Issues in Hybrid Simulator Synthesis

Zhuo Ruan David A. Penry
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT 84602

Abstract

The Simulator Partitioning Research Infrastructure (SPRI) is a project to automate the generation of hybrid architectural simulators. In this paper, we examine the interesting issues and challenges in hybrid simulator synthesis.

Introduction and Motivation

Hybrid simulation[3, 4, 6] has been proposed as an efficient means to accelerate architectural simulation without resorting to complete prototyping of the system. However, the developer of a hybrid simulator must partition the architectural model into software and hardware portions and also implement the hardware. These tasks have proven to be quite time-consuming.

In WARP 2007, we proposed the Simulator Partitioning Research Infrastructure (SPRI) as a means to reduce the time needed to develop a hybrid simulator.[7] SPRI allows the simulator developer to specify what portions of a structural simulation model are to be moved to hardware. It then partitions the model and synthesizes both the interface and the VHDL code for the hardware. Currently SPRI accepts SystemC models as input. This paper discusses the issues which we have encountered in interface and SystemC-to-VHDL synthesis and the solutions we have adopted or are adopting.

Interface issues

- **Shared state.** State which is shared between the hardware and the software must be either allocated in hardware, allocated in software, or allocated in both and kept consistent. We are still exploring the tradeoffs involved. Shared state allocated in hardware is relatively simple to implement, but allocation in memory is more versatile and may be more efficient when the hardware platform supports DMA.

- **Software calling hardware.** A software call to hardware is currently performed by sending a message to the hardware and polling for a status flag to be set. Other interfaces which allow the software to continue execution rather than spin-waiting are envisioned in the future. Note that invocation of synthesized SystemC processes is handled as a software call to hardware.

- **Hardware calling software.** We do not yet allow hardware calls to software, but plan to handle these calls in the future. One proposed method is by synthesizing a message queue which the software periodically polls.

- **Virtualization and debug.** Hardware virtualization and debugging are important features of an interface. We intend to support the strategies employed in RAMP by synthesizing compatible interfaces.

- **Communication minimization.** At present, the synthesized interface is quite simple and communication to different synthesized processes is carried out serially. Batching of this communication will be necessary to achieve good performance. We also plan to implement datatype bit-width analysis techniques from the literature to reduce the amount of data communicated.

SystemC synthesis issues

SPRI is based upon the LLVM compiler framework[5]. We operate directly on LLVM's IR rather than source code; doing so removes the need to parse C++ or deal with the fine points of its semantics. We perform synthesis during a run of the original simulator – after all SystemC objects are elaborated, the partitioner calls the synthesizers to produce the hybrid simulator. Thus we do not need to analyze object instantiation and we gain access to runtime data. This process also allows the user to produce simulators with different structural parameters (e.g. number of cores) from the command line. We chose not to use commercial high-level synthesis tools because they operate only on source code at compile time and do not allow us this flexibility.

The issues we have encountered in synthesis are:

- **Identifying the code.** SystemC makes extensive use of virtual methods in its objects, which complicates call graph analysis. However, pointers to all SystemC object instances are readily available during synthesis, and thus their virtual method tables can be read.

LLVM allows us to map a method pointer back to the original IR for the method, which we then synthesize.

- **State identification and initialization.** All non-local variables of SystemC proceses are simulator state. Not all of this state is target model state, but as a hybrid simulator is *not* a prototype of the system, we do not need to distinguish between the two. Instead, all simulator state is treated as if it were target state; it is synthesized as a signal. To preserve LLVM's sequential semantics as simulator state is updated, all the VHDL processes we generate first read all their input signals into variables. The processes then operate only upon variables. At the end of the process, all output signals are assigned. While the code produced is verbose, the VHDL-to-logic synthesizer later optimizes away simulator state which is not target state. Simulator state is initialized in the hardware; the initial values are known because we are synthesizing after elaboration.

- **Mapping of LLVM IR to VHDL.** Most LLVM instruction and type contructs map quite naturally into VHDL statements and types. The only constructs which present problems are gotos arising from irreducible loops and pointers.

  Pointers are a major issue; we will not be able to efficiently synthesize all possible uses of pointers. (Pointer dereferences can always be synthesized as DMA accesses, but this could be highly inefficient.) However, we do handle the most common uses. First, references to the "this" pointer for module instances are easy to recognize and implement. Second, LLVM encodes array dereferences in an easily-analyzable form allowing us to succesfully synthesize array structures such as register files.

  One important case is that of reference-counted shared data structures. Simulators often use these for "instruction instance" data types and just pass around pointers to the instructions. We do not yet have an efficient solution for this problem.

- **Time.** Simulators often implement counters by inspecting the current simulation time and comparing it with a reference time. A planned optimization is to directly synthesize timers to replace the time access and comparison code.

Interoperability issues

We wish to add both Unisim[2] and LSE[8] as front ends to SPRI because they have better reusability features than SystemC. Furthermore, we wish to be compatible with the RAMP[1] infrastructure and interfaces. These wishes lead to issues with the simulator's model of computation, as each system uses a different model. There is insufficient space here to go into the details of how to transform the model of computation of a simulation model; however, transformation involves the merging and splitting of processes and analysis of the simulator scheduling.

Conclusion

When SPRI is finished, it will be able to synthesize most high-level language features frequently used in architectural modeling and will support LSE, Unisim, and SystemC and will take advantage of the RAMP infrastructure. Currently, SPRI is working from end to end – automatically generating different hybrid simulators given different partitioning specifications – for a simple processor model.

References

[1] Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. CRI: RAMP: Research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform. Technical report, RAMP Project, 2005.

[2] 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*, 6(2):45–48, July–December 2007.

[3] 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*, pages 295–302, 2007.

[4] 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.

[5] 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*, pages 75–86, 2004.

[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*, pages 29–40, Feb. 2006.

[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] 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*, pages 271–282, November 2002.