Write commands happen on every delta cycle before which software-to-hardware signals have changed. If the hardware needs more than one software-to-hardware signal as input to produce an output and those signals do not change at the same delta cycle, multiple write commands will be used.

This excess communication leads us to look to scheduling as a means to reduce the amount of communication.

III. LI-BDNs, Models of Computation, and Scheduling

A. LI-BDNs

A latency-insensitive bounded dataflow network (LI-BDN) [11] is a dataflow network [12] whose nodes (called *primitive LI-BDNs*) are connected by bounded FIFOs of size ≥ 1 and which satisfies certain requirements (the No-Extraneous-Dependency and Self-Cleaning properties) upon when it enqueues and dequeues signals. These requirements ensure that there are no deadlocks. Vijayaraghavan and Arvind [11] showed that state machines which have an enable signal controlling state update can be turned into primitive LI-BDNs with "wrapper" circuitry which controls when outputs are enqueued and when state update is performed. Harris et al. [10] showed that the state machines of [11] can be used to emulate SystemC processes and demonstrated and implemented a systematic, automatable procedure for wrapping them in primitive LI-BDNs, thus providing a latency-insensitive design style for the NBMC interface of [9].

When LI-BDNs are used, each simulated signal takes on exactly one newly-computed value per clock cycle. At any given point in time, the availability of newly-computed values is inherent in the FIFO state. All primitive LI-BDNs must execute at some point during the simulated clock cycle in order to produce their output values; a primitive LI-BDN may execute multiple times if it has multiple outputs with different fan-in cones.

[1] The DRC 1000 and Pico M-505 are commercial platforms; the XUPV5 is a development platform with home-brewed software and firmware.

```
genCommShedule():
  perform HSR or acyclic scheduling to provide process and availability orders
  perform multiprocessor task scheduling to provide delta cycles
  annotate delta cycle of availability for each boundary-crossing signal
  if using delayed communication:
    compute schedule slack for each boundary-crossing signal
    foreach boundary-crossing signal:
      if the signal has slack:
        delay communication to the latest delta cycle in which there is already communication in that direction.
```

Fig. 3. Communication scheduling algorithm

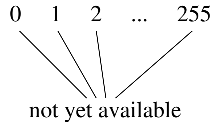

Fig. 4. Semi-lattice of the extended `unsigned char` type

B. Models of Computation

This notion of exactly one newly-computed value per signal per clock cycle with known availability of signals, and with processes that execute at least once, and possibly more times, is not directly compatible with the discrete event model of computation which SystemC uses. In SystemC a process does not execute at all if none of its inputs change value. It is also not usually possible to know when a signal's true new value for the cycle has been computed, because its driving process may execute later within the cycle with different input values.

However, this notion does in fact correspond to a special case of a known model of computation for software: the heterogeneous synchronous reactive (HSR) model of computation [13]. HSR systems consist of processes which compute monotonic functions of their inputs. By extending the datatype of each signal to a two-level semi-lattice by adding a $\perp$ element which is interpreted as "not yet available" as shown in Figure 4, by not setting output signals to available values until all input signals upon which they depend are available (thus guaranteeing monotonicity), and by resetting all signal values to $\perp$ at the start of each simulated cycle, an HSR system satisfies the requirements of an LI-BDN with FIFOs of depth one. In particular, monotonicity satisfies the LI-BDN No-Extraneous-Dependency requirement and resetting the signal values satisfies the LI-BDN Self-Cleaning requirement in a centralized fashion.

For this reason, the NBMC interface modifies the software side of the simulator as described in Section II-B, forcing processes to run and signals to pretend they have changed values. These modifications cause the discrete event SystemC model to behave as such an HSR system, which is what allows it to communicate correctly with the LI-BDNs in the hardware.

