2013-12-01

# Interface Design and Synthesis for Structural Hybrid Microarchitectural Simulators

Zhuo Ruan
*Brigham Young University - Provo*

# Interface Design and Synthesis for Structural Hybrid Microarchitectural Simulators

Zhuo Ruan

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

David A. Penry, Chair
Mike J. Wirthlin
Brad L. Hutchings

Department of Electrical and Computer Engineering

Brigham Young University

December 2013

ABSTRACT

Interface Design and Synthesis for Structural Hybrid Microarchitectural Simulators

Zhuo Ruan
Department of Electrical and Computer Engineering, BYU
Master of Science

Computer architects have discovered the potential of using FPGAs to accelerate software microarchitectural simulators. One type of FPGA-accelerated microarchitectural simulator, named the hybrid structural microarchitectural simulator, is very promising. This is because a hybrid structural microarchitectural simulator combines structural software and hardware, and this particular organization provides both modeling flexibility and fast simulation speed. The performance of a hybrid simulator is significantly affected by how the interface between software and hardware is constructed. The work of this thesis creates an infrastructure, named Simulator Partitioning Research Infrastructure (SPRI), to implement the synthesis of hybrid structural microarchitectural simulators which includes simulator partitioning, simulator-to-hardware synthesis, interface synthesis. With the support of SPRI, this thesis characterizes the design space of interfaces for synthesized hybrid structural microarchitectural simulators and provides the implementations for several such interfaces. The evaluation of this thesis thoroughly studies the important design tradeoffs and performance factors (e.g. hardware capacity, design scalability, and interface latency) involved in choosing an efficient interface. The work of this thesis is essential to the research community of computer architecture. It not only contributes a complete synthesis infrastructure, but also provides guidelines to architects on how to organize software microarchitectural models and choose a proper software/hardware interface so the hybrid microarchitectural simulators synthesized from these software models can achieve desirable speedup.

Keywords: hybrid microarchitectural simulator, software/hardware codesign, SystemC, FPGA

# ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, David A. Penry, for his support and mentoring. He founded the BYU Architecture Design, and Description (BARDD) group in 2007 and has provided an exciting environment for me and others to explore research and development of CAD tools to support hybrid microarchitectural simulation.

Thanks also go to Brigham Young University and NSF grant # CCF-1017004.

I'd also like to thank Trevor Meyerowitz, my colleague at Tensilica, for his generous comments on improving the writing of this thesis.

Finally, a special thank-you goes to my wife (Rachel Yu Liu) and my parents (Yongtie Ruan and Chune Jiang). I can't ever make it this far without their care and support. They sacrificed their dreams for mine.

# Table of Contents

## List of Tables

## List of Figures

# Chapter 1

# Introduction

Microprocessors are everywhere around us! They are widely used in designing digital electronic products such as desktops, smartphones, TVs, home security systems, and wifi routers. In these products, multiple microprocessor cores can be integrated in a System On Chip (SOC) where typically a central controller offloads tasks (e.g. audio, image, network) to special-purpose processor cores. These cores can be easily programed and reprogrammed with high-level languages (i.e. C, C++) after fabrication which significantly increases design productivity and improves time-to-market when they are compared with application-specific integrated circuits (ASICs).

Multiple microprocessors may be planned for a single product. For each microprocessor, computer architects must consider many design alternatives. An efficient way to evaluate each design candidate is constructing a microarchitectural model (also known as a microarchitectural simulator). The execution result of a microarchitectural model can predict the timing and the functionality of a microprocessor design. Microarchitectural models must be easy to create and flexible to modify so that architectural changes can be quickly added. Traditionally, architects choose to verify microarchitecture designs through software simulation rather than hardware prototyping for two reasons. First, the creation of software models is less time-consuming. Second, these software models are easier to create, modify, and debug.

Microarchitectural models are different from physical microprocessor designs or prototypes; they abstract the implementation of a microprocessor design to a higher level. A microarchitectural simulator doesn't necessarily model every implementation detail as a hardware prototype does, but it must capture how target instructions are processed at each simulated cycle in order to be cycle-accurate. A software microarchitectural simulator is typically running on a host platform which can be either a server or a computer workstation. It loads an executable binary for the simulated system and virtually executes each instruction of this binary through the microarchitectural

**Table 1.1:** Performance of software-only cycle-level simulators [5]

| Simulator | Instruction-set Architecture | Microarchitecture | Speed | OS |
|---|---|---|---|---|
| Intel | x86-64 | Core 2 | 1 - 10 KHz | Yes |
| AMD | x86-64 | Opteron | 1 - 10 KHz | Yes |
| IBM | PowerPC | Power5 | 200 KIPS | Yes |
| Freescale | PowerPC | e500 | 80 KIPS | No |
| PTLSim | x86-64 | Athlon | 270 KIPS | Yes |
| Sim-outorder | Alpha | 21264 | 740 KIPS | No |
| GEMS | Sparc | Generic | 69 KIPS | Yes |

model. This dissertation refers to the system being simulated as the target, the platform where the simulation runs on as the host, and the target executable binary as the benchmark.

Software microarchitectural simulators are commonly written in either sequential languages (e.g. C, C++) [1] or structural simulation frameworks (e.g. SystemC [2], Unisim [3], LSE [4]). However, it is more difficult to create timed processor models with sequential languages due to the lack of language constructs that can capture structural and timing features. By contrast, structural simulation frameworks allow the rapid creation of microarchitectural models in a concurrent and structural form that accurately mimics hardware behaviors. Unfortunately, the productivity advantage of structural modeling comes at the cost of simulation performance, because processing structural and timing constructs (e.g. signal, event, process) in software simulation introduces significant execution overhead.

Typically, simulation speed is measured by the number of target instructions executed per second (IPS) or simulated cycles per second (Hz). As target systems become more and more complex, their microarchitectural simulators are increasingly slowing down. Table 1.1 lists the software cycle-accurate or near-cycle-accurate simulators of several modern complex single-core processors. The speed of these simulators are at the level of KIPS or KHz. Simulating a processor of hundreds of cores with similar modeling details would cause a significant speed drop to the level of IPS or Hz. The simulation speed is so much slower than the speed of the physical processor that a relatively complex benchmark that a physical processor can finish in several minutes may take days or weeks to run in software simulation. Architects have become increasingly concerned with this problem, because the slower simulation speed results in a slower evaluation process and fewer design candidates can be evaluated.

Architects have advocated accelerating microarchitectural simulators by moving simulation workloads to hardware, because thousands of host processor cycles spent on fetching, decoding, and executing host instructions can be effectively reduced to a few cycles of hardware work. Hardware-accelerated simulators can be either fully-implemented or partially-implemented in hardware. A microarchitectural simulator that is partially implemented in hardware is named a hybrid microarchitectural simulator. A hybrid microarchitectural simulator is composed of a software portion, a hardware portion, and a software/hardware interface. Hybrid microarchitectural simulators are particularly interesting, because they allow trade-offs to be made between hardware capacity, ease of implementation, and simulator performance. A promising type of hybrid microarchitectural simulator is the hybrid structural simulator. Such a design combines the benefits of structural modeling and hardware acceleration.

Manually designing a hybrid simulator is very costly in that a great amount of time and efforts has to be spent on circuit implementation and interface construction. The inefficient design process extensively restricts the use of hybrid simulators in practice. This thesis starts with discussing the issues and challenges of the synthesis of hybrid structural microarchitectural simulators. This thesis primarily focuses on the interface design and synthesis techniques and explores the interface design space. The evaluation of this work thoroughly studies the important performance factors and design trade-offs (i.e. hardware capacity, interface latency, interface bandwidth, and scalability to simulate multiple processor cores) involved in choosing an efficient software/hardware interface for synthesized hybrid structural microarchitectural simulators.

## 1.1 The Synthesis Problem of Hybrid Structural Microarchitectural Simulators

Hybrid structural microarchitectural simulators combine the benefits of structural modeling and hardware acceleration. However, the time-consuming design process restricts their use in practice. A more efficient way to create hybrid structural microarchitectural simulators would be through an automatic synthesis process.

The synthesis of hybrid structural microarchitectural simulators is challenging. The synthesis techniques for general software/hardware codesign cannot be applied directly for hybrid simulators, although they have been researched for many years. This is because hybrid microarchitectural simulators are written and organized in a different fashion from general software/hard-

ware co-design systems. A wide range of general software/hardware co-design projects target data-intensive systems. These designs are typically partitioned in a certain way that the control logic is kept in software and the computational portion is accelerated in hardware. Unlike those designs, microarchitectural simulators have extensive data and control dependencies between simulator modules, and they commonly use a great amount of high-level language constructs to provide coding efficiency, modeling flexibility, and design productivity. Those features have caused many difficulties in the synthesis of hybrid microarchitectural simulators, because they can not be directly mapped to hardware.

