An Automated Design Flow for FPGA-based Sequential Simulation

Pascal T. Wolkotte, Jochem H. Rutgers, Philip K.F. Hölzenspies,
Mark Westmijze, Remco Blumink, and Gerard J.M. Smit
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
P.T.Wolkotte@utwente.nl

***Abstract*—In this paper we describe the automated design flow that will transform and map a given homogeneous or heterogeneous hardware design into an FPGA that performs a cycle accurate simulation. The flow replaces the required manually performed transformation and can be embedded in existing standard synthesis flows. Compared to the earlier manually translated designs, this automated flow resulted in a reduced number of FPGA hardware resources and higher simulation frequencies. The implementation of the complete design flow is work in progress.**

I. Introduction

The development of large homogeneous and heterogeneous Multi-Processor System-on-Chip (MPSoC) platforms introduces various problems related to HW/SW co-design. The MPSoC architect studies all kinds of trade-offs, e.g. operation bit-widths, memory sizes, and performance parameters and bottlenecks, e.g. latency and throughput. Common practice is to perform extensive simulations of the MPSoC architecture before the system can be realized in silicon. In general the approach of simulating such large MPSoC designs is to either use (non cycle accurate) high level modelling or accept long simulation times with cycle accurate simulations.

For systems consisting of several tens or hundreds of tiles, cycle-true simulation leads to prohibitive simulation times from multiple hours to days. Despite these excessive simulation times, cycle and bit accurate tests are required to verify the design before manufacturing. Using the same concrete implementation (i.e. the same Hardware Description Language (HDL) source code) for simulation as well as synthesis we minimize the risk of errors in the design flow.

In earlier work we presented an sequential hardware-in-the-loop simulation approach to perform these bit and cycle-true simulations using a single FPGA [16, 17]. Using sequential evaluation of the parallel system makes it possible to simulate considerable larger design then would normally fit into any existing FPGA. Despite the sequential simulation, we are several orders of magnitude faster then an existing SystemC based cycle-true simulations. However, the design that is evaluated had to be manually adapted to fit into the simulation framework.

In this paper we describe the automated design flow that will modify and map a given homogeneous hardware design into an FPGA that performs the sequential simulation. The flow can be embedded in existing standard synthesis flows.

The rest of the paper is organized as follows. In section II we present other approaches to simulate hardware architectures. In section III we describe the general principle of sequential simulation framework and present an simple example. In section IV we introduce the design flow to generate the FPGA-based simulator and discuss its major steps. The automated state extraction, which is part of the design flow, is implemented and compared with the manual transformation in section V. In section VI we draw some conclusions.

II. Related Work

There are several methods to analyse large heterogeneous and homogeneous systems. A method is system level simulation such as SystemC [1] at different levels of abstraction. It can be used to describe systems from functional level to Register Transfer Level (RTL) level. The level of abstraction determines the speed of simulations. An example of SystemC simulation for Network-on-Chip (NoC) is the On-Chip Communication Network (OCCN) project, introduced by Coppola et al. [7]. Other examples are the MPARM simulator, which is a multi-processor cycle-accurate architectural simulator [3], or the framework presented by Kogel et al. [11]. The level of detail in the SystemC simulation tremendously influences the speed of simulation. Transaction Level Modelling (TLM), abstract data types and timed simulations showed almost a 3 orders of magnitude speed-up compared to RTL modelling [11]. However, optimizations to increase the simulation performance sacrifice the level of detail in the simulated results.

The general SystemC approach, which supports any design, can also be replaced by simulators dedicated for specific architectures or domains. For example, the OPNET modeler [14] is a domain specific simulator for network systems that has been used by Bolotin et al. to simulate the QNoC architecture [4].

Simulation performance can also be increased by using more processors in parallel or specialized hardware. For very large multiprocessor systems, an FPGA based emulation platform makes accurate and fast system simulation possible, as proposed in the RAMP project [2]. This approach requires multiple FPGA platforms as for example provided by the Zebu-XL system emulator [9] or the BEE2 [5] multi-FPGA board. Njoroge et al. [13] demonstrated the mapping of ATLAS, a chip-multiprocessor that prototypes the Transactional Coherence and Consistency (TCC) architecture, on the BEE2

platform. The ATLAS design requires 25-30% of the VirtexII-Pro Look-up table (LUT) resources and runs 100 times faster than the TCC simulator on a 2GHz PowerPC G5 system. The performance improvement between a SystemC implementation and hardware emulation framework in an FPGA is also demonstrated by Del Valle et al. [8]. With a single Virtex-II Pro FPGA, at 100 MHz, execution speed-ups of two to three orders of magnitude were measured in comparison with the software based implementation on a Pentium IV, at 3 GHz.