C. Scheduling

One important property of HSR systems explored in [13] is that they can be statically scheduled. Because LI-BDNs have an equivalent HSR representation, they can also be statically scheduled. This schedule can be represented both in terms of the order in which signals become available and in terms of the order in which processes should be invoked to produce those signals. In other words, we can know *a priori* a valid order in which to expect signals in an HSR or LI-BDN to become available. The only information needed to produce such a schedule are the data and control dependence relationships between signals. Details of the creation of an HSR schedule and heuristics for improving the schedule when there are processes with multiple output signals can be found in [13]. A similar technique called *acyclic scheduling* [14] has been previously used to schedule SystemC processes and improve simulation speed.

Note that correct static schedules are not unique, and may not directly correspond to the order in which the independently firing processes of LI-BDNs or dynamically scheduled processes of SystemC would actually fire. However, while signals may become available earlier than expected, they never become available later – if a process invocation is expected to produce a signal value, it will either have produced it on an earlier invocation or will produce it on that invocation. Any correct static schedule will provide us with a schedule for the availability of signals passing between hardware and software which will be valid.

IV. An Improved Implementation

With a static schedule of process invocations and signal availability in hand, we can proceed to implement communication scheduling.

A. Generating a communication schedule

The schedule provided by HSR scheduling is an ordered list of process invocations annotated with which signals are expected to become available after each process runs. This schedule is not quite what is needed for communication scheduling – it does not take into account the parallelism of the hardware or the speed of the hardware processes relative to communication latencies. Furthermore, communication will take place inside SystemC interface processes, and we need to place them in the schedule. Thus we introduce a communication scheduling algorithm, given in Figure 3.

The key observation here is that the actual order of invocation of the processes which remain in software is driven by signal availability. In SystemC, processes run whenever their inputs change value. This is not as haphazard as it may seem. In the first delta cycle, combinational processes dependent directly upon signals driven by edge-triggered processes may run. In the next delta cycle, combinational processes directly dependent upon signals driven by the previous set of process may run, and so forth. "May run" can be replaced by "must run" if the HSR-conversion technique previously described is used. Thus it becomes possible to annotate the process

invocation schedule with the delta cycles at which processes are expected to run based upon when input signals become available. This delta-cycle-annotated schedule can then tell us where communication can occur.

The delta-cycle-annotated schedule can be conveniently generated by performing multiprocessor task scheduling with precedence constraints and communication costs using an earliest-time-first heuristic [15]. The original HSR schedule items are used as the task graph nodes and the dependences between the signals being produced by each process in the schedule are used as the precedence edges. Time is measured in delta cycles and the following additional assumptions must be made:

1) There are an unlimited number of processors. This assumption ensures that processes are scheduled to run in the earliest delta cycle consistent with their dependences and the timing requirements to follow. It also makes the choice of tie-breaking heuristic in the priority function in the task scheduling irrelevant.
2) Software processes require one delta cycle to execute.
3) Hardware processes require zero delta cycles to execute. This assumption reflects the fact that the LI-BDNs are autonomous and do not operate upon centrally-coordinated delta cycles. The assumption also reflects the relative speed of computation to communication as seen in Table I; in the time it takes software to set up a request for data from the hardware, the hardware has had hundreds to thousands of FPGA clock cycles in which to execute processes.
4) Software to hardware communication has one delta cycle latency. As communication in an NBMC interface happens inside a SystemC process, the transfer process (which is not in the schedule) must run in the delta cycle after the signal becomes available.
5) Hardware to software communication has zero delta cycle latency. As above, because of relative speeds of FPGA computation and communication, we assume that the transfer process picks up available signals in the same delta cycle that they are computed.

After multiprocessor scheduling of the process invocations, their output signals which become available are annotated to become available at either the same delta cycle as their driver for hardware-to-software signals and the delta cycle afterward for software-to-hardware signals.

B. Delayed Communication Scheduling

At this point, the communication schedule can be used to explicitly perform transfers. However, it is still somewhat inefficient. Consider the case in Figure 5. In this figure, process 2 in hardware depends upon signals from both a clock-sensitive process and a combinational process. The communication and execution schedule would do the following:

Delta cycle 1: Execute process 1, transfer signal *A*.
Delta cycle 2: Transfer signal *B*.
Delta cycle 3: Execute process 2.

