Partitioning and Synthesis for Hybrid Architecture Simulators

Zhuo Ruan
Department of Electrical and Computer Engineering
Brigham Young University
Provo, Utah 84602, USA
Email: zhuo_ruan@hotmail.com

David A. Penry
Department of Electrical and Computer Engineering
Brigham Young University
Provo, Utah 84602, USA
Email: dpenry@ee.byu.edu

***Abstract*— Pure software simulators are too slow to simulate modern complex computer architectures and systems. Hybrid software/hardware simulators have been proposed to accelerate architecture simulation. However, the design of the hardware portions and hardware/software interface of the simulator is time-consuming, making it difficult to modify and improve these simulators. We here describe the Simulation Partitioning Research Infrastructure (SPRI), an infrastructure which partitions the software architectural model under user guidance and automatically synthesizes hybrid simulators. We also present a case study using SPRI to investigate the performance limitations and bottlenecks of the generated hybrid simulators.**

I. Introduction

Computer architects use simulators to evaluate modern complex architectures. Architecture simulators are traditionally implemented in software. Unfortunately, software is too slow to simulate modern multi-core systems; a simulation run of a significant benchmark may require weeks or more of simulation time. Thus, hybrid simulation [6] has been proposed as an efficient method for accelerating simulation with no need to prototype the whole system. A hybrid simulator is not a pure software simulator and is composed of a software (SW) portion, a hardware (HW) portion, and an interface. However, how to partition a software simulator has not been perfected and the time-consuming procedure of hardware design prevents the exploration of different partitionings. Thus, we proposed the Simulation Partitioning Research Infrastructure (SPRI) in [7] to provide the ability to explore partitionings by automating the generation of hybrid simulators. In this paper, we will discuss the development of SPRI, future extensions, and a case study on generating multiple hybrid simulators from a software architecture simulator using SPRI.

II. Background

An important element of designing a hybrid simulator is deciding what portion should be in SW and what portion in HW and then implementing this decision. We call this process *Partitioning*. Previous work has manually designed the SW portions and the HW portions of their hybrid simulators, e.g. [1], [5].

Generally, designing a hybrid architecture simulator is a SW/HW codesign problem involving SW partitioning, interface generation, and HW synthesis. Common SW/HW codesign methodologies use unique languages for system modeling, partition as late as possible in the design process (because it is hard to change the HW design), build the SW/HW interface, and then hand-design or synthesize the HW portion of the design [3], [11]. There are few tools which can fulfill and automate all the steps. For example, [8] specifies a many-cache interface and the generation of spatial-data-flow VHDL blocks, however, it does not discuss the partitioning exploration and software-side interface design. Furthermore, the feasibility of synthesis is a matter of controversy, with some claiming that C/C++/SystemC-to-VHDL synthesis is a finished work [9] with others [2] pointing out that currently available tools impose constraints on the input codes (e.g. most of the synthesis tasks they considered are signal processing applications) or narrow the synthesizable set of C constructs. Reference [12] compares three C-to-VHDL synthesizers (SPARK, DWARV, ROCCC) in terms of the quality of generated VHDL codes and the C constructs that are supported. However, none of them synthesizes virtual methods, shared data, and reference-counted data that are widely used and easily applied in architecture modeling.

III. The Structure of SPRI

Because the process of manually designing and debugging a hybrid simulator is time-consuming, we have proposed SPRI to synthesize hybrid simulators without any manual intervention. SPRI integrates partitioning, interface synthesis, and VHDL generation into one infrastructure, as shown in Figure 1. SPRI has three basic blocks—the partitioner, interface synthesizer, and SystemC-to-VHDL synthesizer. SPRI reads a partitioning specification and separates an input SystemC architecture simulator into a SW portion and a HW portion. It then automatically synthesizes the SW/HW interface and generates VHDL code for the HW portion. A device API library and a HW wrapper library are plugged into the infrastructure to support different interfaces and multiple simulation platforms. SPRI is built on the LLVM compiler framework[4] and takes advantage of LLVM's ability to perform both static and runtime optimization. SPRI operates directly on the LLVM internal representation (IR), which is composed of RISC-like instructions in single-static assignment form.

