Techniques for LI-BDN Synthesis for Hybrid Microarchitectural Simulation

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

***Abstract*—Computer designers rely upon near-cycle-accurate microarchitectural simulation to explore the design space of new systems. Unfortunately, such simulators are becoming increasingly slow as systems become more complex. Hybrid simulators which offload some of the simulation work onto FPGAs can increase the speed; however, such simulators must be automatically synthesized or the time to design them becomes prohibitive. Furthermore, FPGA implementations of simulators may require multiple FPGA clock cycles to implement behavior that takes place within one simulated clock cycle, making correct arbitrary composition of simulator components impossible and limiting the amount of hardware concurrency which can be achieved.**

Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) have been suggested as a means to permit composition of simulator components in FPGAs. However, previous work has required that LI-BDNs be created manually. This paper introduces techniques for automated synthesis of LI-BDNs from the processes of a System-C microarchitectural model. We demonstrate that LI-BDNs can be successfully synthesized. We also introduce a technique for reducing the overhead of LI-BDNs when the latency-insensitive property is unnecessary, resulting in up to a 60% reduction in FPGA resource requirements.

I. Introduction

Computer architects and designers rely upon simulation when they evaluate new ideas, explore the design space, and validate the behavior of a proposed system. Microarchitectural simulators are widely used to make near-cycle-accurate performance predictions. However, as processors have become more complex, microarchitectural simulators have become too slow to permit extensive exploration of complex future multicore systems. Simulation of a single multicore benchmark can require more than a week [1].

Researchers have proposed to use FPGAs to accelerate simulators [1], [2], [3], [4], [5]. These FPGA-based *hybrid simulators* contain a software portion and a hardware portion which communicate through an interface. Such hybrid simulators can provide two orders of magnitude of speedup [1], however, designing such simulators manually has proved to be time-consuming; as a result, it has been proposed [6] that hybrid simulators be synthesized from simulation models written in structural software simulation frameworks such as SystemC [7], Unisim [8], and the Liberty Simulation Environment (LSE) [9].

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

When synthesized simulator components communicate with each other, it is desirable to compose (internally connect) the components in hardware. Composition reduces communication across the hardware/software interface; frequent cross-interface communication has been shown to lead to slow simulators [10].

This desire conflicts with a fundamental limitation of FPGA implementations. This limitation arises because the FPGA must be used to model architectural constructs such as content-addressable memories and multi-ported array structures which are convenient to model using state machines and multiple clock cycles in the FPGA. However, in general, state machines which require multiple clock cycles are not composable.

One proposed solution to this dilemma is to place FIFOs between the state machines implementing individual components. Simulation time is then represented by counting enqueue and dequeue operations. This approach has been taken in [11] and [2] and simplified and formalized as the theory of Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) [12]. When LI-BDNs are used, the state machines in a simulation model communicate with each other through FIFOs. Each state machine is "wrapped" with logic for controlling these FIFOs and the state machines. As long as the wrapping and interconnection obey certain properties, the wrapped state machines may be composed.

Reference [12] describes a procedure to generate the wrappers which LI-BDNs require. However, this description assumes that one state transition of the state machine equals one clock cycle of simulation time and can be computed in a single FPGA clock cycle. An example is given in [12] of an LI-BDN which can take multiple FPGA cycles to model a single cycle of simulation time, but this example cannot be derived from the stated procedure because the clean abstraction of a wrapper around a state machine is lost.

Hybrid simulator synthesis tools will not always be able to generate state machines in which one state transition equals one simulation clock cycle because of the FPGA implementation limitations previously mentioned. Synthesis tools therefore require a new procedure to wrap such state machines into LI-BDNs. This work makes the following contributions:

1) A procedure for wrapping multi-cycle state machines modeling a single cycle of simulation time into LI-BDNs.
2) An implementation of this procedure within a hybrid

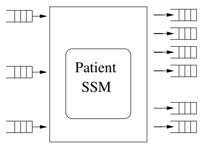


Fig. 1. A primitive BDN

simulator synthesis tool which can synthesize LI-BDNs from a System-C architectural model.

3) A simple technique which removes FIFOs from the synthesized LI-BDN when latency-insensitivity is not required, resulting in a savings of up to 60% of FPGA resources.

As a result of this work, hybrid simulator synthesizers will be able to provide composability in the FPGA implementations. The resulting hybrid simulators will enjoy less communication overhead and more concurrency, resulting in faster simulators and allowing designers to explore a greater portion of the design space, leading to improved designs.