If signal *A*'s transfer is delayed until *B* becomes available, the number of individual communications can be reduced.

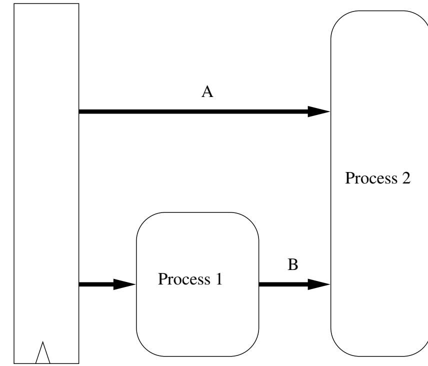

Fig. 5. Example of inefficient communication

```
launch():              // sensitive to clock edge
   clock_delta = sc_delta_cycle()
   transferEvent.notify(SC_ZERO_TIME)

transfer():            // sensitive to transferEvent
   switch (sc_delta_cycle() − clock_delta):
     case 1:
        foreach software−to−hardware signal s
                             scheduled for delta cycle 1:
           WriteDataTohardware(s.read())
        foreach hardware−to−software signal s
                             scheduled for delta cycle 1:
           s.write(ReadDataFromhardware())
        transferEvent.notify(SC_ZERO_TIME)
     case 2:
     ...
```

Fig. 6. Improved interface: Software side

An eager, earliest-time-first schedule reflects faithfully the signal availability order, but in general it will have excess communications when two signals which are used together become available at different times. When this occurs, one of the two signals can be said to have *slack* – it may arrive later without affecting the schedule length or violating signal dependencies.

Therefore, we can improve the schedule by delaying some signals. First we compute the amount of slack (the amount of delay possible) of each signal. Then we attempt to move the signal's transfer as late as possible within that slack, with the constraint that we will only move to a delta cycle if there is already a transfer in the same direction in that delta cycle.

C. Software side

The software side of the improved NBMC interface is shown in Figure 6. Only a single transfer process is required. This process first computes the current delta cycle's offset from the start of the current clock cycle. It then branches based upon the delta cycle to code which transfers boundary-crossing signals scheduled for that delta cycle. Finally, it notifies itself to run again in the next delta cycle, but only if there are more transfers to be done.

Note that there are no longer any signal availability flags; the interface knows which signals are available because it knows which delta cycle is executing. As an additional optimization, the interfaces orders both incoming and outgoing signals in the hardware address space in their transfer order, making it simple to transfer all data in a particular direction for a particular data cycle in a single communication API call.

