View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

PROTOFLEX: Co-Simulation for Component-wise FPGA Emulator Development

Eric S. Chung, James C. Hoe, Babak Falsafi
Computer Architecture Laboratory at Carnegie Mellon (CALCM)
{echung, jhoe, babak}@ece.cmu.edu

1. Introduction

This paper presents PROTOFLEX, a hardware/software co-simulation methodology to facilitate the systematic development of RTL components for an FPGA-based multiprocessor emulator. PROTOFLEX relies on FLEXUS [1], a full-system, cycle-accurate software simulator, to provide the reference behavior of a distributed shared-memory multiprocessor system and its components. The simulated C++ components in FLEXUS can be mapped into RTL descriptions piece-wise, and individual RTL components can be co-simulated with FLEXUS for debugging and testing. The PROTOFLEX methodology enables a steady refinement path toward completing an FPGA-based full-system emulator.

Paper Outline. Section 2 further argues the advantages of a co-simulation framework in developing a FPGA-based large-scale, full-system emulator. Section 3 provides details of the PROTOFLEX methodology. Sections 4 and 5 describe our experience in applying PROTOFLEX to develop the RTL model of a microprogrammable protocol engine for directory-based cache-coherence.

2. Motivations

A major challenge in building an FPGA-based full-system emulator is the verification and composition of multiple, potentially broken RTL components. Below, we argue for the use of co-simulation in component development to address the above challenge.

In-system component testing. Co-simulating individual RTL components with a complete reference software simulator enables early and efficient testing by operating the target RTL component in a *reliable* environment—regardless of the progress of RTL development for the rest of the system. Equally important, the target RTL component can be tested under *realistic* operating conditions that are difficult to create using conventional testbenching. (In our case, FLEXUS simulates in detail a complete distributed shared memory system that can boot unmodified Solaris and execute commercial applications.) The RTL component under test can even be mirrored simultaneously by its original C++ counterpart during co-simulation to detect any divergence from the reference behavior.

Advanced test and debug support. Co-simulation can leverage the capabilities of modern software simulators to achieve more sophisticated and thorough component testing than possible by direct, isolated component testing. Modern simulators are highly configurable (number of processors, device latency, etc) and support a large variety of execution modes (ranging from functional-only to cycle-accurate, support for sampling, checkpointing, etc). Simulation scripts can automatically test an RTL component against a large variety of workloads and system configurations. This can help to "sweep" entire classes of errors that would otherwise be arduous to identify using conventional testbenching. Software simulators also offer much more user-friendly and powerful execution tracing, assertion checking and state inspection—throughout the system—to help error detection and replication.

Concurrent component development. The ability to test isolated RTL components in a complete, simulated environment allows for independent and concurrent development of different components. Components pre-validated using PROTOFLEX are much more likely to work correctly together during the bring-up of the final FPGA-emulated system. This is particularly important in a distributed collaborative development effort.

Component-specific studies. In a separate vein from full-system emulation, co-simulation is also useful for studying a subset of the complete system. The architect can concentrate his RTL development effort on only the subsystem of interest to characterize implementation metrics such as cycle time, area, and power. The architect can nevertheless validate the functional correctness of the subsystem against the complete system behavior.

3. PROTOFLEX

In this section we present PROTOFLEX, a HW/SW co-simulation methodology for piece-wise derivation of RTL components for an FPGA-based emulator from a reference software simulator. We begin with a brief overview of the FLEXUS infrastructure which serves as our reference software simulator. We then discuss the component development flow in PROTOFLEX.

3.1 FLEXUS: A full-system simulation framework

FLEXUS is a component-based C++ framework for creating simulation models of uni- and multi-processor systems. At the heart of FLEXUS is the *Simics* simulator [5], an application which enables functional execution of unmodified, commercial OS and applications. User-developed C++ component models supply timing and structure details for timing- and bit-accurate simulations.