II. Background

A. Latency-Insensitive Bounded Dataflow Networks

This section explains Latency-Insensitive Bounded Dataflow Networks and how Latency-Insensitive Bounded Dataflow Networks can be said to implement state machines. The formal definitions and proofs given in [12] are not repeated here; the reader is encouraged to consult them.

Bounded dataflow networks (BDNs) are dataflow networks [13] whose nodes are connected by bounded FIFOs of size ≥ 1. The individual nodes, called *primitive BDNs*, implement *patient* synchronous sequential machines (SSMs); patient merely means that there is a global enable signal controlling state update. A primitive BDN is shown in Figure 1. FIFOs can be enqueued only when they are not full and dequeued only when they are not empty. All FIFOs are empty to start with. A FIFO's output is connected to a single primitive BDN and its input is also connected to only a single primitive BDN. (Note that forks or fanout can be described as primitive BDNs themselves.)

Bounded dataflow networks are able to implement SSMs if the notion of time is changed from a "wall clock" measurement to a "sampling-period-based" measurement. Sampling periods in the SSM are represented by enqueue and dequeue operations on FIFOs of the BDN. In particular, a BDN is said to *implement* an SSM if and only if:

- There exists a bijective mapping between the outputs of the BDN and the outputs of the SSM and between the inputs of the BDN and the inputs of the SSM;
- the output histories of the SSM (i.e. the sequence of values which its outputs take at the end of each sampling period) and the output histories of the BDN (i.e. the sequence of values which are enqueued into its output FIFOs) match whenever the input histories match; and
- the BDN is deadlock-free.

This redefinition of time as enqueue/dequeue operations on FIFOs provides latency-insensitivity; the implementation of primitive BDNs can take any amount of FPGA cycles to execute, but the simulation time of the simulated SSM increments only when enqueues and dequeues are performed. Note also that there is no need for a global logical time nor global synchronization; individual primitive BDNs are decoupled and may slip time with respect to each other.

Arbitrary combinations of primitive BDNs may not be deadlock-free; however, if the primitive BDNs have two properties, deadlock may be prevented in many situations. These two properties force outputs to be produced and inputs to be consumed in a timely manner and are:

No Extraneous Dependency (NED)
An output value must eventually be produced if all the inputs to which it is combinationally-connected (i.e. all the inputs in its fan-in cone) are available. This property ensures that there are no deadlocks in which outputs are not enqueued because input FIFOs are empty in a cycle.[1]

Self-Cleaning (SC)
If all outputs for a logical timestep have been produced, all the inputs for the logical timestep must eventually be dequeued. This property ensures that there are no deadlocks in which no output can be enqueued because output FIFOs are full in a cycle.

When a primitive BDN possesses both the NED and the SC properties, it is known as a *primitive Latency-Insensitive BDN (primitive LI-BDN)*. Primitive LI-BDNs can be connected together (with FIFOs between them) to form LI-BDNs as long as their composition does not create a combinationally-connected cycle.

Reference [12] provides a simple procedure for wrapping an SSM into primitive LI-BDN which implements that SSM:

1) Transform the SSM to a patient SSM by adding enable signals to all state elements and ANDing existing internal enable signals with the external enable signal.
2) Add enqueue and dequeue signals for outputs and inputs, and maintain a "done" flag for each output, as shown in Figure 2. Enqueue occurs when an output FIFO is not empty, it has not already been enqueued in this logical timestep, and all the inputs upon which the output depends are available. All inputs are dequeued when all outputs have been enqueued and all inputs are available; the enable signal for the patient SSM is asserted in the same FPGA cycle that all dequeue signals are asserted to the input FIFOs.

Reference [12] does not formally prove that this procedure is correct, but it is easily observed that both properties are maintained: an output is enqueued whenever all its the combinationally-connected inputs are available (NED) and all the input queues are dequeued once they are all available and all of the outputs have been enqueued (SC). Furthermore, the enable signal prevents the state from changing until all inputs

[1]There is a technical condition that the property only needs to hold if all other output and input FIFOs have "caught up" to the previous simulated cycle.