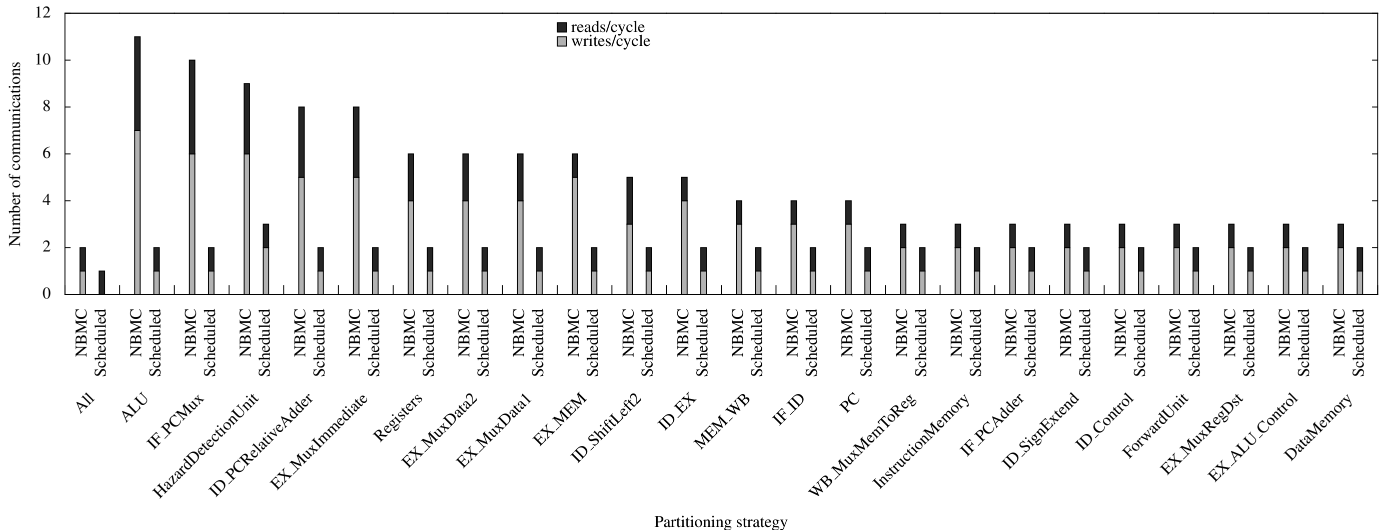

Fig. 7. Number of communications

Because there are no longer any availability flags to be propagated, the HSR conversion of SystemC processes is no longer necessary, and is not performed.

D. Hardware side

The hardware side of the improved NBMC interface is very similar to the original NBMC interface, with modifications because software no longer sets or reads availability flags and for streaming.

In the software to hardware direction, data is buffered much as it was before and is then enqueued into the LI-BDN when there is space available in the LI-BDN FIFOs. Because the software will write to each signal only once per cycle, there is no longer a need to ensure in hardware that only one write per cycle happens.

In the hardware to software direction, if the platform uses a stream interface, data transfer from hardware to software is pushed into the outgoing stream in the scheduled order as soon as it becomes available. LI-BDN FIFOs are dequeued when the last signal in the schedule has been pushed. Pushing removes a write transaction that was required to initiate reads in the case of streams.

V. Evaluation

We evaluate the effectiveness of communication scheduling by implementing it within the SPRI simulator synthesis infrastructure [5] and comparing the speed of simulators built with and without communication scheduling.

A. Experimental Setup

Our experimental platform is a Dell Precision T7610 with two Xeon E5-2609 quad-core processors running at 2.5 GHz with 16 GB of system memory and a Pico Computing Systems M-505 FPGA module as a coprocessor. This coprocessor module consists of a Xilinx XC7K325T-2 FPGA; it is connected to a a PCIe bridge and control FPGA to the system PCIe bus. The driver is set to poll for completion. VDHL-to-bitstream synthesis was done by ISE 14.7. The SystemC version is the reference implementation version 2.0.1.

Two different simulation models are used. The first is a Microlib fine-grain microarchitectural simulator modeling a five-stage, in-order pipeline implementing the DLX instruction set [14]. This simulator is composed of 23 module instances that describe registers and combinational logic; we test hybrid FAME simulators which place each instance individually into the hardware as well as one which places all the instances in the hardware. A simple infinite loop benchmark is run for 40 million cycles. From one word to several words of signal data are transferred per simulated cycle, with the amount depending on the partitioning.

The second simulation model is 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 [16] simulator organization: all instruction-set functional behavior is separated from the timing behavior. The partitioning we use keeps the functional behavior processes in software and synthesizes the timing behavior processes into hardware. The simulator is run to completion on the FFT kernel from the SPLASH-2 [17] benchmark suite with the `-m12` flag and `-p` flags matching the number of cores. LLVM 3.2 was used for compilation of the simulator. Each simulated core produces 4 words of signal data flowing from hardware to software and 13 words of signal data flowing from software to hardware per simulated cycle.