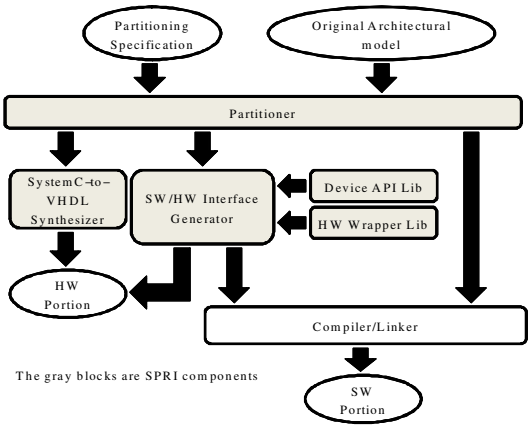

Fig. 1. SPRI Block Diagram

IV. Model Partitioning

Model partitioning uses a simulation database and the LLVM internal representation of code rather than the source code of the architectural model. The simulation database is created by a run of the input SW simulator within the SPRI runtime environment; this run stops after elaboration, but before the simulation starts. The simulation database contains the instantiated SystemC objects and structural information such as process, port, signal, bit width, and sensitivity. The partitioner interprets a partitioning specification, and extracts from the database the structural information for the interface synthesis and the LLVM IRs of the moved-to-HW portion for the VHDL synthesis.

The partitioning specification is given in a simple language that tells the partitioner which portions of the architectural model should be partitioned into HW. Three examples of partitioning specifications are listed in Figure 2: Example 1 means all the components instantiated from DLX_Adder class are moved to hardware; the second one specifies a template class and a normal class; the last one says that only the FrontEdge process of the DLX_IDEX_register class should be implemented in HW.

The partitioning affects the interface synthesis and the VHDL generation. Partitioning along structural lines (e.g process, instance, and class partitioning) is straightforward, since the components in SW and HW are all processes and they communicate with each other naturally through signals. Implementing the interface equals routing correct signals across the SW/HW boundary. However, non-structural partitioning (function-call, data-layout partitioning) is not so simple, because non-structural partitioning influences shared data and global variables appearing on both the SW side and the HW side; other communication methods are required to keep data consistent. The partitioner converts any shared-data and global-variable update into the uniform format of signal sending and receiving. The conversion simplifies the interface designs to manipulate only communication signals, just like a structural interface, permitting switching from one simulation platform to another without design changes by users. However, this method requires that SPRI wrap the non-structural elements of the simulator within structural elements, before any synthesis work starts. The partitioner must manipulate the LLVM IR to achieve this. For example, if a SystemC process contains an unsynthesizable system call, the partitioner should keep that call instruction in SW and split the process into "before and after" processes in HW. Similarly if two processes drive the same ports or signals, the partitioner should merge them together, because directly mapping them to VHDL would cause a multi-source driving error.

```
Specification Example:

1. tohw[::"DLX_Adder"] ;

2. tohw[::"DLX_MUX<3,DLX_UInt32>"]|[::"DLX_Registers"];

3. tohw[::"DLX_IDEX_Register"::"FrontEdgeProcess"%func%];
```

Fig. 2. Partitioning Specification Examples

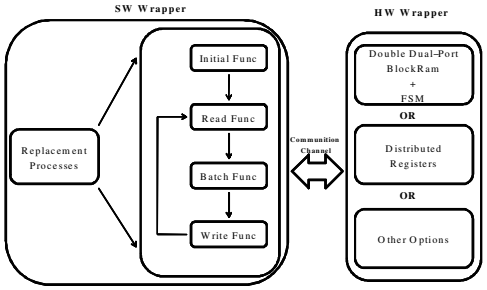

Fig. 3. Synchronous SW/HW Communication

V. Interface Synthesis

We encapsulate the SW/HW communication actions within a SW wrapper and a HW wrapper. With the support of a device API library and a HW wrapper template library, the interface synthesizer can produce compatible SW wrappers and HW wrappers for different simulation platforms. The SW wrapper calls replacement processes to start the data transfer with the HW instead of executing the original functions which have been removed by the partitioner. Figure 3 shows the synchronous communication procedure that is currently supported. On the SW side, the synthesizer generates a polling interface for all SW components, which sequentially initializes HW registers/memories (initial func), stores data from input ports to a SW buffer (read func), streams the data into the HW and pulls the results back (batch func), and then returns the results from the SW buffer to output ports (write func); on the HW side, it has two interface options— a FSM-controlled double dual-port BlockRams design and a distributed-register design, as shown in Figure 4.

The synchronous hardware invocation discussed above limits the speed of hybrid simulation, as will be demonstrated in Section VII. Overlapping the communication and the computation can enhance the performance and be accomplished through the use of asynchronous interface which is able to concurrently invoke multiple HW instances. There are no deadline

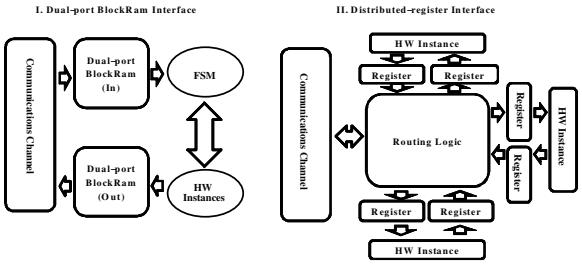

Fig. 4. Two Currently Supported Interfaces

constraints on hybrid simulation, that means, spending multiple HW (FPGA) cycles within one target simulation cycle would not cause any timing violation. However, independent HW components must not rely on the latency of neighboring components. Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) have the property that each BDN is latency-insensitive with each other in that different LI-BDNs may take different cycles to compute and therefore they can be composed arbitrarily [10]. Thus, transforming our input SystemC model to LI-BDNs would loosen the synchronous constraint and allow the creation of asynchronous interfaces. We are currently extending the interface synthesizer in this direction.