FLEXUS simulates a system by connecting *components* through well-defined timing-independent interfaces called *ports*. *Components* correspond to portions of the system being modeled (e.g., a processor core or the cache-coherence engine). Source code for different components

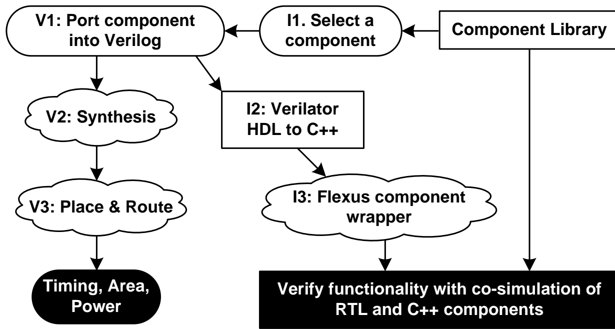

Figure 1 PROTOFLEX methodology

(and their internal hardware algorithms) are clearly separated and timing-independent. *Ports* provide the basic point-to-point channel abstraction between the connected components. Developers can specify the direction of the port and select from a collection of pre-defined handshakes (e.g., asynchronous push, pull, etc.). A port communicates data across components in packets of arbitrarily-complex user-defined data structures called *transports*. In each simulated cycle, FLEXUS'S cycle-based simulation kernel invokes each component's *drive* function, which performs a cycle's worth of logic and communication activities. Timing effects (e.g., cache miss or network delay) are introduced in a component by postponing the processing of *transports* by an appropriated number of *drive* function invocations.

The FLEXUS component and port abstractions are the key enablers for integrating an RTL model with the full-system simulator for co-simulation. The *port* interface abstraction in FLEXUS lends itself naturally to a hardware timing-independent interface used by the RTL modules. Therefore, an RTL model can be directly mapped into a FLEXUS component by translation between *transports* and RTL signals. Liberty [3] and ASIM [4] are other examples of component-based simulators with well-engineered modularization and component interfaces.

3.2 PROTOFLEX methodology

The key steps in the PROTOFLEX component development flow are given in Figure 1. A developer begins by choosing a hardware component of interest from the FLEXUS component library (Step I1) and manually ports the cycle-based C++ FLEXUS component model to its corresponding Verilog RTL model (Step V1). In Step V1, the C++ component model conveniently serves as an unambiguous design specification. The current release of FLEXUS (http://www.ece.cmu.edu/~simflex/) contains over 20 base library components ranging from processors (e.g., x86, SPARC) to memory controllers and interconnects.

The completed RTL model is then converted back into a C++ object using the Verilator tool [6] (Step I2). This RTL-derived C++ object is instantiated within a FLEXUS component wrapper (Step I3). In Step I3, the developer is only responsible for writing a wrapper that converts between exposed RTL signals and software *transports* that move over *ports*. This usually involves copying between *transport* data and RTL signal fields. In some cases, *transports* contain simulation-specific metadata (e.g., statistics and counters) that must be preserved in the wrapper.

Lastly, the RTL-derived C++ model is co-simulated within the full-system model for testing and debugging. Verifying the functionality of the RTL component by co-simulation involves a combination of tracing and assertion-checking. The developer can easily place checks and debug statements in the components surrounding the ported component. The FLEXUS framework provides a rich debugging infrastructure that allows printing of debug statements in selectable components. In parallel with functional testing, the Verilog RTL model also can be synthesized to assess implementation metrics such as timing, area, and power characterizations (Steps V2, V3).

Because each component is validated against the presumed-to-be-correct reference simulator, the porting of different components can be achieved concurrently by multiple independent developers. By pre-validating individual components against a full-range of behaviors in a complete system, one greatly increases the likelihood of successfully combining the RTL components in the final FPGA-emulated multiprocessor system.

4. Design Study: Cache-Coherence Engine

We next present an application of the PROTOFLEX discipline to develop the synthesizable RTL model of the cache coherence protocol engine modeled in FLEXUS. In this section, we briefly introduce the design specification of the cache coherence protocol and the protocol engine. The next section reports the development experience and the lessons learned during the design, verification, and characterization process.

Cache Coherence Protocol. The distributed shared-memory multiprocessor system simulated by FLEXUS employs an aggressive, directory-based MSI protocol. Beyond basic distributed MSI protocol designs, the FLEXUS protocol has been aggressively optimized to minimize transaction occupancy at the home nodes and to minimize the number of network hops per transaction. For example, a read transaction to a remote, dirty cache line is satisfied in three hops where 1. the request reaches the home node 2. the home node forwards the request to the current owner of the dirty cache line (and does not wait for further acknowledgement) 3. the owner of the dirty cache line responds directly to the requestor. The protocol requires three virtual channel priorities but does not require point-to-point ordering. Key properties of the protocol specification have been formally verified using Murphi [2].

Protocol Engine. FLEXUS simulates, at a cycle- and bit-accurate level, a protocol engine on each processing node to implement the aforementioned cache-coherence protocol. The protocol engine on each node actually

consists of three independently operating submodules: the *local engine* (LE), the *home engine* (HE) and the *remote engine* (RE). The HE and RE are microprogrammable to allow late binding of protocol features.