### 1.1.1 Challenge 1: Simulator Partitioning

Simulator partitioning is challenging for two reasons. First, where a hybrid simulator is partitioned significantly affects performance [6]. Second, it should be flexible to change the partitioning boundaries so architects can explore arbitrary partitionings before selecting the best one to proceed. Simulator partitioning splits a microarchitectural simulator into a software portion and a hardware portion. The software portion and the hardware portion communicates with each other through an interface. A poor software/hardware interface boundary may cause extensive overhead on the interface and thereby leads to a hybrid simulator that runs even slower than a software simulator.

1. **Where to partition.** The software/hardware partitioning impacts three aspects in a hybrid simulator design: the amount of data that needs to be communicated, the amount of parallelism that can be exploited, and the amount of hardware that is used. Microarchitectural simulators have a great amount of data and control dependency between simulator modules. Simulator modules must frequently communicate with each other for synchronization. No matter how a microarchitectural simulator is partitioned, the software portion and the hardware portion must update processor state consistently to guarantee a correct execution sequence of target instructions. A ideal partitioning should place closely-dependent modules on the same side (either software or hardware) to minimize synchronization cost, and it also should place enough modules in hardware to achieve maximum speedup. However, the ideal partitioning may not stay the same for every hybrid simulator design in practice. For example, when accelerating a single-core model, one would prefer moving as much as

possible to hardware in order to reduce the communication overhead. This partitioning may not work effectively for a many-core model; a better partitioning option here would have a smaller portion of each core implemented in hardware with the parallelism between cores exploited to improve performance.

2. **Partitioning flexibility.** To provide the opportunity of exploring different hybrid simulators, a partitioning technique should support simulator partitioning along arbitrary boundaries. The more partitioning options architects have, the better decisions they can make on how to pick the best hybrid design.

   Partitioning flexibility allows different trade-offs to be made between the software and the hardware. These trade-offs exist, because simulator partitioning is constrained by both the hardware capacity and the synthesis capability. It is significantly beneficial to provide partitioning flexibility so architects are able to avoid dealing with the simulator portions that are either currently "unsynthesizable" or do not fit in the hardware.

How one partitions a microarchitectural simulator for the "hybrid" transformation depends on what this simulator models, how it is organized, and what host platform it runs on. Because architects are aware of all the details of how target simulators are written and which host platform is chosen, it may be better for architects to explore partitioning alternatives and find the best solution rather than replying on an automatic partitioning-selection process. Thus, our insfrastructure is designed to fit this need. The partitioning process is automatic but the partitioning decision and exploration are left to users.

### 1.1.2 Challenge 2: Hardware Synthesis

The hardware synthesis of hybrid structural microarchitectural simulators is challenging, because some high-level language constructs used in software microarchitectural models can not be directly and properly mapped to hardware descrption languages. To guarantee the functional and timing correctness of microarchitectural simulator after synthesis, novel techniques are required to interpret and translate high-level language features to hardware.

Typically, architects model microarchitectures in a structural simulation framework such as SystemC [2]. SystemC models are written with a great amount of high-level language constructs

to permit code reuse and coding efficiency. Table 1.2 list a set of high-level language features that are commonly used in SystemC microarchitectual models. For example, complex data with nested struct/array types can be used to contain the state information of each pipeline stage or decoded instruction information; a templated SystemC module class can be used for register modeling so the registered data can be different types for different register instances; command-line arguments offer a flexible way to parametrized the number of simulated processor cores, cache size, or register-file depth.

**Table 1.2:** List of high-level language constructs commonly used in software microarchitectural models

| Construct Category | Construct Example | Construct Usage |
|---|---|---|
| Complex Data Types | Nested struct/Array, Vector, List, etc. | Signal data type, Register type, Decoded instruction information, Pipeline stage state, etc. |
| Command-line Argument | Dynamic-allocated pointer, etc. | Cache size, Register-file size, Processor core number, etc. |
| Code Reuse | Template, Virtual methods, Class inheritance, Operator overload/override etc. | Register, Buffer, Instruction/data memory, etc. |
| Coding Efficiency | Shared variable, Pointer, Reference, Global variable, Cross-object call, Function pointer, etc. | Module state, Pipeline state, State lookup, etc. |
| Complex Operation | Multiply operator, Shift operator, Division operator, etc. | ALU, Memory address/data manipulation, etc. |
| Complex Control Flow | Non-bounded loop, Nested loop, Break, Continue, etc. | TLB, Cache lookup, Tag compare, etc. |

These high-level language features are problematic in synthesis for two reasons. First, it is difficult to determine statically what exactly the code and data to synthesize are. Second, there are no equivalent HDL constructs that they can be directly mapped to. We solve these problems by using a run-time synthesis technique to automatically produce VHDL [7] from SystemC. This technique offers accesses to in-memory simulator objects at runtime after elaboration and applies a set of dynamic and static analysis for code identification and optimization. During the synthesis, it interprets high-level language contructs based on the context where they are used and output VHDL code with equivalent functionality and timing.

### 1.1.3 Challenge 3: Software/Hardware Interface Synthesis

The software/hardware interface design is constrained by both partitioning and hardware synthesis capability. The software/hardware interface synthesis of hybrid structural microarchitectural simulators is challenging, because this process must automatiaclly produce both software interface code and hardware interface code for user-specified partitioning boundaries. Most importantly, the synthesized interface must synchronize processor state cross the software/hardware boundary correctly and efficiently. A poor interface would introduce extensive communication overhead or consume too much hardware resource. To develop a technique for the interface synthesis, three important traits regarding the interface design must be properly addressed.

1. **Correctness.** A usable interface for hybrid microarchitectural simulator must be correct. In another word, simulator states transferred between the software and the hardware must be correct, and the timing of state updates on both sides must be correct. Incorrect data and timing of simulator state synchronization will cause simulation failure or inaccurate performance prediction.

2. **Concurrency.** Interface concurrency is of great importance, because it reduces communication overhead by exploiting parallelsim in simulator execution and leads to the improvement of simulator performance. The coprocessor-style interface, also known as a polling interface, is the most commonly-used interface [8] for software/hardware co-design systems. Through such an interface, the software blocks to wait till the hardware finishes processing and polls the hardware results back. Due to lack of concurrency, a polling interface does not work efficiently for hybrid microarchitectural simulators where an extensive amount of dependencies exists between software modules and hardware modules. The execution of any one of them may require a synchronization request to cross the interface. To overcome these drawbacks, an optimized interface should overlap communication and computation and exploit not only the parallelism that is internal to the hardware but also the parallelism between the software and the hardware.

3. **Scalability.** Scalability is a significant concern when we are synthesizing interfaces for hybrid simulators. Interface scalability can be evaluated in two aspects: hardware cost and

bandwidth cost. Both hardware cost and bandwidth cost will rise, when the amount of information shared between the software and the hardware increases. The increased communication traffic leads to a higher interface overhead. Eventually the overhead will reach a point that it cannot be mitigated anymore by the speedup achieved through the hardware acceleration; then the hybrid simulation becomes slower than the software-only simulation. Every interface design has a bottleneck like this. It is necessary to understand the bottlenecks of different interfaces, because the insights can provide guidelines for architects on where to partition a software simulator in order to achieve desirable speedup but within their hardware budget.

As a matter of fact, various interface designs can be used for hybrid structural microarchitectural simulators, as long as they provides correct synchronization. The primary research questions we should answer are:

1. what interfaces can be applied for hybrid structural microarchitectural simulators;
2. how these interfaces can be automatically generated;
3. which one is better and why.

## 1.2 Research Objectives and Contribution

A good partitioning scheme, a complete synthesis strategy, and an efficient interface design combine to produce a high-performance hybrid simulator. The work of this thesis is the first effort to synthesize hybrid structural microarchitectural simulators, but this thesis primarily discusses how we solve the problem of interface design and synthesis for hybrid structural microarchitectural simulators. Our contributions are:

1. identifying the design space of interfaces for synthesized hybrid structural microarchitectural simulators.
2. providing synthesis techniques for several such interfaces in the design space.
3. determining the trade offs between simulator performance and hardware utilization which must be considered when choosing an interface design.

A complete infrastructure, named Simulate Partitioning Research Infrastructure (SPRI), has been also contributed to automatically produce hybrid structural microarchitectural simulators from existing software simulators. We demonstrate these contributions by implementing the synthesis of several such interfaces within the SPRI infrastructure. This capability provides an unique opportunity for this thesis to thoroughly analyze a set of synthesized hybrid structural microarchitectural simulators and evaluate scalability v.s. trade-offs between hardware capacity, interface latency, and interface bandwidth. The insights of this study are essential. They can lead to better decisions on how to organize a simulator, how to partition a simulator, and how to choose an interface, no matter whether architects are planning on applying SPRI to generate equivalent hybrid simulators or manually creating hybrid simulators from scratch.