The PROTOFLEX hybrid simulator is another FPGA based simulation and emulation platform proposed by Chung et al. [6]. This simulator has two enabling techniques: 1) hybrid transplant simulation and 2) multiple-context emulation engines. The transplant option makes it possible to accelerate simple and/or frequently used operations in the FPGA (e.g. Arithmetic Logic Unit (ALU) operations). Complex and infrequent behaviours (e.g. Disk I/O) are simulated on the simulator host (i.e. Personal Computer (PC)). A small number of FPGAs host multiple-context emulation engines. Multiple processors are mapped onto the emulation engines. Simple time interleaving of the processors on the engines decouples the simulated system size from the required hardware resources of the FPGA host system. The latter technique is similar to the method described in this paper.

Several FPGA based implementations to validate NoCs are described in literature. Marescaux et al. [12] describe their implementation of interconnection networks on an FPGA. Genko et al. [10] propose a NoC emulation framework implemented on a Virtex-II FPGA. The emulation platform combines traffic generators, network interfaces, routers and traffic receptors. The platform is controlled by the FPGA's PowerPC and can work at 50 MHz, which is 2500 times faster compared to the SystemC simulator MPARM [10]. Genko's approach is bounded by the maximum available slices. For example, a 6 router network required 79% of a Virtex-II Pro VP20.

III. Sequential Simulation Approach

This section describes the simulation approach that is used to simulate large SoCs consisting of tens to hundreds of tiles on limited FPGA resources. The approach is developed to simulate a large number of clock cycles in a short time and study various design parameters and their effects without sacrificing accuracy or detail.

Simulation of a system at cycle-accurate and bit-level detail causes a continuous update of its exact internal state based on its inputs and current state. The state of a synchronous system is stored in registers and updated each clock cycle. The speed of a cycle-accurate simulator of a synchronous system is determined by the time it takes to update all registers of the system, i.e. how many cycles can be evaluated per time unit of the simulator platform.

The attempt to simulate a System-on-Chip (SoC) in an FPGA was inspired by the fact that an FPGA has parallel access to a lot of internal storage, which enables updating a large number of registers in a single FPGA clock cycle. In case the FPGA has enough resources to update all registers in a single cycle, the whole simulated design can be instantiated in the FPGA. However, the architecture designs of tomorrow will not fit in today's largest FPGA. Therefore, we generate a FPGA-based simulator which sequentially updates the whole state.

The sequential update of the design's state in multiple cycles, requires per cycle only a partial state vector and the corresponding combinatorial functionality to update this partial state. This method is based on the two level timing model that was introduced in Consensus Language (CONLAN) [15] and included in the simulation mechanism for Very-High-Speed Integrated Circuits Hardware Description Language (VHDL). If identical combinatorial functionality is present in multiple parts of the design, e.g. routers and processing tiles in a homogeneous SoC, the combinatorial resources can be re-used within the FPGA. Because only a small part of the state is required per cycle, the whole state can be stored in a large on-chip RAM. The combinatorial resource re-use and state storage in RAM overcomes the resource limitations within a single FPGA. The sequential simulator makes another trade-off between hardware resource requirements and simulation speed.

In the explanation of the methodology we use *system cycles* (i.e. real time) and *delta cycles* (i.e. computation steps). A delta cycle is defined as a clock cycle in the sequential simulator that evaluates one function but does not advance the simulation time. A system cycle is a clock cycle in the simulated parallel system and corresponds to a real clock cycle. A system cycle consists of multiple delta cycles.

A. Simulation Example

Any synchronous hardware design is represented by a set of synchronous elements that store its current state and a network of combinatorial elements that described the functionality of the design. Large hardware design, like SoCs, quite often have a repetitive structure in which large identical cells are interconnected in a certain topology, e.g. a mesh topology of routers. For the sequential simulator we partition the design at the level of these identical cells. In case of a NoC, for example, this cell is a router, which has reasonably large functionality.