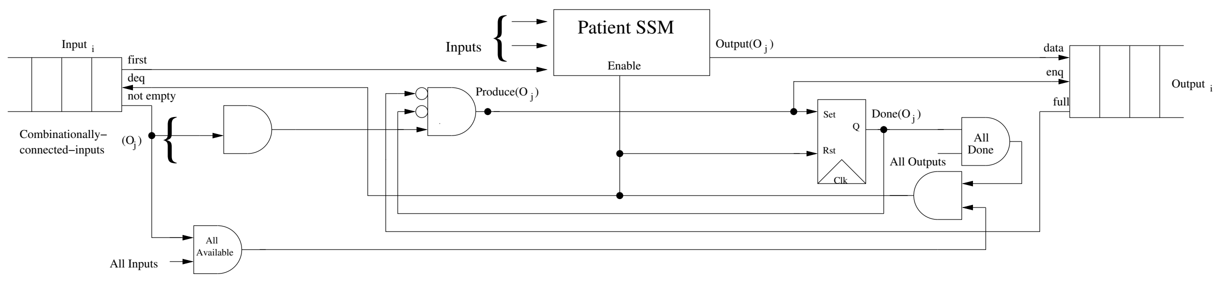


Fig. 2. LI-BDN wrapper from [12]

are available and outputs are created, allowing the output histories to match.

Limitations of the LI-BDN wrapping procedure

This procedure assumes that the SSM to be wrapped is an SSM whose behavior is to be modeled by the LI-BDN: one state transition of the SSM equals one cycle of logical time which becomes one set of enqueue and dequeue operations of the LI-BDN. Furthermore, the SSM's calculation of outputs and next state must take only a single FPGA cycle.

Synthesized hybrid simulator components are SSMs, however, these SSMs *simulate* a cycle of logical time. Multiple transitions of the SSM may be required to compute a single logical cycle. As a result, the procedure of [12] is not applicable. Reference [12] does go on to argue that an LI-BDN could take multiple FPGA cycles to compute its outputs; indeed, this is part of the argument for using LI-BDNs. However, no general procedure for forming such an LI-BDN is given. There is one example given of refining an LI-BDN into one which uses multiple FPGA cycles in computation, but this example is manually generated and loses the clean abstraction of a wrapper around a state machine; the state machine and the LI-BDN control state are fused into one state machine which is then refined for multi-cycle behavior.

B. SystemC

SystemC [7] models a design as a collection of functions known as processes which are interconnected by directed signals. When invoked, processes read their input signals and write to their output signals. The signal values form the *environment* in which processes run. SystemC processes have the following properties that are important to hardware synthesis:

Edge-triggered A process is fired, or run, by the SystemC framework when any event upon which it is waiting occurs. The process is said to be sensitive to these events, which are usually changes in input signal values.

Non-preemptive A process runs until it either returns or explicitly waits on an event in any firing of the process.

Output-inseparable Any firing of the process updates outputs based on the current values of the inputs and state. It is impossible to update only the outputs that are affected by the inputs that have changed.

Non-concurrent Processes may not execute concurrently unless the behavior appears identical to non-concurrent execution.

In addition to these process properties, signals must maintain delta-delay semantics; new signal values cannot be read in the same timestep in which they are written. In conjunction with non-preemptive execution and non-concurrency, the implication is that inputs to a process may not change while a process is firing, and outputs from a process do not change the environment until the process returns or waits.

SystemC processes may execute arbitrary code, however not all code nor design styles can be readily synthesized into hardware. Exactly what is considered synthesizable depends upon the capabilities of the synthesizer used. In this work, will assume that processes to be synthesized obey a simple set of rules similar to those supported by commercial vendors [14], [15] and proposed in the draft SystemC Synthesizable Subset standard [16]:[2]

- A process may only be sensitive to its inputs.
- A process may be either combinational or sequential. A combinational process must be sensitive to all of its inputs, and a sequential process may only be sensitive to the clock.
- A combinational process must produce all of its outputs whenever it is fired.
- A combinational process may not have internal state.
- A process may only alter its outputs and internal state and may not have side-effects.

C. SystemC Process Synthesis

Hybrid simulator synthesizers transform SystemC processes into FPGA hardware. We will call the generated hardware *FPGA-implemented Processes (FIPs)*. FIPs inherit the properties of SystemC processes. They may also require multiple cycles to execute because of structures or operations that cannot be synthesized as purely combinational elements. The environment of a FIP is the hardware between FIPs which maintains signal values or which communicates signal values to/from software.

The properties of a SystemC process, and hence a FIP, imply an interface like that shown in Figure 3(a). The signals are as follows:

[2]Processes remaining in software have no such restrictions.

Signal	Type	Function
Go	input	Signals to a FIP when to fire. Asserted when inputs have changed. Cannot be asserted while `Busy` is asserted.
Busy	output	Signals that the FIP is firing. In order to preserve the appearance of delta-cycle semantics for signals, the inputs may not change and new values produced by the FIP may not be seen by other FIPs while the FIP is busy. If only a single FPGA cycle is required to complete the firing, `Busy` is not asserted.
Input(I_i)	input	The data for a given input.
Output(O_j)	output	The data for a given output.
Write(O_j)	output	Signals to the environment that the corresponding output is being written in this FPGA cycle. The output signal is valid only while this signal is asserted.