## 1.3 Organization of the Dissertation

This thesis starts by introducing microarchitectural simulation and discussing the motivation for and issues of hybrid microarchitectural simulators. It then presents the SPRI infrastructure. It primarily focuses on discussing the design space of interfaces for hybrid structural microarchitectural simulators and the interface synthesis techqnues. The experimental results demonstrates the interface design trade-offs and the performance scalability for synthesized hybrid structural microarchitectural simulators.

Chapter 2 introduces the concept of microarchitectural modeling and the different modeling methodologies. It also discusses the acceleration methods for software microarchitectural simulator. Chapter 3 presents background information about hybrid microarchitectural simulators. It introduces several host platforms for hybrid simulation and examines the previous works of manually-created hybrid microarchitectural simulators. Chapter 4 displays the developed infrastructure, discusses the synthesis flow, and reveals the primary components of this tool-chain. Chapter 5 explores the interface design space, discusses the synthesis techniques for each interface in the design space, and conducts a thorough analysis and study on the interface evaluation results. Chapter 6 summarizes and concludes this thesis.

# Chapter 2

# Microarchitectural Simulation

Microarchitectural simulation allows architects to evaluate novel ideas without implementing a physical microprocessor. Microarchitectural simulators typically model both functional and timing behaviors of a microprocessor, and thereby are capable of predicting the performance. Like a hardware prototype, microarchitectural simulators execute processor instructions in simulation but they abstract target processor designs at microarchitectural level rather than register-transfer level or gate level. As a result, microarchitectural models implement fewer design details and can be created more rapidly than hardware prototypes, which significantly shortens the evaluation phase of target processor designs.

Microarchitectural simulators can be classified in two categories based on the inputs: trace-driven and execution-driven. A trace-driven simulator executes prerecorded streams of instructions with some fixed input and only needs to maintain the microarchitectural state. An execution-driven simulator executes instructions dynamically depending on the inputs and must track architectural state in addition to microarchitectural state. Execution-driven simulation can capture the dynamic properties of target benchmarks. The microarchitectural simulators discussed in this dissertation refer to execution-driven simulators. They run on a host platform and start simulation by loading a binary of target instructions.

## 2.1 Microarchitectural Simulator Design

A microarchitectural simulator is evaluated based on three criteria: speed, accuracy, and flexibility [9]. All three criteria are expressed as a triangle in Figure 2.1 and they can't be fully achieved at the same time in a simulator design. A good microarchitectural simulator design must balance the trade-offs between these three criteria. Hardware prototypes are accurate but slow and

not flexible; detailed software models are accurate and flexible but slow; abstract software models are fast and flexible but not accurate.

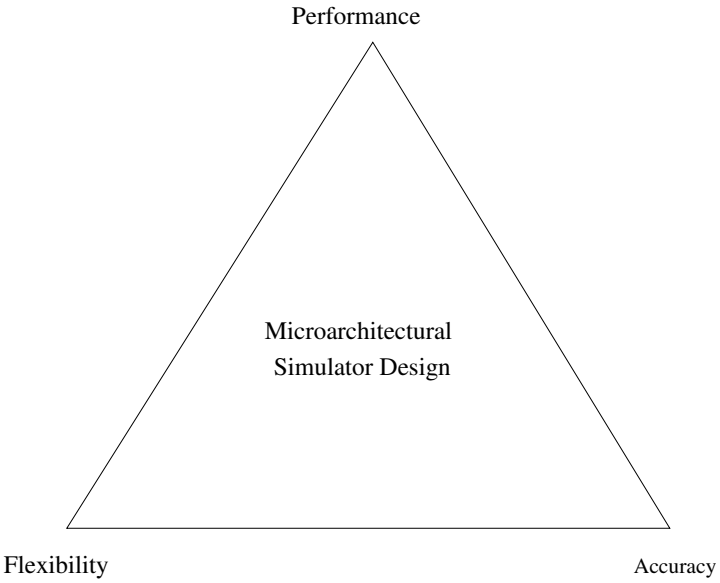
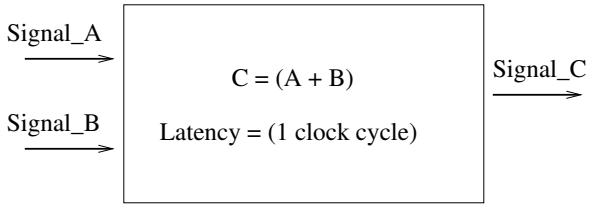
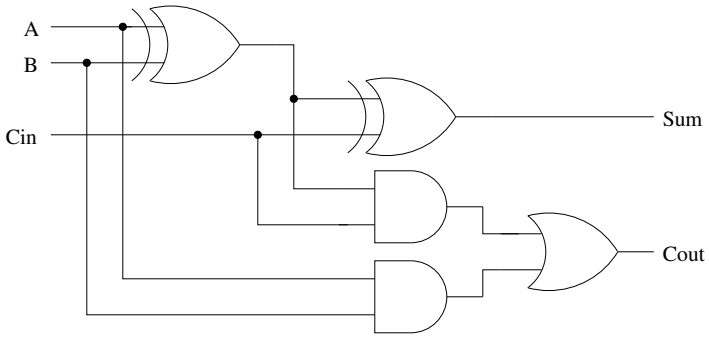



**Figure 2.1:** Trade-offs for microarchitectural simulator design [9]

Microarchitectural simulators are different from hardware prototypes. They are modeled at a higher-abstraction level in order to capture important processor behaviors but obtain simulation efficiency at the same time. Higher-level abstraction in a simulator design is desired to improve clarity and speed. Because a microprocessor is a timed logic design, what matters to the correctness of microarchitectural models is the value changes of signals at each clock edge. This fact makes modeling within-clock-cycle behaviors for each wire of a microprocessor unnecessary [10]. Figure 2.2 diagrams a adder example to demonstrate the difference between modeling and hardware implementation. A 1-bit adder model can be simply be created as an addition operation on two integers with a latency rather than a collection of gate-level logic blocks. The microarchitectural model effectively expresses the timing and the functionality of this operation and can be easily created with high-level computer languages.




(a) Microarchitectural Model of One−bit Adder




(b) Gate−level Implementation of One−bit Adder

**Figure 2.2:** Example: model and implementation of 1-bit adder

### 2.1.1 Design Tools

Microarchitectural simulators are commonly written as software models. Software models run on servers or workstations, and can be easily programed or reprogrammed for design adjustment. Software modeling, therefore, offers more flexibility than hardware does. Sequential languages such as C and C++ [1] are widely applied in microarchitectural modeling, because architects are very familiar with them. Sequential languages semantically mismatch the structural and concurrent form of target systems which leads to increased simulator design and validation time as well as inflexibility . It is possible to model concurrent behaviors through a global scheduler which manages a sequential execution order for a logically-concurrent simulator, however, it results in a more complex simulator which increases the difficulty of understanding and debugging the simulator code [11]. The modeling complexity with sequential languages grows extensively when target microarchitectures scale to many cores and increased system integration.

The limitation of sequential languages has led architects to use structural simulation frameworks (e.g. SystemC [2], Unisim [3], LSE [4]) for modeling. These frameworks provide users with structural and concurrent constructs to create models that naturally mimic hardware behav-

iors. Structural microarchitectural simulators can be created in a hierarchical fashion through simulator modules. A simulator module typically consists of one or several concurrent processes. The processes in the same module or different modules can be interconnected through signals and they are invoked to produce outputs when clock edges arrive or input changes are detected. Structural simulation frameworks, unfortunately, have to trade off simulator performance for the modeling productivity [12]. A significant amount of overhead is introduced in the execution of structural simulators, because the simulation kernels must maintain signal updates, invocation sequences, etc. Most structural simulation frameworks also allow the use of object polymorphism and non-structural constructs (e.g. shared/global data structure, cross-object call) to permit code reuse, increase modeling flexibility, and improve simulation speed. As a result, structural microarchitectural models are usually formed from a wide range of high-level language constructs including both structural and non-structural ones.

### 2.1.2 Design Organization

A microarchtectural simulator primarily models two aspects of a target microprocessor: functionality and timing. Functionality refers to the execution correctness of target instructions and timing strives to reproduce the execution latency of target instructions at each processing stage. The organization of a microarchitectural simulator reflects how the simulation of functionality and the simulation of timing interact with each other. Commonly, microarchitectural simulators can be classified in two base organizations as displayed in Figure 2.3: integrated and decoupled. Different simulator organizations emphasize different design goals: accuracy, speed, or flexibility.