VI. VHDL Code Generation

The VHDL generator creates the VHDL blocks that implement the portions which have been partitioned out from the SW simulator. In order to avoid parsing SystemC source code and dealing with the fine points of its semantics, our VHDL synthesis is performed on the LLVM IR by translating each IR element to the corresponding VHDL codes or statements.

Unlike other synthesis tools, SPRI synthesizes during a run of the original SW simulator, after all SystemC objects are elaborated. Thus, we gain access to runtime data and do not need to analyze SystemC object instantiation. For example, we can figure out what virtual function is being called by tracing the object pointer through a virtual method table; we can also obtain the initial values for the registers, states and memories of the target simulator by reading the correct locations of the system memory. We synthesize all non-local variables of SystemC processes to be simulator state, e.g. VHDL signal. Not all of them are target model state. We do not need to distinguish between the two, since a hybrid simulator is not a prototype of the system. This process also allows users to produce simulators with different structural parameters (e.g. number of cores, interface methods) from command line options. We chose not to use commercial high-level synthesis tools because they operate only on source code at compile time and do not provide this flexibility.

In order to produce synthesizable VHDL codes, our VHDL synthesizer utilizes three methods—direct mapping, indirect translation, and IP-core instantiation. We summarize the synthesis details categorized by their implementation methods in Figure 5, Figure 6, and Figure 7. All these elements can be synthesized at present. Figure 8 lists the additional synthesis work that needs to be done.

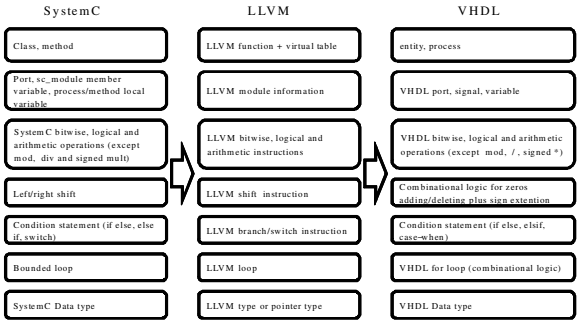

Fig. 5. Direct Mapping

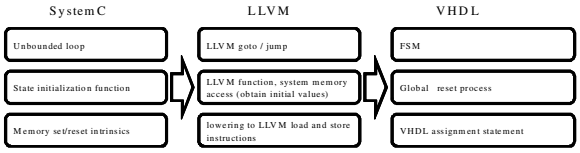

Fig. 6. Indirect Translation

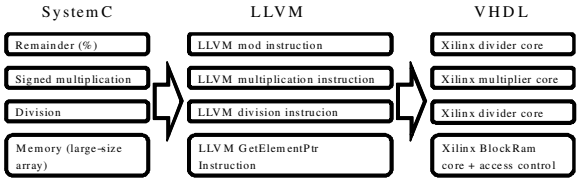

Fig. 7. Xilinx Core Instantiation

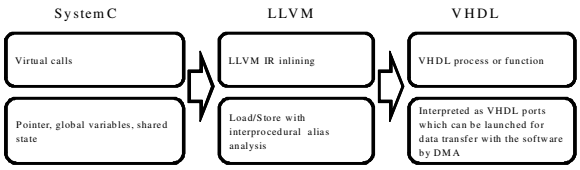

Fig. 8. Future Extensions

VII. A Case Study

In this case study, we use the DRC 1000 system as the simulation platform. It has a dual-core AMD Opteron-275 CPU running at 2.1Ghz with a 2GB system memory and a Xilinx XC4VLX60-11 FPGA as the coprocessor. They are connected by the HyperTransport DMA channel with a bandwidth of 800MB/s and a latency of 300ns per roundtrip communication.

We apply SPRI to a fine-grain microarchitecture simulator modeling a five-stage in-order pipeline implementing the DLX instruction set to test the correctness of this infrastructure and learn how different partitionings affect the performance of the different hybrid simulators. The DLX architectual model is composed of 23 module instances and uses such SystemC constructs as 1-D port arrays, 2-D variable arrays, 1-D structure arrays, local pointers, left/right shift operators, moduli operators, bit-wise/arithmetic operators, initialization functions, bounded loops, conditional statements and template classes. The original SystemC simulator has around 2200 lines

of code. When the entire simulator is moved to hardware, SPRI produces 5311 lines of VHDL code, 2000 lines of which are used for initialization. The simulator, synthesized by Xilinx ISE 8.2, supports a maximum clock frequency of 125 Mhz and occupies 37 percent of the FPGA slices and 30 percent of the FPGA FIFO/RAMB16s. We run the synthesized logic at 100 Mhz and the HW interface at 200 Mhz.