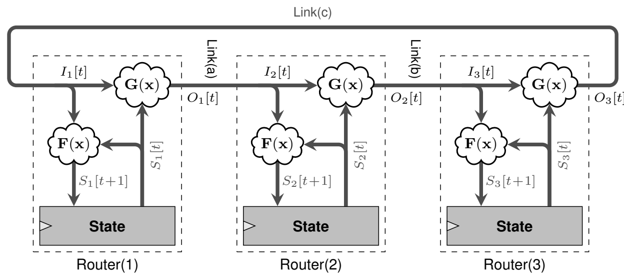
In this simulation example we consider a homogeneous architecture. However, the sequential simulation is not restricted to homogeneous designs. Heterogeneous designs are supported to by the instantiation of multiple unique cells with different combinatorial functionality.

In general, each cell has its local state and the cells interchange information via links (i.e. wires without registers). The cells outputs can directly change due to a change of the input signals, similar to a Mealy based state machine.

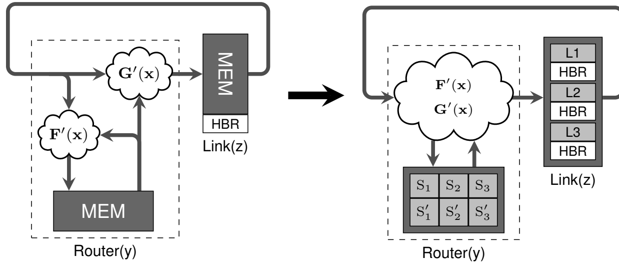
Figure 1a depicts a simple system that represents such an architecture. The figure depicts three identical cells interconnected via a ring topology. The functionality of a cell (c) is described by a set of functions, $F_c(x)$ and $G_c(x)$. These functions describe the cell's outputs signals ($O_c[t]$) and next *cell's internal state* ($S_c[t+1]$), and depend on the current cell's input signals ($I_c[t]$) and internal state ($S_c[t]$):

$$S_c[t+1] = F\left(I_c\left[t\right], S_c\left[t\right]\right) \tag{1}$$

$$O_c[t] = G\left(I_c\left[t\right], S_c\left[t\right]\right) \tag{2}$$



(a) Parallel representation



(b) Sequential representation

Fig. 1. System with combinatorial boundaries

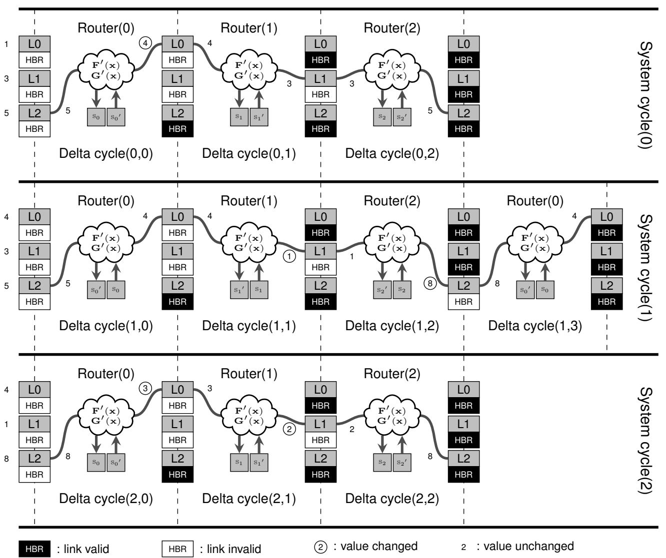


Fig. 2. Possible dynamic schedule

For the parallel design we can update the whole state ($S[t]$) by the applying the functions of $F_c(x)$ and $G_c(x)$ of all cells in the design. Due to resource constraints, not all functions can be instantiated in a single FPGA. However, in case of identical cells, we can re-use a single instantiation of the functions and sequentially update the whole state. This set of functions, $F_c(x)$ and $G_c(x)$ of a single cell, will be evaluated in parallel in the sequential simulation.

In figure 1a a combinatorial loop seems to be present in the homogeneous parallel design. However, this is due to the simplified graphical representation. For this approach, we assume that the original parallel system can be synthesized and does not contain a combinatorial loop that can cause oscillations. In general, a combinatorial loop will be detected during the synthesis of the parallel design.