A microarchitectural simulator can be written in a integrated fashion as a phyical microprocessor design where functionality and timing are tightly coupled. Microarchitectural simulators designed in this fashion can be very accurate, but at the cost of design flexibility. Integrated simulator allows the modeling of detailed operations for all system components. These system components closely interact with each other and simulation states of each module must be frequently updated. Integrated microarchitectural simulators will become more and more difficult to design and debug with the increasing complexity of target systems.

A microarchitectural simulator can also be organized in a decoupled fashion. The simulator is split into a functional portion and a timing portion and they communicate with each other for

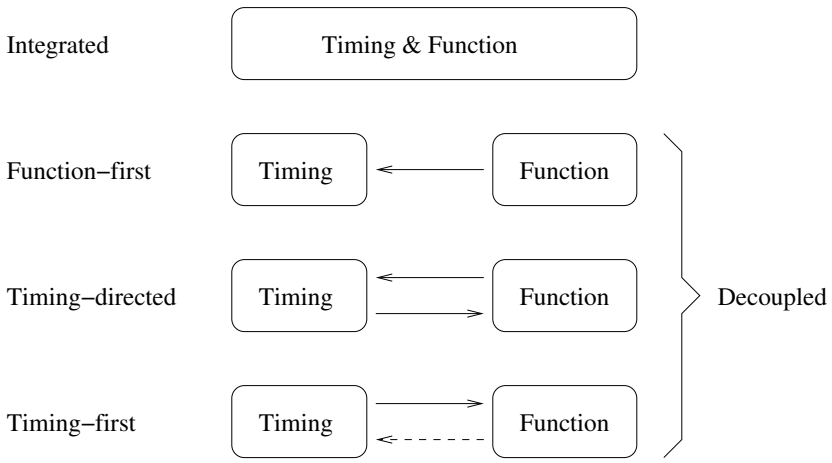



**Figure 2.3:** Microarchitectural simulator organization [13]

synchronization. Doing so addresses functional accuracy and timing accuracy in separate code and reduces modeling complexity. The functional model simulates each target instruction by executing an equivalent code-routine on the host, the process of which is therefore named instruction-set simulation. The timing portion models timing-related components (e.g. pipeline, memory hierarchy, branch predictor) of the target microprocessor to calculate execution latency for each target instruction through microarchitectual simulation. Based on different synchronization mechanism between the functional model and the timing model, decoupled simulators can be categorized as functional-first, timing-directed, or timing-first. In a functional-first simulator, the functional model generates a sequence of committed target instructions and feeds it to the timing model. One-way communication from the functional model to the timing model makes the simulator easy to design and fast to execute; however, this fashion can not capture timing-dependent outcomes like branch misprediction. A type of functional-first simulator — the speculative functional-first simulator is introduced to solve this problem [14]: the functional model speculatively executes instructions and is rolled back to re-execute only if the branch prediction of the functional model does not match that of the timing model. In a timing-directed simulator, the timing model gives explicit orders to the function model at certain timestamps on what behaviors to simulate (e.g. instruction fetch, instruction decode, and instruction commit). This timing-directed fashion increases simulator accuracy at the cost of simulator speed and complexity. A timing-first simulator, on the other hand, allows the timing-model to execute timing-dependent (dynamic) instructions ahead of the function portion.

The timing portion must be modeled with enough microarchitectural detail to support speculative execution and the functional portion will be invoked to check for execution deviations. A timing-first simulator can also be thought of as a near-complete integrated microarchitectural simulator plus a functional checker.

The split of the functional model and the timing model simplifies dependency between simulator modules and reduces modeling complexity. There is no absolute answer to which organization is the best, however, the choice of simulator organization does reflect architects' emphasis on accuracy, speed, and flexibility.

## 2.2 Microarchitectural Simulator Acceleration

Microarchitectural simulation often takes days or weeks to run in software. The faster simulation runs, the more design alternatives can be evaluated. Software microarchitectural modeling permits a fast and flexible evaluation process for new computer designs, however, the simulation speed increasingly slows down when target computer systems scale to many-core or increased system-integration designs. A significant amount of effort has been devoted to improving the performance of microarchitectural simulators. Software microarchitectural simulators can be accelerated typically in two ways, either by reducing simulator workloads or by pursuing execution parallelism.

Two techniques that can effectively reduce simulator workloads are input-stream shortening [15, 15] or sampling simulation [16, 17, 18, 19]. Input stream shortening leads simulators with fewer or smaller input sets instead of complete benchmarks. By contrast, a simulator that uses simulation sampling still executes a full benchmark, but it only cycle-accurately simulates samples of this benchmark and the instructions between those samples are fast-forwarded through functional simulation. The simulation-sampling technique usually requires simulators to have two simulation modes — a functional-simulation mode and a cycle-accurate model to switch during simulation. The simulators that use these workload-reduction techniques may not fully characterize the significance of the chosen benchmarks through the executed instructions, because they can not completely enumerate all the simulation states. Thus, the simulation speedup is achieved by sacrificing simulation accuracy.

Another method to accelerate microarchitectural simulation is to parallelize simulators (simulator parallelism). A microarchitectural simulator can be parallelized because it describes hardware behaviors that are considered naturally and logically concurrent. This concurrency can be internal to simulated modules, between simulated modules or between simulated cores. The research community of computer architecture has proposed two approaches to parallelize microarchitectural simulators: software parallelization and hardware parallelization. Software parallelization decomposes a simulator into multiple tasks that can be run in parallel. Hardware parallelization exploit finer-grain (gate-level) parallelism to speed up simulation by directly implementing simulators in circuits.

### 2.2.1 Software Parallelization

A software microarchitectural simulator is parallelized by distributing decomposed simulator tasks to execute on different physical processors. The techniques of software paralellization can be categorized based on the granularity of the decomposed tasks.

- Microarchitectural Parallelization: each simulated processor core is mapped to a different physical processor for simulation and a separate processor is used to simulate the memory hierarchy [20, 21].
- Simulation Trace Parallelization: several copies of a simulator are run in parallel; a instruction trace is divided into equal-length chunks and each chunk is fed into one of the parallel simulators [22].
- Structural Parallelization: a structural simulator can be partitioned into a set of structural modules. This simulator is thereby transformed to a parallel program. The parallel program can be compiled and statically scheduled to run on a multiprocessor host platform with a shared memory [10, 23, 24].

Microarchitectural parallelization and simulation trace parallelization partition simulators into "naturally parallel" tasks which are limited in interacting with each other. They either ignore the communication between parallel portions or utilize a simplified scheme to reduce synchronization complexity and overhead. Both methods may cause incorrect simulation results due to the lack of state consistency between parallel portions.

Structural parallelization adopts a different approach. The method decomposes a structural simulator into a set of structural modules and relies on compiler techniques to find a fixed concurrent execution order for them. The discovered execution order must not change with input data. As a matter of fact, the simulator is transformed into a statically-compiled program that can be scheduled based on module dependency to run in parallel on a host multiprocessor. The simulation performance depends on the quality of the discovered schedule, and the performance improvement can be achieved without losing accuracy. Unfortunately, microarchitectural models tend to have an extensive amount of data and control dependency between simulator modules. The difficulty in finding an efficient static schedule grows dramatically with the increasing size and complexity of target computer systems. Finer-grain parallelism must be pursued to permit further performance enhancement.

### 2.2.2 Hardware Parallelization

Instead of being implemented in software, a microarchitectural simulator can be designed directly with circuits in hardware. The hardware version of the simulator exploits finer-grain (gate-level) parallelism to accelerate simulation speed. A hardware-accelerated simulator can be significantly faster than a software simulator for two reasons:

1. Different modules of the software-only simulator can be executed in parallel in hardware. This is particularly true because the software is modeling hardware that is inherently parallel [25].

2. Individual modules of a software-only simulator become significantly faster when implemented in hardware. Several hundred software instructions can easily become one or two hardware cycles worth of work. Although hardware (i.e. FPGA) frequency is significantly lower than host CPU frequency, this speedup still occurs because overhead caused by instruction fetching and decoding is no longer necessary, and because fine-grain parallelism within a module can be exploited.

Two approaches have been proposed for hardware parallelization. The first approach [26] is to implement a microarchitectural simulator fully in hardware. The design process of a full-hardware microarchitectural model that can accurately simulate both functional and timing behav-

iors is difficult and time-consuming. It ends up implementing a multithreaded pipeline in hardware to execute target instructions issued from multiple cores, the design complexity of which is similar to a hardware prototype. The second approach [25, 14, 27] is a hybrid microarchitectural model where a portion of the simulator is moved to hardware, the other portion stays in software, and they communicate through a software/hardware interface. Hybrid simulators allow architects to select what to move into hardware based on simulator organization and hardware capacity. A hybrid simulator design scales better and offers more flexibility than a full-hardware design when target computer systems become increasingly complex. In a hybrid simulator, architects can simply keep the hardware-unfriendly portion in software and avoid spending a great amount of effort on hardware implementation. They can also mitigate the constraint of hardware capacity by reducing the size of the hardware partition at the cost of simulation performance. The design details of hybrid microarchitectural simulators are discussed in the next chapter.