The baseline SW simulator runs at 149.7 KHz. The low speed of the baseline simulator is due to the very fine-grained model of the target architecture. By changing the partitioning specification, we can move individual module instances of the original DLX simulator from SW to the FPGA one by one. Figure 9 shows the performance of 12 different hybrid simulators with increasing numbers of instances moved to the FPGA compared with the LLVM-compiled SW simulator. When the number of the instances in HW is less than 5, adding instances into HW always increases the simulation time because the communication cost has also been raised; thereafter, keeping moving instances to HW makes the HW portion more internally-connected and thereby less data is transferred across the SW/HW boundary. The fastest hybrid simulator, implementing all the instances in HW, is 1.21x faster than the LLVM-compiled SystemC simulator.

Two things can be learned from this case study. First, the synchronous interface design for the fine-grain architectural model is the main reason why we didn't achieve high speedup, even when SPRI moves the whole model into the HW. Moving only fine-grain components such as multiplexers and registers into the HW does not provide sufficient work for the HW to trade-off the increased communication cost. Note that the communication cost still exists, even when all the instances are moved to the HW, because the SW portion is synchronized with the HW portion in every target simulation cycle through the synchronous interface. An asynchronous interface, on the other hand, would allow a flexible rearrangement of the invocation of HW components and hide a part of the communication cost within the computing delay. Thus using coarser-grain models of multi-core architectures with an asynchronous interface will be important to improve the speed of hybrid simulation. Second, the automatic generation process provides an opportunity to explore partitionings in an efficient way. SPRI is able to generate the VHDL description of a hybrid simulator in about a minute, thus allowing many partitioning strategies to be explored. The all-component-in-HW simulator is the best for this case, however, a more-complex multi-core design will be constrained by FPGA resources. Partitioning exploration will be necessary in that case.

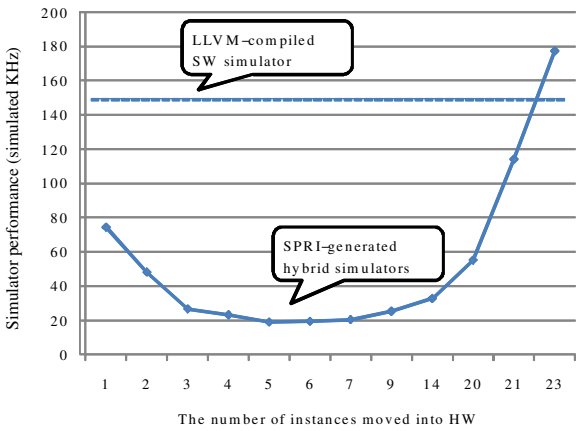

Fig. 9. Simulator Performance

VIII. Conclusion

Designing a hybrid architecture simulator is a SW/HW codesign problem. SPRI has a specific application domain – hybrid simulation – which makes it different from other SW/HW codesign infrastructures. SPRI automates the three design steps (partitioning, interface synthesis and VHDL generation) in one infrastructure. This paper shows our research and development of SPRI and future extensions. SPRI is a flexible infrastructure which supports arbitrary partitionings, multiple interface designs, and multiple simulation platforms. When it is completely in place, we also expect to research automatic search for good partitioning decisions.

References

[1] 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 International Conference on Computer-aided Design*, 2007, pp. 295–302.

[2] C. Cote and Z. Zilic, "Automated SystemC to VHDL translation in hardware/software codesign," in *Proceedings of the 9th International Conference on Electronics, Circuits and Systems*, vol. 2, 2002, pp. 717 – 720.

[3] R. Gupta and G. D. Micheli, "Hardware-software co-synthesis for digital system," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29–41, September 1993.

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

[5] 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.

[6] 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.

[7] 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.

[8] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers, "Chimps: A C-level compilation flow for hybrid CPU-FPGA architecture," in *Proceedings of the 2008 International Conference on Field-Programmable Logic and Applications (FPL)*, 2008, pp. 173–178.

[9] J. Sanguinetti, "A different view: Hardware synthesis from SystemC is a maturing technology," *IEEE Design and Test of Computers*, vol. 23, pp. 387–387, 2006.

[10] 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.

[11] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, April 2003.

[12] Y. Yankova, K. Bertels, S. Vassiliadis, R. Meeuws, and A. Virginia, "Automated HDL generation: Comparative evaluation," in *Proceedings of 2007 International Symposium on Circuits and Systems*, 2007, pp. 2750–2753.