The logic of figure 1a is changed to figure 1b, where all state registers are mapped into a single memory. Per delta cycle we load the current input signals and state of a single cell and update both its next state and output signals in parallel. A number of delta cycles are required to update the complete state. For all parts of the system a previously calculated state value and current input link value is used at the inputs of the combinatorial circuit to calculate the new state value and current output value. After all cells are evaluated and are stable, the new state becomes valid, which can be copied to the current state of the registers, and then a new system cycle can be started.

We cannot evaluate these cells in a fixed order. Any given sequence of cells will give problems, because there is always a link that is read before it is updated by the preceding cell. This is caused by the Mealy based design.

Changing the partitioning (e.g. merge part of $G_c(x)$ with either $F_c(x)$, or move it to the preceding or next cell) to create a Moore based design instead of a Mealy based design can be examined. However, modifying the design, as previously stated, is generally undesirable. Problems occur if and only if the update of the preceding cell changes the values of the link after the link is read.

For the links only the most up to date value is of interest. Therefore, the links have a separate memory, where every link has only a single memory position, in contrast to the state memory, where we store both current and new state. All signals of a link, which connects two cells, that have the same direction (e.g. from router A to router B), can be grouped into a single memory word on a single memory position.

Each memory position is tagged with an status bit (HBR). This bit indicates whether the last written value *has-been-read* from this link. Per cell we group all has-been-read (HBR) bits that correspond with the links that are connected to its input ports. If not all of these bits are valid, the cell is considered to be *non-stable* and has to be evaluated. A scheduler, for example round-robin, can decide which non-stable cell to evaluate. If all cells are stable, the design is considered to be completely evaluated and ready for the next system cycle.

An example of three system cycles is given in figure 2. Every system cycle is started by resetting all HBR bits to invalid, which makes all cells non-stable. Because all status bits are reset to zero at the start of a system cycle, it is guaranteed that all cells are evaluated at least once. This is necessary as a cell might change its outputs independent of its inputs. After a cell is evaluated all links that are connected to its input port will make their status bit valid. Furthermore, if the cell writes a value to a link, which is not equal to the current value in the memory, it will invalidate this link's status bit. The cell that has this link as an input will become non-stable and has to be (re-)evaluated. In figure 2 all values that are different from the current value are highlighted. This

happens in delta cycle (1,1), (1,2), (2,0) and (2,1). In case of delta cycle (1,2) link(2) is updated, but had already been evaluated in delta cycle (1,0). In this specific case the HBR-bit changes from valid to invalid and router 0 has to be re-evaluated. In all other cases, the updates of the links do not result in extra evaluation cycles as the HBR-bit was still invalid and routers had to be evaluated anyway.

IV. Design Flow

For a traditional Hardware-In-the-Loop (HIL) simulation of a design, described in any HDL, it is analysed, synthesized to a specific FPGA technology and finally placed and routed onto the targeted FPGA. However, a lot of architecture designs will not fit in today's largest FPGA.

As described in section III-A, we use the repetitive structure of many-core architectures. By mapping functional identical parts of the design onto the same FPGA hardware, we reduce this resource requirement. Instead of the instantiation of the whole architecture in parallel in the FPGA, the individual cells are evaluated sequentially. We use the general term cell to denote any distinct design element, e.g. a single processor core or a router.

To automatically generate the FPGA based simulator we have to transform the designer's architecture. We *partition* the design by identifying the various cells in the design. We assume each pair of cells to be highly similar or completely different. Therefore, we group cells that are highly similar and for each group of cells we create a single *hypercell*. Small differences between cells within a group are preserved in the hypercell. Per hypercell we *extract the state*, such that the combinatorial functionality can be re-used on a delta cycle basis within the FPGA.

In the designer's architecture, the identified cells are interconnected by links. This interconnection of the individual cells is described by the architecture's *topology*. This topology is used to *generate* the simulator, which consists of the link memory, state memory and a central controller. The simulator and transformed hypercells are combined, synthesized, and placed and routed to the targeted FPGA.

Figure 3 depicts the modified design flow for an arbitrary hardware design that requires fast cycle-accurate simulation. In the next subsections we describe the individual parts of the design flow.

A. Netlist representation

In the intermediate descriptions during a normal synthesis, either before or after the technology mapping, the design is described as a *netlist*. This netlist describes the functionality of the design by a interconnection of primitive components. For the sequential simulation we use one of the two possible intermediate descriptions. We convert the netlist description into a graph, which is used by the tools to apply transformations, partitioning and other operations.

Every hardware technology, ASIC and FPGA, has a library with standard cells or *primitives*. An AND, OR, LUT, and D-FF are examples of such primitives. Primitives can be instantiated.