B. Scheduling and Delta Cycles

The Microlib model partitionings are useful for investigating the effectiveness of scheduling when input signals to a process are computed at different delta cycles. Each of the partitionings (except for two) moves a single SystemC process to the hardware. The minimal communication schedule for such a partitioning would have one write to and one read from the hardware in each simulated cycle, except for the `All` case, where there are no input signals to the hardware and no writes are required.

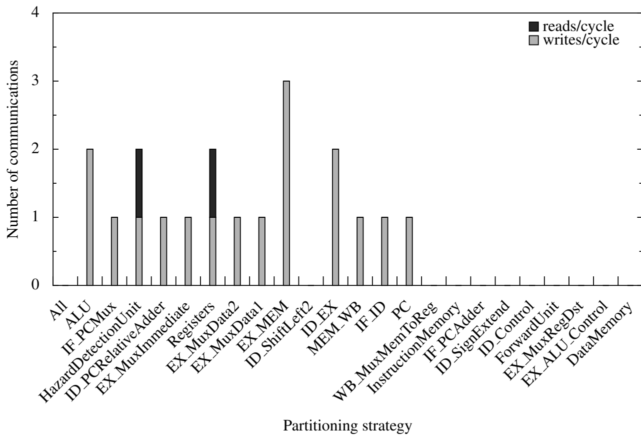

Fig. 8. Improvements due to delayed communication

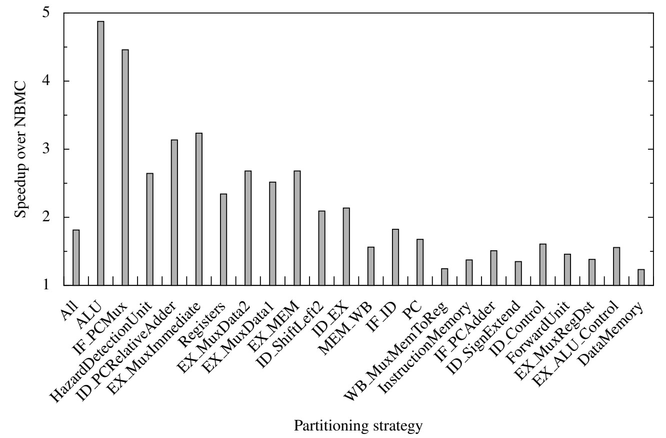

Fig. 9. Speedup of Scheduled interface vs. NBMC

Figure 7 shows the actual number of communications per cycle for each partitioning. The partitioning on the furthest left has all processes in hardware; the remainder are ordered by descending number of communications in the NBMC interface. In each case scheduling achieves the minimum number of communications, with one exception. In the `HazardDetectionUnit` case, there is an additional write which the communication scheduler cannot remove because the original HSR schedule invokes the moved-to-hardware process twice.

Figure 8 shows the effectiveness of delayed communication scheduling by showing the number of additional writes and reads which would have occurred if delayed scheduling had not taken place. In about half of the cases, delayed communication scheduling provides no benefit. Delayed scheduling does reduce writes when the hardware process models a stage register and sometimes when it models combinational logic: these reductions occur when input signals arrive at different times without delayed scheduling. It reduces reads when hardware processes appear multiple times in the HSR schedule, and in the case of `Registers`, when there are both combinational and sequential hardware processes producing hardware-to-software signals.

Figure 9 shows the effect of the reduction in communication: the speedup of the scheduled simulator with respect to the original NBMC interface. The speedup tracks the reduction in the number of communications quite closely ($r^2 = 0.98$).

C. Scaling with Model Size

The multicore model is useful for investigating the effect of model size on the effectiveness of scheduling. In this case, the style of the partitioning is the same as the model increases and the timing of process invocations with delta cycles does not change. The minimal communication schedule has one write to and one read from the hardware in each simulated cycle, but the amount of data written and read is proportional to the number of cores.