The `Go` signal maintains the edge-triggered property and the `Busy` signal indicates when the FIP is finished, allowing non-preemptive behavior to be maintained. The environment must maintain the appearance of non-concurrency by ensuring that the inputs do not change.

FIPs do not need to provide state elements for outputs which are driven directly from state because in SystemC, this state is maintained by the signals in the environment. After synthesis, the environment (i.e., the logic outside of FIPs) retains this responsibility. Thus the output signals of FIPs derived from sequential processes are actually the "next state" values of those signals.

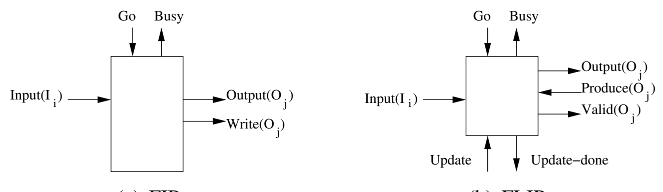
We will call FIPs derived from combinational processes *combinational FIPs* and FIPs derived from sequential processes *sequential FIPs*. This nomenclature does *not* imply that the FIP itself is implemented as purely combinational or sequential logic.

III. Composing FIPs

Composition is useful in a hybrid simulator because it eliminates round-trip communication from the host to the FPGA. FIPs cannot be directly composed because of the variable completion time for each FIP. Wrapping a FIP into a primitive LI-BDN can create a composable network.[3]

In order to achieve LI-BDN wrappers for FIPs, two things are necessary. First, FIPs must be transformed to be compatible with LI-BDNs in much the same way that SSMs need to be transformed into patient SSMs to be compatible with LI-BDNs. Second, appropriate control signals must be generated in the LI-BDN wrapper.

[3]Once LI-BDNs have been composed, delta-cycle accuracy of the simulator is not maintained. However, microarchitectural simulators generally do not require delta-cycle accuracy and neglecting it is a common optimization technique, e.g. [17].



(a) FIP (b) FLIP

Fig. 3. FIP and FLIP interfaces

A. LI-BDN-compatible FIPs

FIPs must be transformed before they can be wrapped into LI-BDNs. LI-BDNs require that every output signal be enqueued exactly once per simulated clock cycle, however FIPs do not guarantee this behavior. FIPs only react to events; combinational FIPs may fire multiple times per simulated clock cycle or even not fire at all if their inputs do not change. Furthermore, a combinational FIP may write a signal multiple times in a single firing. We call FIPs which have undergone transformation to become LI-BDN-compatible *FPGA-implemented LI-BDN-compatible Processes (FLIPs).*

Sequential FLIPs

Sequential FLIPs must produce state outputs for the *current* simulated clock cycle, not the next cycle. Thus the first step in sequential FLIP transformation is to change the outputs of a sequential FIP to reflect the current state instead of the next state. As a result, the FLIP must maintain the state of the output internally instead of relying on the environment.

The second step is to separate the production of output signals from simulated state update and from each other; output-inseparability must be overcome. Output signals may need to be produced at different times because of differing output FIFO availability. State update needs to be delayable until the logical clock cycle is finished, just as was required of patient SSMs.

Combinational FLIPs

The first step in combinational FLIP transformation is to separate the production of output signals from each other, just as was required for transformation of sequential FIPs. Output signals may need to be produced at different times because of differences in both output FIFO availability and input FIFO readiness. Note that combinational FIPs are not allowed to contain internal simulated state and thus do not need to delay state update.

The second step is to ensure that the outputs of the FLIP are only written once per firing. This is because an LI-BDN can only write one value per simulated clock cycle and does not know which value will be the final, correct one. If the FIP may write multiple times in a firing, then the FLIP must buffer the values to be written and only write the last such value through its interface.[4]

[4]It is possible for this problem to be resolved as part of the LI-BDN wrapper, but we feel it more convenient to include this logic as part of the FLIP.