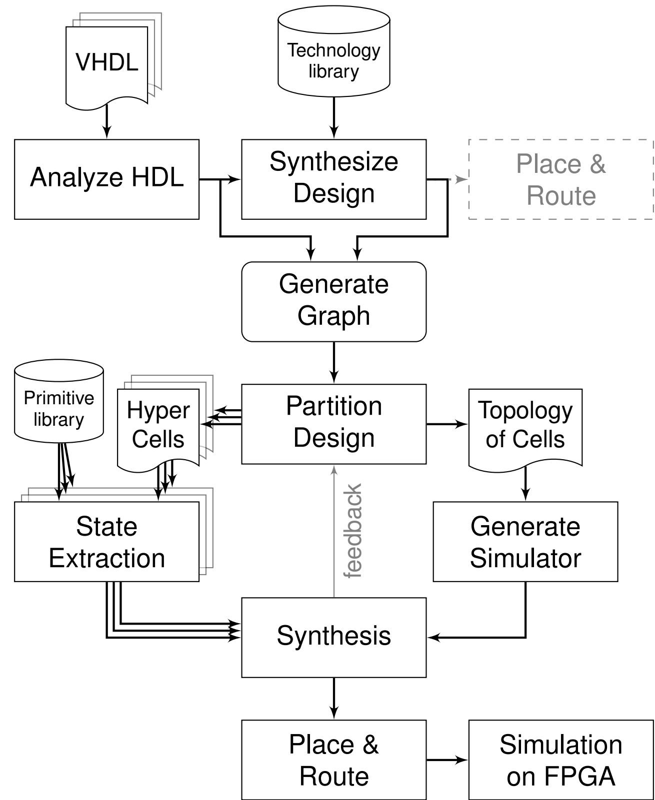


Fig. 3. Simulator's Design Flow

We represent each primitive by a small graph in which the primitive's function and each port is represented by a vertex. Each vertex of a primitive has a unique label.

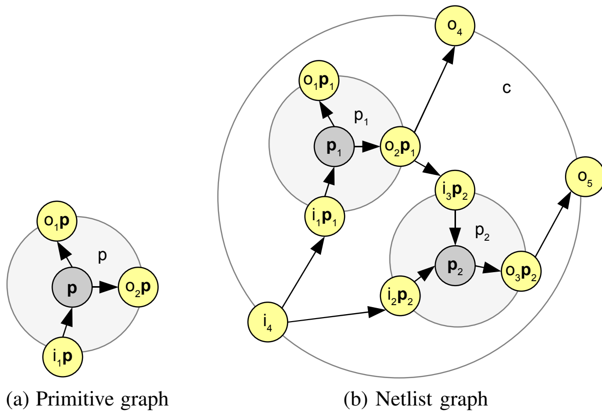
The netlist is the hierarchical description of cells interconnected by wires. The instance of a primitive in a netlist is an example of a cell. Larger cells consist of multiple instantiated primitives and other cells. Each cell in the netlist is represented by a netlist graph in which vertices are uniquely described by their label and their hierarchical position (denoted with a sequence of numbers) in the corresponding netlist.

Thus, the graph of an instance of a primitive extends its primitive's graph with such a hierarchy annotation on every vertex. The first number of this annotation corresponds to the cell number on the top level of the netlist, the second corresponds to the cell number within the top level cell, etc.

Figure 4 visualizes a graph representing a primitive and depicts a netlist graph with two primitives. The conversion of a technology primitive into a graph results in multiple vertices. The graph representation of a netlist is therefore large. For example, a NoC router, such as used in [17], is built of about 15k Xilinx Virtex-II FPGA primitives. Its equivalent graph has 80k vertices and 114k edges.