Figure 10 shows the speedup of the scheduled interface with respect to the original NBMC interface for different numbers of simulated cores. In each case, scheduling removes one write operation per simulated cycle because the Pico stream interface does not then require a write operation to command a read. Delayed communication is required to achieve the minimal schedule. Speedup is modest, consistent with a 1/3 reduction in communication costs, but increases slightly as the number of simulated cores increase. This increase in speedup can be explained by Figure 11, which shows the amount of user mode time spent per simulated cycle. As the number of simulated cores increases, and the number of signals and processes increase, the amount of software overhead saved by not doing HSR conversion increases, leading to increasing speedup.

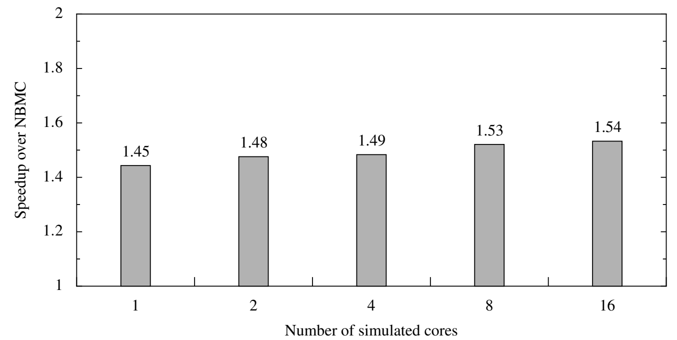

Fig. 10. Speedup of Scheduled interface vs. NBMC

VI. Conclusions

This paper has demonstrated that scheduling the communications between hardware and software can improve the per-

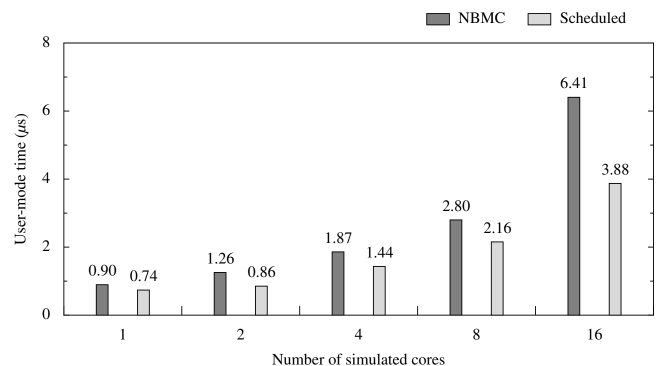

Fig. 11. Time spent in user mode (μs)

formance of an structural hybrid FAME simulator significantly, yielding a speedup of 1.54 over previous work for a 16-core processor model. The speedup is primarily due to a reduction in the number of communications and secondarily due to a reduction in software overhead. We have also introduced a simple scheduling improvement called delayed scheduling which reduces the number of communications required in a schedule. In most cases, this improvement achieves the minimum number of communications. As a result, synthesized structural hybrid FAME simulators will be faster, making hybrid FAME simulation more generally useful.

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] 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.

[8] Z. Ruan, K. Cahill, and D. A. Penry, "Elaboration-time synthesis of high-level language constructs in systemc-based microarchitectural simulators," in *Proceedings of the 2010 International Conference on Computer Design*, 2010.

[9] Z. Ruan and D. A. Penry, "Interface design for synthesized structural hybrid microarchitectural simulators," in *Proceedings of the 2012 International Conference on Computer Design*, 2010.

[10] 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.

[11] 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.

[12] G. Kahn, "The semantics of a simple language for parallel processing," in *Information Processing '74: Proceedings of the IFIP Congress*, 1974, pp. 471–475.

[13] S. A. Edwards, "The specification and execution of heterogeneous synchronous reactive systems," Ph.D. dissertation, University of California, Berkeley, 1997.

[14] D. Gracia Pérez, G. Mouchard, and O. Temam, "A new optimized implementation of the SystemC engine using acyclic scheduling," in *Proceedings of the Design Automation and Test in Europe Conference*, February 2004, pp. 552–557.

[15] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, April 1989.

[16] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 64–67, July 2009.

[17] 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.