# Chapter 3

# Hybrid Microarchitectural Simulation

An effective technique to accelerate software microarchitectural simulators is hardware parallelization. Hardware parallelization allows the pursuit of parallelism at a finer-granularity level through circuits and the circuit implementation only takes one or two hardware cycles to accomplish the amount of work that must require several hundred software instructions. The concurrency internal to simulated modules, between simulated modules or between simulated cores can all be explored to speed up simulation.

One group of hardware-accelerated microarchitectural simulators only implement portions of microarchitectural models in hardware, and therefore are named hybrid microarchitectural simulators. Hybrid microarchitectural simulators are very promising, because they allow trade-offs to be made between simulator speed, ease of implementation, and hardware resources. A hybrid microarchitectural simulator is composed of a software portion, a hardware portion, and a software/hardware interface for synchronization. It runs on a host platform that has a hardware accelerator attached to a computer workstation through a physical communication channel. The software portion is compiled and executed on the general-purpose processor of the computer workstation; the hardware portion is implemented as a circuit and ported to the hardware accelerator. The performance of a hybrid simulator is greatly influenced by the host platform in three aspects: hardware capacity, communication latency, and communication bandwidth. Hardware capacity decides the maximum amount of the moved-to-hardware portion; communication latency and bandwidth together set the minimum interface overhead for a hybrid simulator design.

## 3.1 Host Platform for Hybrid Simulation

Hybrid microarchitectural simulators must run on a host platform which consists of a general-purpose processor for the software portion, a hardware accelerator for the hardware por-

tion, and an communication channel for the software/hardware interface. Because the hardware accelerator and the communication channel significantly affect the performance of hybrid simulation, an ideal host platform must fulfill several requirements.

The hardware accelerator must allow the pursuit of fine-grain parallelism (i.e gate-level). Fine-grain parallelism fundamentally improves simulator performance by avoiding the costly execution process of host instructions in software. The hardware accelerators commonly used for data-intensive applications (i.e. GPU) are typically not suitable for hybrid microarchitectural simulators, because frequent synchronization between parallel threads serializes the execution. The hardware accelerator must also be flexible to "program" and "reprogram" similarly to software. This feature is necessary, because the architecture of a microprocessor under consideration may require a lot of adjustments which must be easily added for simulation during the evaluation phase.

The communication channel between the host processor and the accelerator must be efficient enough that the software/hardware interface built on top of it will not introduce an enormous overhead to overshadow the speedup obtained through the hardware accelerator. Both low latency and high bandwidth are desired for the communication channel. Lower latency shortens the round-trip cost for data transmission; higher bandwidth means a faster data-transfer rate so that a larger amount of data can be sent across the interface in a given time period.

Additional attention should be paid to the capacity of the hardware accelerator. A large hardware capacity allows more simulator modules to be implemented in hardware, and leads to a higher speedup. In fact, only one hardware accelerator may not be enough when the target computer system scales to a design with tens or hundreds of cores. The hardware extensibility of a host platform must be considered so that multiple accelerators may be stacked up in a host platform to expand the capacity.

### 3.1.1 FPGA

Field Programmable Gate Arrays (FPGAs) are commonly-used hardware accelerators for hybrid microarchitectural simulation. An FPGA is a integrated circuit which allows logic configuration and reconfiguration after the chip has been manufactured [28]. FPGAs allow designers to improve simulator speed through circuit-level parallelism and to reconfigure circuit functionality based on microarchitectural changes. Like software, the design flexibility of FPGA offers the po-

tential to simply build and test design candidates repeatedly, thus resulting in a low nonrecurring engineering cost which well suits the design requirement of microarchitectural simulators.

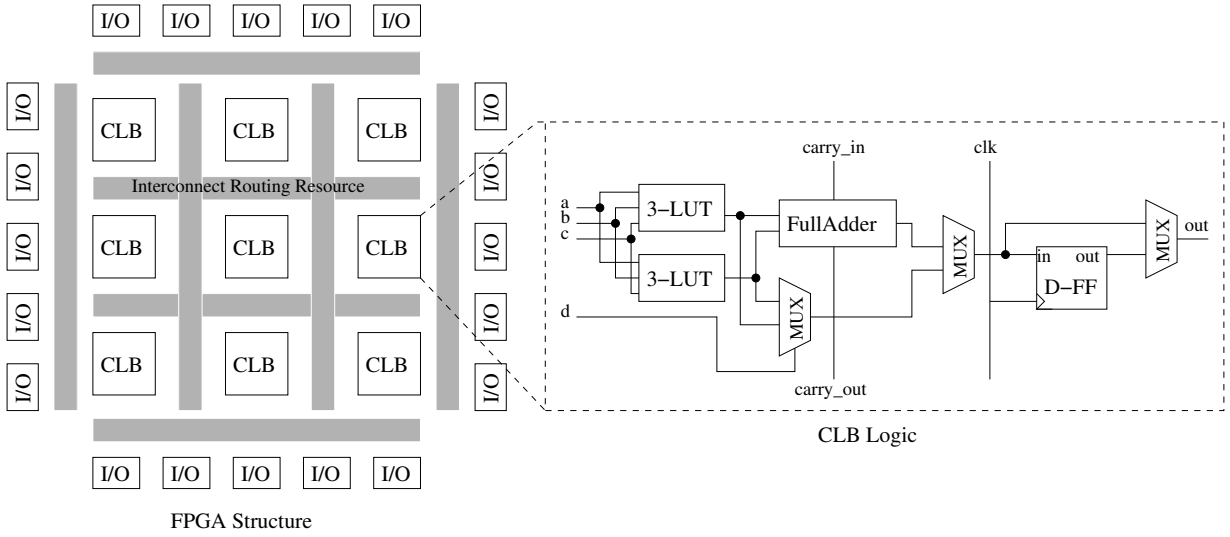
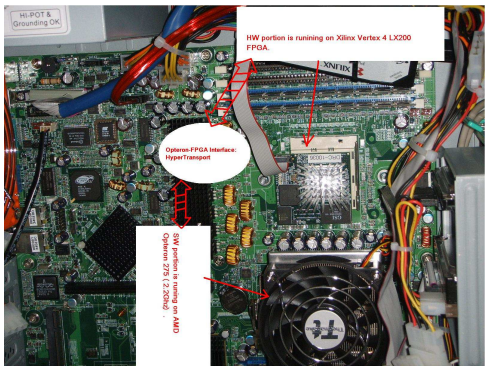



**Figure 3.1:** Simplified FPGA structure and CLB logic

The basic FPGA architecture consists of configurable logic blocks (CLBs), interconnect routing resources, and I/O ports. A simplified FPGA architecture and CLB logic are shown in Figure 3.1. CLBs can be programmed with different logic functionality because of the built-in programmable units called lookup tables (LUTs). The computed result of each CLB can be registered by a D-type flip-flop and carried through other CLBs by customizing the routing connection with switches. CLBs and routing resource are distributed across the whole FPGA chip, thereby permitting the implementation of millions of operations spatially and simultaneously [29]. FPGAs can be programed using hardware description languages (HDLs) such as Verilog [30] and VHDL [7]. FPGA companies (e.g. Xilinx, Altera) and the third party EDA companies (e.g. Synopsys, Cadence) both provide sophisticated software development environments for FPGA development and simulation. Bitsteam files that describe circuit connection and functionality can be produced by those tools and downloaded to FPGAs for configuration and reconfiguration.

(a) XUPv5 Board


(b) BEE3 Board




(c) DRC-1000 System

(d) ACP System

**Figure 3.2:** Several host simulation platforms

### 3.1.2 FPGA-based Host Platform

An FPGA-based host platform typically consists of a computer workstation and one or more FPGA accelerators. FPGA accelerators are connected to the computer workstation through a communication channel. The interconnection mechanism between FPGA and general-purpose processor determines the hardware extensibility (hardware capacity) and the communication performance (latency and bandwidth) which both significantly affect the design and efficiency of hybrid microarchitectural simulators. In this dissertation, FPGA-based host platforms are classified based on the interconnection mechanism: network connection, I/O connection, and CPU socket-level bus connection. Typically, network and I/O connects standalone FPGA boards (i.e.

Figure 3.2(a), Figure 3.2(b)) to the host; CPU socket-level bus connections are used in integrated systems (i.e. Figure 3.2(c), Figure 3.2(d)) to directly interface FPGAs with CPUs.