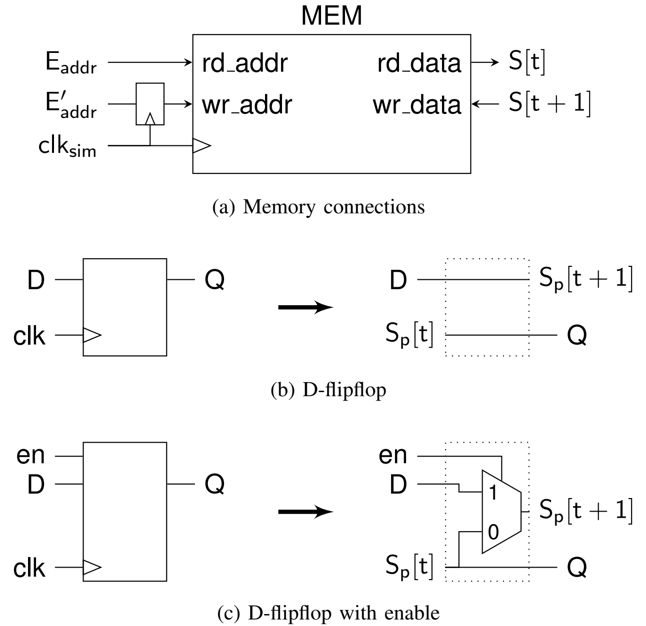
B. Partitioning of the design

After the conversion to the graph representation we apply transformations to the graph such that a sequential simulator can be generated. The first step in this transformation is a



(a) Primitive graph (b) Netlist graph

Fig. 4. Graphs representing hardware



(a) Memory connections

(b) D-flipflop

(c) D-flipflop with enable

Fig. 5. Examples of extraction of synchronous elements

partitioning of the graph. In the partitioning process we have to identify the identical cells within the graph. As described in section III-A we sequentially simulate large cells of the design. The cell can be of any size, for example, a single processor, a router, or the processor's ALU. Large cells will update a large part of the state vector at once. Small cells increase the utilization due to the event driven simulation, but will require more simulator overhead due to the increase in inter-cell communication.

As all cells in a homogeneous many-core architecture are identical, we only have to instantiate a single core's, i.e. largest replicated cell, combinatorial functionality in the FPGA. In heterogeneous multi-core designs, the individual cells are different. The difference can range from highly similar to completely different. For example, a NoC with routers that only have a different local address or a SoC with various types of processing cores.

A simple approach is the instantiation of a transformed cell for each variation of a cell in design and select the right cell on a delta-cycle basis. However, this might result in a large resource requirement and a low utilization of all the parallel available cells. Therefore, we group cells that are highly similar and for each group of cells we create a single *hypercell*. Small differences between cells within a group are preserved in the hypercell. This is done by adding multiplexers that allow enabling of the correct parts during simulation on a per delta cycle basis.

A group of cells is only merged into a single hypercell, if the overhead of the added multiplexers does not outweigh the resource reduction by instantiating the similar parts only once. This will reduce the resource requirements and increase the utilization of this new hypercell.

C. State Extraction

As depicted in figure 1b we separate the combinatorial functionality from the synchronous elements within the netlist graph. This extraction of the synchronous elements is performed on per hypercell basis. Each instantiate synchronous primitive within an hypercell, e.g. D-flipflop, is replaced by its original ports and in addition an old state input port ($S_p[t]$), an new state output port ($S_p[t{+}1]$), and some combinatorial logic. This combinatorial logic is required to selectively change the new state, such that the functionality of synchronous element is emulated. For example, a D-flipflop with enable only updates its state if the enable port is active.

Using such control logic will virtually enable the hypercell to swap states of multiple entities in the original design. That is, the state is not stored within the hypercell itself, but an entity's old state can be read by the hypercell and the updated state of this entity can be read from the hypercell.

All the state signals of a hypercell are grouped into two large state vectors that represent the old and new state of an hypercell. The two added state vectors are connected to data ports of the simulator's state memory (see figure 5a). The state of a specific entity E, stored in the state memory, will be updated by: 1) offering two addresses (E_{addr} and E'_{addr}) to the state memory, which respectively read and write the entity's state, and 2) connect the state memory's data ports to the entity's corresponding hypercell. Those addresses and interconnection are controlled by the simulator's scheduler.

If we assume only rising edge sensitive synchronous elements and a rising edge sensitive state memory, any D-flipflop can be replaced by the interconnection of the D-flipflop's ports with the added external state interface port as depicted in figure 5b. More advanced synchronous elements, e.g. D-flipflop with enable or reset, will have to be replaced by additional combinatorial functionality. An example is depicted in figure 5c.

This extraction is automated by transformation of the hypercell's netlist graph. For all synchronous primitives in the technology library used, we have an corresponding combinatorial replacement primitive. The algorithm will search for all instantiation of the synchronous primitives (using the labelling function) and replace the primitive's graph by its corresponding combinatorial replacement primitive. For the

extraction we include a parameter for the edge sensitivity of an entity. The combinatorial replacement primitive is different for only rising edge sensitivity or both rising and falling edge sensitivity.