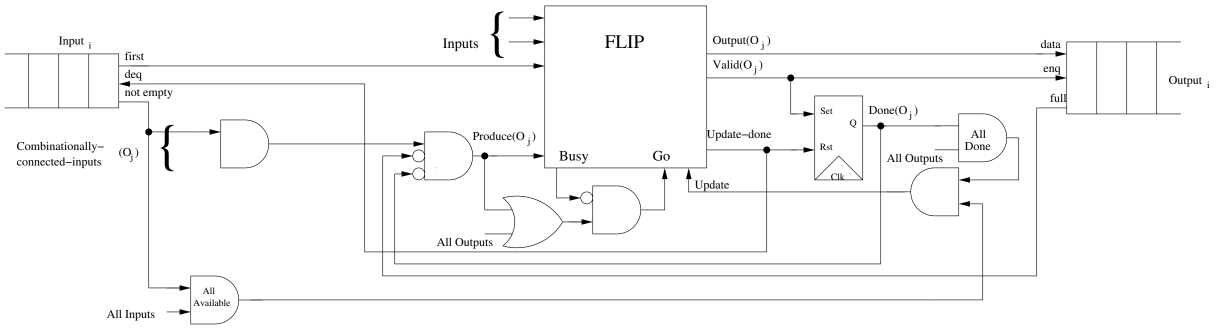


Fig. 4. An extension to [12]'s LI-BDN control circuit that supports FLIPs.

FLIP Interface

The above requirements imply an interface like that shown in Figure 3(b). The new signals are as follows:

Signal	Type	Function
Update	input	Signals the FLIP to update itself to the next state. Asserted when all outputs have been produced and all inputs are available and will remain asserted until `Update-done` is asserted. Only one update should be triggered per assertion.
Update-done	output	Signals that the FLIP has completed its transition to the next state. Must be held until `Update` is deasserted.
Produce(O_j)	input	Asserted simultaneously with `Go` to command that the corresponding output should be produced. Cannot be asserted when inputs required to compute the output are not yet available. The value is arbitrary when `Go` is not asserted.
Valid(O_j)	output	Asserted when the corresponding output is ready to be enqueued. May only be asserted if a produce command was issued when `Go` was asserted and may only asserted once per `Go` assertion. Replaces the Write(O_i) signal from the FIP interface. The given output signal is valid only while this signal is asserted.

The `Update`/`Update-done` signals form the analog of the enable signal of patient SSMs. A combinational FLIP simply ties `Update-done` to `Update`. For a sequential FLIP, the calculation of new state now is fired by `Update`/`Update-done` instead of `Go`/`Busy`; the latter signals are used solely for driving outputs.

The `Produce`/`Valid` signals are the key to overcoming output-inseparability. By requiring that `Valid` be asserted only when `Produce` was previously asserted at `Go`, the LI-BDN wrapper is given the ability to control the production of each output individually. Note that the FLIP may still compute the value of each output; indeed, output-inseparability implies that it must. However, outputs which are invalid for this firing of the FLIP are masked out (ignored) because the `Valid` signal will not be asserted.

Note that this interface forces the FLIP to be responsible for tracking which outputs have been masked and producing `Valid` accordingly. It is alternately possible to place this responsibility on the LI-BDN controller, and just allow the FLIP to assert `Valid` once per firing when it computes the corresponding output. The decision to make the FLIP contain the masking state was made to simplify the LI-BDN controller circuit with only a moderate change to the synthesis engine, to enable shorter block latency by allowing them to ignore operations that are masked and have long latency, and to allow easy reduction of LI-BDN resource requirements, as will be discussed in Section IV.

B. *LI-BDN Wrappers for FLIPs*

The LI-BDN wrapper which surrounds a FLIP to form a primitive LI-BDN must do five things: it must ensure that the NED property is maintained, that outputs are enqueued once and only when they are valid, that the FLIP is triggered until all outputs have been enqueued, that state is updated when a simulated clock cycle is finished, and that the SC property is maintained. Figure 4 shows the LI-BDN wrapper for a FLIP. The wrapper contains a `Done` flag for each output and combinational logic to generate all the control signals. As in [12], we do not provide formal proof of the wrapper's correctness, but instead give informal arguments.

The first two requirements are met by 1) asserting the `Produce` signal for an output only when the combinationally-connected inputs for the output are available, the output FIFO is not full, and the output has not been previously enqueued in this simulation cycle; and 2) connecting the `Valid` signal directly to the enqueue signal of the output FIFO. The conditions for asserting `Produce` and the FLIP's rules for asserting `Valid` imply that the `Valid` signal will only be asserted once when the output is to be enqueued.

Computing the combinationally-connected relation requires that the synthesis tool know which outputs depend upon which inputs. This knowledge can either be supplied by user annotation of the dependences or by analysis of the

SystemC process function.[5] Sequential processes have no combinationally-connected inputs for any output.

The triggering requirement is met by asserting `Go` whenever the FLIP is not busy and an output can be produced which has not yet been produced.

The state update requirement is met by asserting `Update` once all outputs have been enqueued and all inputs are available. The final (SC) requirement is maintained by dequeing all inputs and clearing all `Done` flags when the `Update-done` signal is completed.