1. **Network Connection** Network connection allows workstations to interconnect in a sparse mesh fashion through network cables. Network connection offers great extendability, however, has relatively low bandwith and high latency. A commonly-used network connection is Ethernet connection. A standalone FPGA board typically provides one or more Ethernet ports for off board communication. In a Ethernet network, hundreds of FPGA boards can communicate with each other or the host. Typically, the Ethernet latency from a FPGA board to the host is in the range of hundreds of micro-seconds per round-trip, and the maximum bandwidth is commonly in the range of 10Mbs – 1Gbs depending on the types of Ethernet network and the used Ethernet devices. Ethernet relies on complex protocols to provide reliable connections between workstations, but the complex transmission protocols introduce a large amount of processing overhead. Thus, an efficient interface between FPGAs and workstations for hybrid microarchitectural simulators can be constructed as a local-area network (LAN) running a simplified version of TCP/IP in order to reduce communication latency and overhead.

2. **I/O Bus Connection** Computer I/O buses provide communication channels between the computer host and peripherals. FPGAs can be considered as peripherals and connected to the host computer through computer I/Os. I/O sockets are placed on the motherboard of the host and peripherals can be directly plugged in. When compared with network connection, I/O connection has higher bandwidth and lower latency, but only a limited number of peripherals can be connected depending on available I/O slots. PCIe is one of the fast computer I/Os. The fastest version (v4.0) of PCIe connection permits a bandwidth up to 1969 MBs per lane (one direction). Most standalone FPGA boards provide PCIe ports to connect with host computers. The PCIe communication is typically memory-mapped and the communication protocol is significantly less complicated than TCP/IP. We measured the PCIe performance of a single PCI lane between a XUP board and a workstation. From the stand point of CPU, the bandwidth is 126 MB/s read and 27.9 MB/s write for 4 KByte transfers; the read latency is 1.6 micro-second minimum.

3. **CPU Socket-level Bus Connection** CPU socket-level buses are the communication channel to provide high-speed data transaction between CPUs and system memories. Different CPU manufacturers have different names for it. Intel names it the front-side bus (FSB); AMD calls it the HyperTransport bus. FPGA-based accelerators can be hooked up as coprocessors on CPU socket-level bus to directly communicate CPUs and access system memories. This bus connection has the shortest latency, but the extendability is very limited. The CPU socket-level bus can run faster than the I/O buses such as PCIe because the clock of the secondary buses is usually derived from the clock of CPU socket-level buses. We measured the HyperTransport performance on a DRC-1000 workstation. From the stand point of CPU, the bandwidth is 224 MB/s read and 354 MB/s write for 4 KByte transfers; the read latency is 1.4 micro-second minimum.

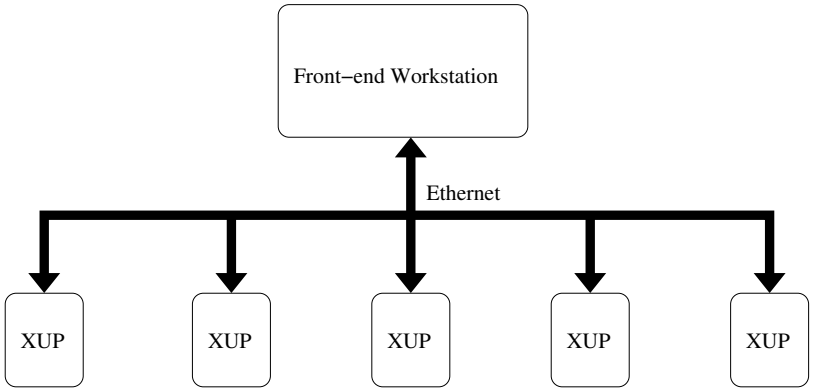



**Figure 3.3:** A high-level view of an XUP-based platform (i.e RAMP Gold)

Figure 3.2 shows several commonly used platforms in hybrid microarchitectural simulation. XUP [31] and BEE [32] are standalone FPGA accelerators. Both of them can be connected to host workstations through either ethernet or PCIe. DRC [33] and ACP [34] are integrated systems where FPGAs are hooked up via the CPU socket-level buses.

XUP is a single FPGA board and priced less than $1000 for academic purchase. It is an affordable and scalable platform solution for hybrid microarchitectural simulation. The RAMP Gold simulator [26] runs on five XUP boards. These boards are not interconnected with each other but they are attached to the same front-end workstation through ethernet. A high-level view of this

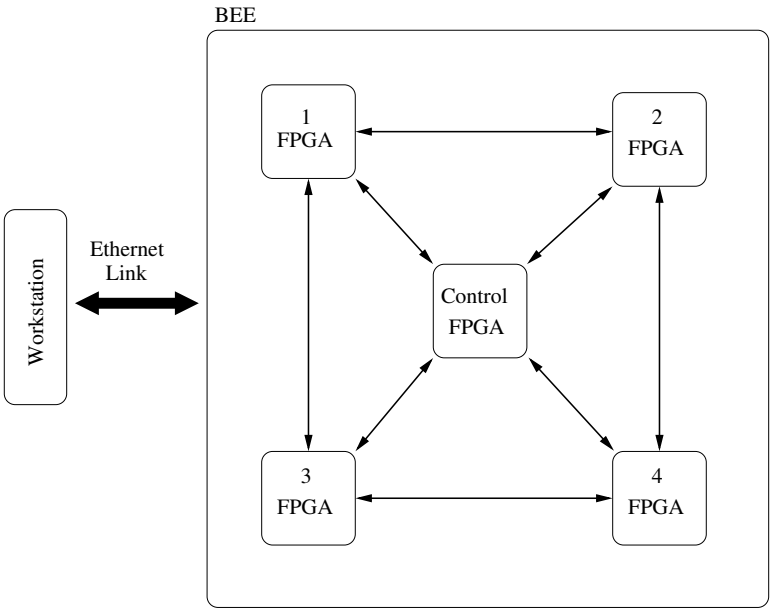



**Figure 3.4:** A high-level view of BEE-based platform

host system is shown in Figure 3.3. Because the ethernet connection is relatively slow, people who adopts a similar organization for the host platform typically tend to move simulator modules to the hardware as much as possible in order to minimize off-board communication overhead. The RAMP Gold simulator can be ported to BEE. This board consists of five interconnected FPGAs: one FPGA is configured as a controller and the other four are processing units as in Figure 3.4. BEE has much higher hardware capacity than XUP, but at a much higher price (over $10,000).

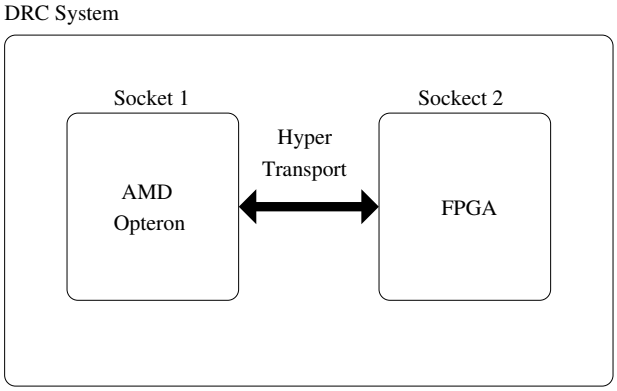



**Figure 3.5:** A high-level view of DRC

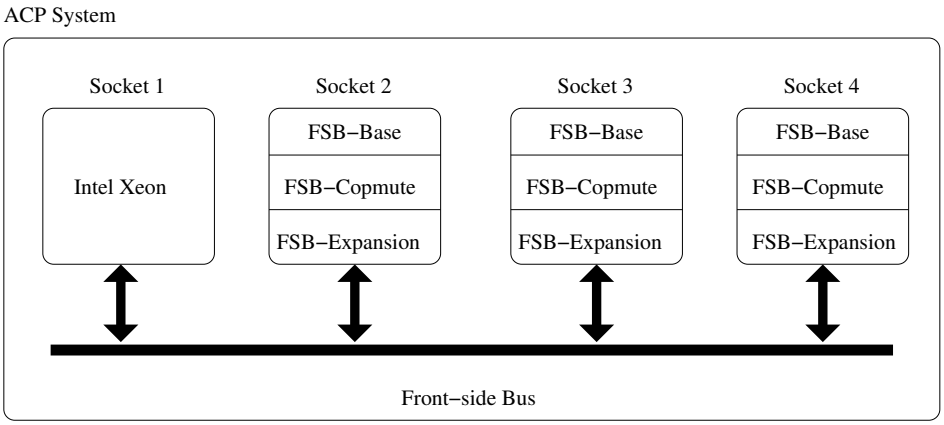



**Figure 3.6:** A high-level view of ACP

DRC and ACP provide a complete platform solution for hybrid microarchitectural simulation. DRC and ACP have FPGA accelerator modules directly plugged to additional CPU sockets as coprocessors. Most FPGA-integrated systems come with device APIs and pre-synthesized FPGA IPs to set up the communication between the host processor and FPGAs. FPGA-integrated systems have limited extendability, because only a small number of FPGA-accelerator modules can be added on board. DRC is a AMD-chip-based system. This system, as shown in Figure 3.5, has a two-socket Opteron server with one Opteron core replaced by an FPGA . The Opteron core and the FPGA are connected through the HyperTransport bus and no FPGA extension is allowed. ACP, by contrast, is a four-socket Intel server. ACP allows FPGAs to directly access system memory, but DRC does not. Users can populate all 4 sockets with different combinations of FPGA accelerators and Intel Xeon processors. Up to five FPGAs can be stacked up within a single socket and up to 3 sockets can be customized with FPGA accelerators. Nallatech [34] has been licensed to manufacture three modules for the ACP platform: FSB-base, FSB-compute, FSB-expansion. FSB-base is the interface board between the Intel front-side bus and FPGAs. FSB-compute is the FPGA accelerator board with two FPGAs and FSB-expansion is the accelerator extension board with only one FPGA. A high-level view of this platform is diagrammed in Figure 3.6. The UT-FAST hybrid simulator [14] was implemented first on a DRC-1000 system. The software portion of the simulator runs on the Opteron core and the hardware portion is implemented on the FPGA. The researchers of this project attempted to transplant the hybrid design to a ACP system, but reliability issues in

the ACP hardware caused data corruption during transfer and prevented them from running and measuring the performance of the entire system [35].