D. Generation of the Simulator

After the partitioning of the hardware design into multiple unique hypercells and a topology description of the original cells we are able to generate a simulator. The simulator consists of:

- one or multiple instantiations of each unique hypercell;
- a large state memory with both the current and new state of the system;
- a link memory that stores the most up-to-date value of the links between cells;
- two interconnection networks that respectively interconnect:
  - the state ports of the instantiated hypercells with the state memory,
  - the input and output link ports of the instantiated hypercells via the link memory;
- a scheduler and controller.

The scheduler determines per delta cycle which original cell's state is updated by an hypercell. The schedule is dynamically updated using the stable status of each cell in the current system cycle. The controller generates the right addresses for the state and link memories and controls the interconnection network, such that the determined schedule is executed.

The instantiated hypercells and state memory will be generated by the state extraction block and all other components are primarily generated using the topology description of the original parallel design. All blocks can be combined into a single design, which is synthesized to the targeted FPGA.

E. Feedback

We want to optimize the performance of the simulator. The selection of the groups of cells, which create the hypercells, will have a major influence on the performance. Large hypercells will update a large part of the state vector, however they require more resources. Therefore, we include a feedback between the synthesis and partitioning of the design, such that the partitioning can be tuned. With the feedback we can for example generate small hypercells that can be instantiate multiple times in the simulator or large hypercells that are instantiated once.

V. Results

The design flow as presented in the previous section is work in progress. In earlier work, we manually transformed a packet switched NoC architecture, such that it fits within the sequential simulation framework [17]. The hypercell consisted of a single router and a stimuli interface, which enabled the injection of traffic into the network. The modified router was used to evaluate the performance of the NoC.

In this section we present some early results of the automated state extraction process. For this extraction we use the original VHDL sources of the packet switched router. This router is initially synthesized using the Virtex-II technology library. The non-transformed router's resource requirements is listed in table I.

TABLE I
ROUTER'S RESOURCE REQUIREMENTS

Design	Function Generators	Dffs or Latches	BlockRAMs
Non-transformed	3734	1732	0
Automated transformed	10660	0	0

TABLE II
SIMULATOR'S RESOURCE REQUIREMENTS

Design	Function Generators	Dffs or Latches	BlockRAMs
Simulator w.o. hypercell	1050	301	69
Automatically generated simulator	8267	300	69
Manually created simulator	7152	1275	71

The synthesized router's netlist is described as a graph and a single router is partitioned as a hypercell. The state of the hypercell is extracted and replaced by combinatorial replacement primitives with both rising and falling edge sensitivity. The bottom of table I lists the resource requirements of this transformed router. Compared to the original router, all flip-flops (Dffs) are gone and replaced by function generators. On average, a flip-flop is replaced by almost 4 multiplexers, which are individually mapped on 4 function generators. This can be optimized somewhat by combining multiple multiplexers into one function generator, which is done during the synthesis of the whole simulator.

The simulator itself is synthesized apart from the router's hypercell. The simulator instantiates a black box hypercell design. In this design, all state and link memories, interconnection logic and control, the scheduler, overall control unit, interfaces, pipeline overhead, etc. are included. This simulator is capable of simulating 64 instances. The resource usage of this simulator framework, i.e. simulator without hypercell, is presented in table II.

The last step is the combination of both hypercell and simulator framework, such that system is optimized for placement and routing. The resulting resource usage of this full design is also shown in table II. The usage of the function generators is significantly less than the sum of the function generators of the non-optimized transformed hypercell table I and the simulator framework after synthesis. During the mapping phase, multiple multiplexers are combined in one function generator.

Timing analysis shows that the system has a maximum operational clock frequency of 23.5 MHz. This is almost twice the operational frequency of 13.2 MHz of the manually created simulator.

In the manual transformation, there is no clean separation between the router entity and the simulator framework. It would be interesting to compare the change in resources after

transformation, but this is not possible. For example, the router entity includes the state memory already. The required resources for the manual transformation of the entity, including the link and state memories and the interconnection logic and control, excluding the scheduler and overall control unit, are shown in the bottom line of table II. As can be seen, the manually transformed simulator, even without the scheduler and control, generates more flip-flops and block RAMs than the automated one. The size of the scheduler and overall control unit in the manually created simulator can not be exactly determined, but their size is in the order of 500-1000 function generators and a few 100 flip-flops. This makes the automatically generated simulator equivalent in the resources used.