Figure 2 illustrates the high-level organization of the three protocol engines relative to other modules in a processing node. The L2 cache controller services cache misses by sending requests to the LE. The LE is a hardwired controller optimized to handle and respond to accesses to the directory and local data in the main memory. The LE forwards requests requiring inter-node communication to the HE or the RE as appropriate. The HE handles 1. cache requests to local memory blocks unserviceable by the LE (e.g., request to a local memory location being modified remotely) and 2. requests for local memory locations from a remote node. The RE handles cache coherence for memory blocks mapped to a remote node. All communication to external agents is achieved through links between the HE/RE and the network interface (NIC).

The HE and RE are implemented as micro-programmable controllers. The HE and RE hardware is nearly identical, except for minor differences in state registers. The HE and RE support overlapping threaded execution of multiple outstanding transactions. Each new transaction starts a new thread and is allocated a private transaction state register. A thread executes until it encounters a blocking operation (e.g., executing a RECEIVE microinstruction to wait for a coherence message). A thread can also be suspended when acquiring a directory lock or waiting for directory state to return from memory. A thread scheduler dynamically monitors the state of each outstanding transaction to select a ready thread to execute on the protocol engine.

Protocol Firmware. The protocol firmware is coded in a high-level language with symbolic arguments and C-style code blocks. An assembler compiles the high-level protocol description to microcode for the HE and RE, respectively. The microinstruction set comprises only 14 instruction types, sufficient to describe the complete set of actions by the HE and the RE in 471 and 928 instructions. Most transactions require only a handful of instructions. Examples of typical instruction types are SEND, RECEIVE, TEST, and SET. RECEIVE and TEST support conditional control flow to multiple destinations.

The RTL model of the coherence engines in this case study is binary-compatible with the original C++ model in FLEXUS. In other words, the protocol engine in RTL can directly inherit the fully-debugged microcode and thus cache-coherence protocol from FLEXUS.

5. Implementation Experience

We conclude with a discussion of our experience in implementing the protocol engines using PROTOFLEX. Although not central to the PROTOFLEX methodology, we opted to develop synthesizable hardware models using the

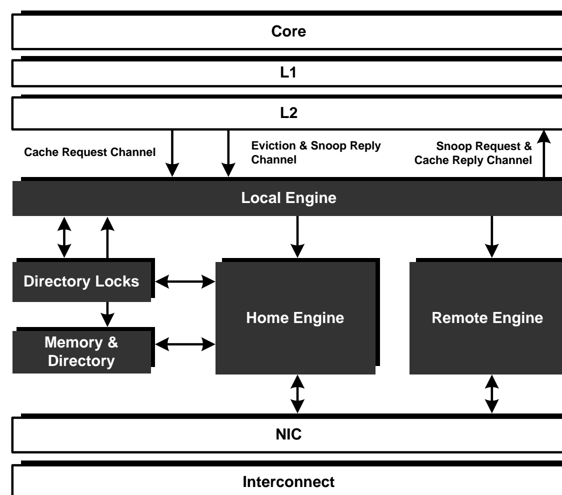

Figure 2 Components that maintain cache coherency in a distributed shared memory architecture

Bluespec System Verilog (BSV) language and compiler. We begin with a brief overview of BSV.

5.1 Bluespec: High-Level Description and Synthesis

BSV is a high-level, strongly-typed operation-centric hardware description language. A valid BSV description can be compiled to a cycle-accurate C simulator or to a synthesizable Verilog RTL for further backend synthesis.

Hardware structures in BSV are represented by modules—recursive encapsulation of states, rules, and interfaces. In a module, states (e.g., FIFOs and registers) are explicitly declared and operated on by rules and interface methods.

Hardware behaviors in BSV are specified in terms of guarded atomic actions (or rules) that transform the states. Transformations on state are applied only when a rule can fire (i.e., when its guarding condition is satisfied). Recasting hardware concurrency as atomic rules greatly simplifies the treatment of race conditions when a shared state can be modified from multiple sources. While the programmer perceives simplifying serial execution semantics during development, the BSV compiler synthesizes highly concurrent hardware implementations by allowing as many possible rules to fire in parallel without violating the sequential semantics—that is, the concurrent firing of multiple rules must obey atomic execution semantics.