## 3.2 Previous Hybrid Microarchitectural Simulators

Three types of hybrid microarchitectural simulators have been advocated. In a hybrid function-timing simulator, the functional behavior of instructions is performed in software while the time it takes to execute an instruction is calculated in hardware. In a hybrid transplanting simulator, most behaviors of the simulator are implemented in hardware, but when a difficult-to-implement operation such as simulated I/O device behavior must be performed, the hardware calls upon the software to complete the operation. In a hybrid structural simulator, a collection of processes from a structural simulation model are implemented in hardware. The three hybrid microarchitectural simulators are partitioned differently which leads to different interface strategies.

### 3.2.1 Hybrid Function-Timing Simulator

Function-timing simulators are split into a functional portion and a timing portion. Simulation of instruction behavior and timing behavior are decoupled in this process and they must synchronize in some way to keep simulation states consistent on both sides. Several software-only simulators such as FastSim [36], Asim [37], and M5 [38] are partitioned in this fashion.

The UT-FAST project [14] has created a hybrid function-timing simulator with the function model in software and the timing model in hardware. The UT-Fast simulator is implemented on the DRC-1000 System. The functional portion runs on the Opteron core of the DRC box and simulates the instruction-set architecture and system calls. The timing portion is implemented on FPGA and models detailed actions of different pipeline stages to calculate execution latency. This strategy allows the software to run first and speculatively execute instructions; it feeds the hardware with instructions that has been executed; the hardware can roll back the software to re-execute instructions, if the software branch predictor and the hardware branch predictor mismatch each other. Thus, the cross-boundary interface is constructed in a asynchronous fashion using a commercial FPGA message passing interface (MPI) library; the software and the hardware communicate through asynchronous queues to provide execution concurrency. The average number

of round-trip communications is reduced to less than one per simulated clock cycle. The simulation speed is around 1.2 MIPS. UT-FAST didn't implement a baseline software simulator to compare against with so that it is impossible to evaluate the performance increase caused by using the hardware accelerator.

HAsim [39] is a hybrid version of Asim which is internally used in Intel. Asim is a function-timing split simulator where the timing portion is build in a structural fashion. The functional and timing models of HAsim are closely coupled. The timing model explicitly requests that the functional model executes an instruction at the corresponding simulated time stamp . This organization results in a high degree of communication between these two partitions. In order to loose the timing constraints on the interface, HAsim applies a method named A-Port in the simulator design [40]. The A-Port scheme is similar to FIFO. Modules can be connected through A-Port, and the global synchronization is localized to enqueue and dequeue operations of FIFOs. This project has demonstrated its scaling capability of constructing many-core simulators on a single FPGA (up to 19 target cores). The overall simulation performance peaks at 3.2 MHz, but no performance comparison against an equivalent software-only simulator has been reported.

### 3.2.2 Hybrid Transplanting Simulator

The ProtoFlex system [27] adopts an alternative approach that combines software simulation with FPGA-accelerated instrumentation. This simulator currently only supports functional simulation, but uses a different partitioning strategy. This system implements a processor in hardware to simulate an instruction-set architecture. This instruction emulation engine is implemented on a FPGA, but it is not complete and some complex and rare operations and conditions (e.g. system I/O, FP operations, and certain traps) are handled in software by invoking either a host processor or FPGA PowerPC core with necessary state transfer upon request. Multiple processor contexts are supported to allow multiprocessor functional simulation. This simulator can achieve a simulation throughput as high as 62 MIPS, but only simulating instructions. ProtoFlex didn't provide a equivalent software simulator for comparison either.

### 3.2.3 Hybrid Structural Simulator

The Liberty project at Princeton has demonstrated the feasibility of a hybrid simulation approach based upon a structural partitioning strategy in [25]. This project uses the LSE framework to create a software-only simulator in a concurrent and structural form [41]. Later on, the processor core in the created chip multiprocessor simulator are replaced with the physical PowerPC cores on FPGAs. Both functionality and timing of the processor cores are simulated in hardware and the memory architecture is simulated in software. The FPGA PowerPC cores are wrapped with a communication interface; the interface transfers data with the software portion through a adapter module that calls the device drivers. The interface implements a polling scheme and makes no attempt to optimize the communication between software and hardware: the software requests the hardware to execute and polls for hardware completion. This simulator is parallelized at a coarse-granularity level through four physical PowerPC cores on the same FPGA. This hybrid design is constrained by the number and performance of available FPGA PowerPC cores. This hybrid simulator achieves up to 5.82 speedup, compared with the corresponding software-only simulator. The performance comparison between the hybrid simulator and the software simulator is not accurate. The hybrid simulator uses physical PowerPC cores, however, the baseline simulator is a simplified microprocessor model so that the achieved simulation speedup is much less than it would be with a detailed baseline simulator.

### 3.2.4 Discussion

Several hybrid microarchitectural simulators have been introduced in different categories. It is impossible to compare their simulation speed with each other, because those simulators models different processors at different abstraction levels and they are organized and partitioned differently. Most of the hybrid simulator designs didn't provide a baseline software-only simulator to compare with so that we are not able to investigate the impact of the hardware acceleration and the interface overhead on the simulation performance. In addition, all the previous hybrid simulators are created manually, so it is very time-consuming to design and debug the hardware portion and the interface. The inefficient design process prohibits a widespread use of hybrid microarchitectural simulators for evaluating microprocessor candidates in both academia and industry.

The third category, hybrid structural microarchitectural simulator, is particularly interesting. It combines the benefits of structural modeling and hardware acceleration. The design flow starts with a software model created under a structural simulation framework, and portions of the software structural model are replaced with equivalent hardware components for acceleration. The performance loss of structural simulation can be recovered with the use of FPGAs. When the transformation from software structural models to hybrid simulators is automated, this category of hybrid simulators does have the potential to be easily created and efficiently executed.

# Chapter 4

# SPRI: Simulator Partitioning Research Infrastructure

Hybrid structural microarchitectural simulators combine the benefits of structural modeling and hardware acceleration in a flexible fashion that architects are able to make trade-offs between hardware capacity, ease of implementation, and simulator performance. Unfortunately, it is very time-consuming to create a hybrid structural microarchitectural simulator by hand. Architects have to plan a software/hardware partitioning, design both the software and the hardware, and create a software/hardware interface. The manual design process become increasingly prohibitive, when target systems scale to many cores and higher system integration. To promote a widespread use of hybrid structural microarchitectural simulators, we have proposed to automate the design process through synthesis. This chapter introduces the synthesis flow for hybrid structural microarchitectural simulators by describing the SPRI infrastructure in which the contributed techniques have been implemented and integrated.

SPRI automatically synthesizes hybrid structural microarchitectural simulators from software structural microarchitectural models. The input models must have been developed under a structural simulation framework before being passed to SPRI. The output simulator consists of a software portion, a hardware portion, and a software/hardware interface. The executable binary of the software portion runs on the host processor; the hardware portion is downloaded to the FPGA after being synthesized to gates. The design efforts of architects are only dedicated to implementing the input structural microarchitectural models; no extra effort is needed to transform them to the equivalent hybrid simulators. SPRI allows users to guide the partitioning and the interface selection depending on how they organize their simulators. Given a software simulator as input to SPRI, a set of equivalent hybrid simulators can be rapidly generated with different selection of partitionings and interfaces.

## 4.1 SPRI Input

SPRI has two inputs: SystemC microarchitectural models and partitioning specifications. The partitioning specifications guide SPRI to move portions of the input software simulators into hardware. The input simulator must have been developed and debugged; the users of SPRI must be responsible for its correctness and accuracy.