Note that unlike SystemC processes, the inputs to a FLIP may change while it is firing because these changes do not affect the enqueued output values. For sequential FLIPs, outputs do not depend on inputs and state is not updated until all inputs are already available. For combinational FLIPs, outputs are ignored until all inputs needed for their computation are available; inputs which become available while the FLIP is firing do not cause outputs to become unmasked because the `Produce` signal is specifically considered valid only when `Go` is asserted.

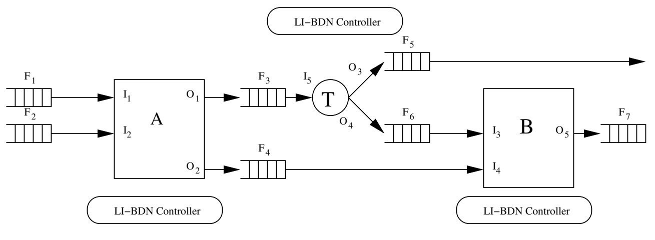
IV. FIFOless composition

LI-BDNs provide composability, but also require resources for FIFOs and wrapper logic. Resources might be saved if composition can be done without using FIFOs. Such composition would trade off FPGA resources with both clock cycle time and the ability for the primitive LI-BDNs to slip time relative to each other. In micro-architectural simulation the blocks are modeled after physical circuits, which leads to synthesized FLIPs with a manageable number of levels of logic. Additionally, the capability to slip time may not be very useful if the hybrid simulator must communicate between hardware and software in each simulated clock cycle. Therefore, the advantages to FIFOless composition make it extremely attractive for micro-architectural hybrid simulators.

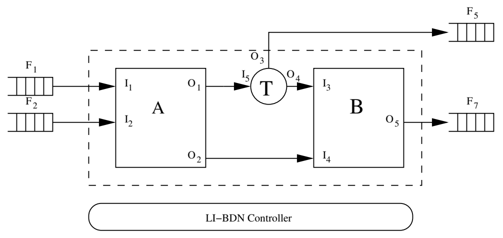
The proposed FLIP interface has been designed to allow FIFOless composition in many situations. In order to be composed, the primitive LI-BDNs must contain FLIPs which never assert `Busy` and whose outputs are always valid on the same FPGA cycle that `Go` and `Produce` are asserted. The FLIPs are connected together without FIFOs and a new LI-BDN wrapper is formed around them, creating a *composite primitive LI-BDN* which obeys all the properties of a primitive LI-BDN.

Consider Figure 5(a) which contains 3 primitive LI-BDNs A, B, and T. If the FLIPs inside these primitive LI-BDNs can be composed, then FIFOs 3, 4, and 6 may be eliminated and the controller circuits merged, resulting in Figure 5(b).

The following rules are used for forming the composite control signals in the LI-BDN wrapper:



(a) Three independent primitive LI-BDNs



(b) A composite primitive LI-BDN.

Fig. 5. FIFOless LI-BDN composition

- In the composite primitive LI-BDN, only FLIP outputs that go to FIFOs retain their `Done`(O_i) register. The removed `Done` flags are replaced with a logic high into the `All Done` AND gate and a logic low into the `Produce` AND gate. An internally-connected FLIP output is thus continuously produced while its inputs are ready, allowing this output to be available as the input to another FLIP.
- For any eliminated FIFO, all uses of the `not empty` signal are replaced with the `Valid` signal of the output that feeds that FIFO. All uses of the `full` signal are replaced with logic low.
- The `All Done` and `All Available` signals in the wrapper must be formed using all the remaining `Done` registers and all the remaining input FIFOs' `not empty` signals, respectively.
- The signal which dequeues all input FIFOs and clears all `Done` registers is the AND of all of the `Update-done` signals.

The NED property is maintained via simple signal connectivity without having to re-analyze combinational connectivity. If it were to be reanalyzed, the `Produce` signal for each output of the composite primitive LI-BDN would need to be asserted when the output FIFO is not full, the output has not already been enqueued in this simulation cycle, and all the inputs in the transitive closure of the combinationally-connected relation are available. However, consider the `Produce` signals for FLIP outputs which are internally connected. The outgoing FIFO has been eliminated and the `Produce` signal only computes whether all combinationally-connected inputs are available. If these inputs are in turn internally connected, then they are available if and only if their combinationally-connected inputs are available. Thus a simple inductive proof shows that the `Produce` and `Valid` signals for internal FLIP outputs are simply the ANDed availability of the transitive

[5]Hand-annotation can be tedious, as the user must think about the dependence relation, but the cost can be amortized when modules are reused. Analysis requires compiler techniques for dependence analysis which are beyond the scope of this paper; development of these techniques is a subject of ongoing investigation.