VI. Conclusion

In this paper we presented the design flow to automatically generate an FPGA based sequential hardware-in-the-loop simulator to simulate large multi-core architectures. Where in previous work the architecture to be simulated was manually transformed, the already developed tools of the design flow enable automated transformation of the design. The tool that automatically extracts the hypercell's state, such that the risk of errors during the transformation is minimized.

The size of the resulting automatically generated simulator is almost equivalent to the manual transformation. The automated flow generates a less block RAMs and flip-flops, but require some additional function generators. The development of the complete design flow is work in progress.

Acknowledgment

This research is conducted within the FP7 Cutting edge Reconfigurable ICs for Stream Processing (CRISP) project (ICT-215881) supported by the European Commission.

References

[1] "Open systemc iniative osci, systemc documentation," 2004. [Online]. Available: http://www.systemc.org

[2] Arvind, K. Asanovic, D. Chiou, H. a. . Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrynek, "ramp: research accelerator for multiple processors - community vision for a shared experimental parallel hw/sw platform," MIT, Tech. Rep., 2005.

[3] L. Benini, D. Bertozi, A. Bogliolo, F. Menichelli, and M. Olivieri, "Mparm: Exploring the multi-processor soc design space with systemc," *Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, September 2005.

[4] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "Qnoc: Qos architecture and design process for network on chip," *Journal of Systems Architecture*, vol. 50, no. 2–3, pp. 105–128, 2004.

[5] C. Chang, J. Wawrynek, and R. W. Brodersen, "bee2: A high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 114–125, March-April 2005.

[6] E. S. Chung, E. Nurvitadhi, J. C. Hoe, Falsaf, and K. Mai, "Protoflex: fpga-accelerated hybrid functional simulation," Computer Architecture Lab at Carnegie Mellon (CALCM), Tech. Rep. 2007-2, February 2007.

[7] M. Coppola, S. Curaba, M. D. Grammatikakis, Locatell, G. Maruccia, and F. Papariello, "occn: A noc modeling framework for design exploration," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 50, no. 2–3, pp. 129–163, 2004.

[8] P. G. Del Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini, and G. De Micheli, "A complete multi-processor system-on-chip fpga-based emulation framework," in *IFIP International Conference on Very Large Scale Integration*, October 2006, pp. 140–145.

[9] Emulation and Verification Engineering, "Zebu-xl system emulator," 2007. [Online]. Available: http://www.eve-team.com

[10] N. Genko, D. Atienza, G. De Micheli, J. M. Mendias, R. Hermida, and F. Catthoor, "A complete network-on-chip emulation framework," in *Conference on Design, Automation and Test in Europe*. Washington, Washington, D.C.: IEEE Computer Society, March 2005, pp. 246–251.

[11] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens, "A modular simulation framework for architectural exploration of on-chip interconnection networks," in *International Conference on Hardware/Software Codesign and System Synthesis*. New York, New York: ACM Press, 2003, pp. 7–12.

[12] T. Marescaux, J.-Y. Mignolet, A. Bartic, Moffa, D. Verkest, S. Vernalde, and R. Lauwereins, "Networks on chip as hardware components of an os for reconfigurabl systems," in *Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, vol. 2778/2003. Springer Berlin / Heidelberg, 2003, pp. 595–605.

[13] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "atlas: A chip-multiprocessor with transactional memory support," in *Conference on Design, Automation and Test in Europe*. Washington, Washington, D.C.: IEEE Computer Society, 2007, pp. 3–8.

[14] OPNET, "Opnet modeler," 2008. [Online]. Available: http://www.opnet.com

[15] R. Piloty and D. Borrione, "The conlan project: Status and future plans," in *Design Automation Conference*. New York, New York: ACM, June 1982, pp. 202–212.

[16] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit, "Using an fpga for fast bit accurate soc simulation," in *IEEE International Parallel and Distributed Processing Symposium, Reconfigurable Architecture Workshop*. Los Alamitos, CA, USA: IEEE Computer Society, March 2007, p. 167.

[17] ——, "Fast, accurate and detailed noc simulations," in *ACM/IEEE International Symposium on Networks-on-Chip*, P. Kellenberger, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, May 2007, pp. 323–332.