Generally, we found that BSV helped to capture a complex design with fewer bugs—due to its strong type system and atomic rule semantics. The LE took approximately three weeks to design and test, and the HE and RE together took six weeks (including multiple iterations of simulation and debugging). The ability to parameterize a design (e.g., FIFO sizes, or even the number of threads in the HE/RE) in BSV was very useful in evaluating how the overall system performance was impacted when replacing a software component with its RTL counterpart in multiple configurations. Furthermore,

the ability to vary the number of threads supported by the HE and RE gives further validation of the safety of the protocol design.

5.2 Hardware model development

We applied PROTOFLEX to independently develop the LE, HE, and RE as three separate components.

Porting the Local Engine. The LE is a hardwired-controller and was implemented using only five rules to handle interactions (through channels) with other components, namely the directory, main memory, HE, RE, and the L2 cache controller. In all, the LE was implemented in approximately 2000 lines of BSV code (compared to 1000 lines in the FLEXUS C++ component).

Porting the Home and Remote Engines. The HE and RE (nearly identical) were each implemented using ten rules. Most of the rules dealt with managing concurrent accesses to the transaction state register file. Using BSV's composable interface abstraction, both the HE and RE reused much of the same descriptions (e.g., thread scheduling, microcode storage) except where it was necessary to maintain separate interfaces for engine-specific operations (e.g., only the HE accesses the directory). The HE and RE were altogether implemented in approximately 4000 lines of BSV code (not including the embedded microcode). As a reference, the original FLEXUS C++ models for the HE and RE were altogether implemented in approximately 7000 lines of code.

The LE, HE, and RE were all synthesized for timing and area estimates for the Xilinx Virtex II Pro 70. An LE capable of supporting 16 threads occupies 14199 slices and operates at 84 MHz. (The LE required a large number of slices due to a large number of fully-associative lookups. This will be substantially reduced in future iterations by replacing several unnecessary fully-associative structures by more compact indexed arrays.) An HE capable of supporting 16 threads requires 7862 slices and operates at 46 MHz while the RE occupies 8510 slices and operates at 47 MHz.

5.3 Co-simulation and debugging

The LE, HE, and RE were each independently tested by compiling their BSV description to a synthesizable Verilog RTL description and then by the Verilator to a simulatable C++ object. Each component was tested against FLEXUS in both functional- and timing-accurate full system simulation of multiple system configurations and a variety of scientific and commercial workloads.

Running different workloads exercised a wide variety of test cases. For example, OLTP on IBM's DB2 and Oracle DB exercised a variety of race conditions due to migratory sharing of critical sections. Ocean, a memory-bandwidth bound benchmark, tended to maximize the number of concurrent threads. Checkpointing support in FLEXUS allowed rapid execution of selected pre-determined program phases to acquire additional coverage for long running applications with exceptionally large-scale program phases.

Varying system configuration parameters also helped to exercise a variety of test cases. For example, the L2 cache size was reduced to generate significant writeback traffic and to exercise complex writeback races. Recreating representative test cases for such programs and configuration-specific scenarios involving multiple nodes in standalone component testing would have been extremely difficult.

For all of the components, it was easy to create a debugging package in BSV, which allowed insertion of high-level print statements, which could be inserted into any point in action methods or rules. This provided the power of C-like debugging in an RTL environment. These statements remained in the synthesized Verilog surrounded by synthesis pragmas, which are pruned during synthesis. As a result, the Verilator-generated C++ object still printed the high-level debug statements during FLEXUS execution, which aided with viewing internal signals in the RTL components.

The debug package helped especially with documenting protocol test cases. One approach used to increase coverage was to identify entry points into the microcode and to maintain a running list of protocol transitions as well as rare race conditions being exercised. The FLEXUS environment helped to identify several rare race conditions and also exercised all state transitions in the protocol diagram.

6. References

[1] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. SIGMETRICS Performance Evaluation Review, May 2004.

[2] D. Dill. The Murphi Verification system. In 8th International Conference on Computer Aided Verification, pages 390-393, 1996.

[3] M. Vachharajani, N. Vachharajani, D. A. Penry, J. Blome, and D. I. August, 2004. The liberty simulation environment, version 1.0. Performance Evaluation Review: Special Issue on Tools for Architecture Research 31, 4 (Mar.).

[4] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, T. Juan. "Asim: A Performance Model Framework," Computer, vol. 35, no. 2, pp. 68-76, February, 2002.

[5] M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, P. S. Magnusson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50-58, February 2002.

[6] Verilator tool suite. http://www.veripool.com.