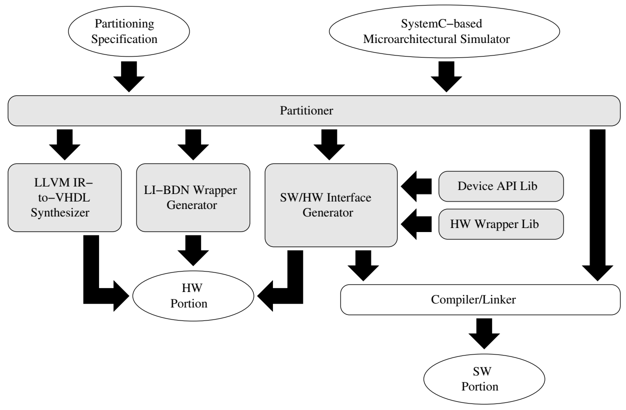

Partitioning Specification
SystemC−based Microarchitectural Simulator
Partitioner
LLVM IR−to−VHDL Synthesizer
LI−BDN Wrapper Generator
SW/HW Interface Generator
Device API Lib
HW Wrapper Lib
HW Portion
Compiler/Linker
SW Portion

Fig. 6. Modified SPRI synthesis flow for hybrid microarchitectural simulators

closure of the combinationally-connected relation. For the outputs of the composite primitive LI-BDN, the `Produce` signals are therefore asserted under the same conditions that a re-analysis would have derived.

Note that FIFOless composition can be applied to both combinational and sequential FLIPs. Indeed, sequential FLIPs which implement their internal state using flops are expected to always be composable because their only outputs are driven directly from the internal state. It is even possible, if there are no FLIPs requiring multiple cycles to produce an output, to reach FIFOless composition of all FLIPs. One benefit of the LI-BDN wrapper we have described is that it allows decisions about FIFOless composition to be made after FLIPs have been synthesized and does not require resynthesis after composition.

V. Evaluation

We demonstrate the new LI-BDN wrapping procedure by adding it to the SPRI hybrid simulator synthesis tool flow [10] and then synthesizing a hybrid simulator which uses FLIPs and LI-BDNs to implement SystemC processes. We then run the hybrid simulator and compare its running time to that of a software-only simulator. Note that it is not possible to compare results directly with those of [12], as that work only introduced a procedure without implementing or evaluating it. The current work represents the first attempt to actually create LI-BDNs in a simulator synthesis tool chain.

The original software-only simulator uses SystemC to model a 16-core chip multiprocessor. Each core is a simple five-stage in-order pipeline implementing the PowerPC instruction set. The cache hierarchy is extremely simple and there are no shared caches. The simulator uses a speculative functional-first organization [18]: a single SystemC module calls a functional simulator to simulate instruction-set behavior; this module then communicates information such as branch results, effective addresses, and register specifiers to other SystemC modules which compute the timing by modeling the hardware.

Figure 6 shows the modified SPRI synthesis tool flow. The SystemC model and a partitioning specification are the input to the flow. We used a partitioning specification which assigns the functional simulator module to software and the rest of the model to hardware. The LLVM IR-to-VHDL synthesizer produces FLIPs from SystemC processes. The LI-BDN wrapper generator produces primitive LI-BDNs that wrap the controller, FLIP, and FIFOs into a valid network.

We validated the synthesized hybrid simulator by running a multi-threaded benchmark – the FFT kernel from the SPLASH-2 benchmark suite [19] with arguments `-p16` – on the simulator and comparing both the program results and the number of simulated cycles with those reported by the software-only simulator.[6] Both simulators were run 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 fitted on the HyperTransport bus as a coprocessor.

The hybrid simulator achieves a simulation speed of 73.2 KHz while the software-only simulator achieves a speed of 14.7 KHz, for a speedup of 4.97. While this speedup is not particularly large, it is limited by the size and complexity of the model, which in turn limit the amount of computation which can be moved into hardware. As models become larger, there is often more parallelism available to be taken advantage of in the hardware. As they become more complex, the execution time of a software-implemented process usually grows more rapidly than the execution time of its hardware implementation.

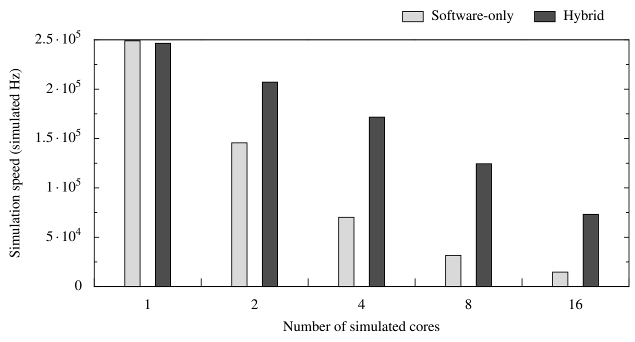


Fig. 7. Simulation Speeds

We demonstrate the effects of model size on speedup by creating a family of hybrid chip multiprocessor simulators modeling varying numbers of cores. Figure 7 shows the simulation speed achieved by these simulators when running the FFT benchmark. As the number of cores increases, the hybrid simulator slows down at a lesser rate than the software-only simulator, yielding higher speedups.

The primary bottleneck is the communication latency from hardware to software, which is quite high in the DRC 1000 system because all communication from hardware to software requires that the host driver provide a DMA read command to the FPGA and then poll the FPGA's status registers until the DMA completes. For the single-core hybrid simulator, communication occupies a staggering 80% of the execution time and no speedup is achieved. However, as the models become larger, the synthesized HW/SW interface code batches

[6]The cycle counts match to within 0.03%; an exact match is not possible because the benchmark output is expected to differ slightly from run to run.

communication whenever possible. Thus communication cost does not grow nearly as rapidly in this family of models as do either SystemC overhead or the aggregate execution time of the models' processes. For the 16-core hybrid simulator, communication is down to 45% of execution time. The net result is that as models become larger, the speedup increases.

Hardware capacity eventually limits the speedup of large models: in this case, a 32-core chip multiprocessor simulation does not fit within the available hardware. Platforms with multiple large FPGAs and/or better-organized communication (e.g., allowing FPGA-initiated transfers) will be necessary to achieve truly impressive speedups.

To evaluate FIFOless composition, we synthesized a version of the simulator where all possible FIFOs which could be removed were. The original simulator without FIFOless composition utilized 26,118 4-input LUTs and 26,062 slice flip flops. When FIFOless composition was added, the simulator utilized 9,429 4-input LUTs and 13,854 slice flip flops. This represents a 63.9% reduction of LUT resources, and 46.8% reduction in slice flip flops. FPGA clock cycle time was not affected, as the FGPA's critical path remained in the FPGA's interface with the HyperTransport bus.

VI. Conclusions

Computer architects and designers need fast near-cycle-accurate simulation to evaluate new ideas and guide their exploration of the design space of new systems. Synthesized hybrid simulation promises to produce such simulators without requiring excessive simulator design effort. However, FPGA implementations of simulator components require composability in order to achieve the best simulator performance.

We have demonstrated a procedure for forming LI-BDNs from the multi-cycle state machines for modeling a single simulation cycle which arise in hybrid simulators. We have demonstrated this procedure within a hybrid simulator synthesis framework. We have furthermore shown that a simple technique for composing LI-BDNs without intermediate FIFOs can reduce FPGA resource usage by 60%.

As a result of this work, hybrid simulator synthesizers will be able to provide both timing flexibility and composability in the FPGA implementations. The resulting hybrid simulators will enjoy less communication overhead and more concurrency, resulting in faster simulators and allowing designers to explore a greater portion of the design space, leading to improved designs.

VII. Availability and Acknowledgments

Source code for the simulator synthesis framework can be downloaded at http://bardd.ee.byu.edu.

The authors thank the anonymous reviewers for their helpful comments.

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, J. C. Hoe, and B. Falsafi, "ProtoFlex: Co-simulation for component-wise FPGA emulator development," in *Proceedings of the 2nd Annual Workshop on Architecture Research using FPGA Platforms*, 2006.

[4] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2008, pp. 1–10.

[5] 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.

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

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

[8] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, July–December 2007.

[9] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2002, pp. 271–282.

[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] M. Pellauer, M. Vijayaraghavan, M. A. Arvind, and J. Emer, "A-Ports: An efficient abstraction for cycle-accurate performance models on FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays '08*, 2008.

[12] 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.

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

[14] S. Sundralingram, "SystemC modeling, synthesis, and verification in CATAPULT C," Mentor Graphics, Tech. Rep., February 2010.

[15] "ESL design with the agility compiler for SystemC," Celoxica Software-compiled System Design, 2004. [Online]. Available: www-ti.informatik.uni-tuebingen.de/~systemc/.../Presentation-10-SF_3_sullivan.pdf

[16] *SystemC Synthesizable Subset 1.3 draft*. OSCI, 2009.

[17] 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.

[18] 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.

[19] 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.