Where to partition a SystemC microarchitectural simulator is of great importance in constructing a hybrid structural simulator. The simulator partitioning determines what to synthesize and how to interface in later processing. Thus, the hardware cost and the hybrid-simulation speed may vary depending on the chosen partitioning boundary. SPRI takes partitioning instructions from the users and finishes the partitioning of the input SystemC microarchitectural models automatically. This feature offers the users the flexibility to choose the partitioning boundary based on the hardware capacity and the input simulator's organization. Because there is no one partitioning that works effectively for every simulator, the users are in charge of exploring partitioning options and selecting the one that leads to the best hybrid simulator.

### 4.1.1 SystemC

There are several structural simulation frameworks (e.g. Liberty, Unisim, and SystemC) that can be used to develop microarchitectural models in software. SystemC microarchitectural models are chosen as input to SPRI particularly, because SystemC has several important features.

- SystemC is an additional library attached to C++, including a set of C++ classes and macros. This feature enables architects to simulate concurrent hardware processes using plain C++ syntax which they are familiar and comfortable to work with.

- SystemC not only offers a rich amount of structural and concurrent constructs (e.g. process, signal, event) to express model hierarchy and timing, but also has non-structural features (e.g shared/global variable, cross-object call) and object polymorphism inherited from C++ to enhance modeling productivity and permit high-level code reuse.

- SystemC allows modeling at different abstraction levels (e.g. behavior level, RTL) or with a mix of several. This is important for microarchitectural modeling because detailed models

may be needed for a portion of a microarchtectural model that architects want to thoroughly evaluate and less details may be described for the other portion in order to reduce modeling complexity.

- Sophisticated SystemC simulation environments and debugging tools (e.g. CoCentric from Synopsys, ModelSim from Mentor Graphics, OSCI SystemC Simulator) have been developed for years to facilitate software modeling.

SystemC [2] is an event-driven simulation framework; as events occur in the system, processes which are sensitive to those events are invoked. In a typical detailed microarchitectural model where only cycle-level accuracy matters, the only events are signals changing values. Processes in these models can be divided into those that are sensitive to clock edges and those that are not.[1] Typically, such processes can be triggered in three ways: immediate notification, delta notification, and timed notification. Immediate notification is processed first at the beginning of each simulation time-stamp. Immediate notification occurs when a process is sensitive to events fired using the `event.notify()` method; there is no time duration between the invocations of the current process and the previous process. Delta notification, by contrast, introduces a non-zero (delta) duration between process invocations; a simulation clock cycle could have any number of delta-time intervals. Delta notification occurs when updating the values of process-sensitive signals or using the `event.notify(SC_ZERO_TIME)` method. When processes assign values to signals, the assignment does not take place until after a delta cycle – a cycle with zero imputed time – has taken place. During simulation, delta cycles occur as long as events are generated by signal assignments or `event.notify(SC_ZERO_TIME)`; once no more such delta events occur, time advances to the next timed event which in a cycle-level model will be a clock edge. SystemC also allows users to trigger processes to run after a certain amount of simulated time using `event.notify(t)`.

SystemC uses a multiple-list event handling implementation, described in the pseudo-code in Figure 4.1. A simulation time-step begins by running all processes on the runnable list. As they run, they push immediate-notified processes back to the runnable list and assign values to signals; these value changes are enqueued on an update queue. After there are no more processes to run,

[1] In SystemC, processes are known as methods or clocked threads. The other type of process (a thread) may wait on events within the body of the process; it may be transformed into a clocked thread if only clock edges are waited upon.

```
1  do {
2    do { // delta cycle loop
3      foreach process in runnable list
4        dequeue and execute process
5        add immediate-notified processes to runnable list
6      foreach update in update list
7        dequeue and perform update
8      foreach event in delta event list
9        dequeue event
10       add delta-notified processes to runnable list
11   } while runnable list not empty
12
13   advance time to minimum time in timed event list
14   foreach event in timed event list at the new time
15       dequeue event
16       add sensitive processes to runnable list
17 } while timed event list not empty
```

**Figure 4.1:** Pseudo-code of the SystemC main loop

the update queue is processed. When a signal's update causes it to change values, a delta event is added to the delta event list. After all updates are handled, the delta events are processed by adding the processes which are sensitive to the events to the runnable list. At this point, if the runnable list is not empty, another delta cycle begins. Once the runnable list becomes empty, time is advanced, all processes sensitive to that time (e.g. clock edges) are added to the runnable list, and a new delta cycle begins.

### 4.1.2 Partitioning Specification

The partitioning instructions for SPRI are given by a partitioning specification language (PSL). The PSL is a set-based language. The PSL permits basic set operations such as union, addition, and subtraction when describing the partitioning of microarchitectural simulators. The entire input simulator is considered as `all`; the moved-to-hardware portion can be expressed either as `tohw [all - software set]` or as `tohw [hardware set]`. The software set or hardware set may contain three types of simulator elements: class, instance, process. A "class" element represents all the instances that are instantiated from it; a "instance" element includes all the processes that are used in it; a "process" element covers all the function calls that are invoked from it. Four examples of the PSL are shown in Figure 4.2. Example 1 moves everything to hardware except all the instances of the `Adder` class. Example 2 specifies a template class and a normal class and the instances instantiated from these two classes should be moved to hardware. Example 3 parti-

```
Example 1: set adder = tohw [::``Adder''];
           tohw all - adder;

Example 2: tohw [::``MUX<3, Uint32_t>''] | [::``RegisterFile''];

Example 3: tohw [PowerPC0.ALU];

Example 3: tohw [::``IDEX_Register''::``FrontEdgeProcess''%func%];
```

**Figure 4.2:** Partitioning specification examples

tions the `ALU` instance of a PowerPC core to hardware. The last example indicates that only the `FrontEdgeProcess` process of the `IDEX_Register` class should be implemented in hardware.

PSL specifications are provided by users and parsed by SPRI to indicate the set of processes that will be moved to hardware. SPRI extracts the information (i.e simulator object, object connectivity) for the moved-to-hardware processes based on the parsed PSL specification and passes them to the rest part of SPRI that performs code modification and code generation for the output hybrid simulator.

## 4.2 SPRI Output

The output of SPRI is a complete hybrid structural microarchitectural simulator. It is composed of a SystemC portion, an FPGA portion, and a SystemC/FPGA interface. The SPRI-generated hybrid microarchitectural simulator must run on a host platform that has a general-purpose processor for the software and FPGAs for the hardware. The software portion, running on the host processor, is able to interact with the OS to simulate device I/O and system calls; the hardware portion is expressed in VHDL and synthesized to gate for FPGA configuration using commercial tools (i.e Xilinx ISE). The FPGA can be arbitrarily reconfigured for simulator changes, and it can provide on-chip RAMs and DSP units to support the simulation of memory hierarchy and complex computational operations (e.g. floating point). The software portion and the hardware portion are synchronized through a SystemC/FPGA interface. The stay-in-software SystemC portion is linked with a software wrapper that transfers data with the hardware through device APIs. The moved-to-hardware SystemC portion is transformed to VHDL code with an additional VHDL wrapper that communicates directly with the software wrapper. This interface remarkably impacts the simulation performance. Because the modules of a microarchitectural simulator

are tightly-coupled, the SystemC/FPGA interface must be efficient enough to handle the frequent cross-boundary synchronization; otherwise, a high communication overhead would overshadow the simulation speedup achieved by using FPGAs.

Interfacing SystemC and FPGAs is very challenging. The cross-boundary interface must maintain state synchronization between SystemC and FPGA. This synchronization has two phases: before simulation and during simulation. Before simulation, the hardware portion should be initialized with the same states as the software portion. SPRI can either synthesize the initial states into VHDL directly or modify the software code to initialize the hardware at the beginning of simulation. During simulation, simulator states must be synchronized frequently so that both data correctness and timing correctness can be guaranteed: correct data needs to pass through the correct signals at the correct simulated time-stamp. Because microarchitectural models are timed, what matters to the simulation correctness is the value changes of signals at the simulated clock edge. Synchronizing clocked processes on the software side and the hardware side is easy to implement; however, combinational processes are problematic. SystemC introduces the delta-cycle concept to allow multiple invocations of a combinational process whenever its inputs change, and its outputs must be stabilized before the simulation moves on to the next time-stamp. Delta cycles do not exist in hardware, so the interface must either reflect the value changes immediately to the hardware or wait for the stabilized values and transfer them once for the current timestamp where the latter is more optimized and efficient.

When target systems scale to many cores or higher system integration, a frequent communication with a extensive amount of data cross the software/hardware boundary can easily mitigate performance increase and may lead to a simulation even slower than the software-only one. Thus, interface optimization becomes increasingly critical. An optimum interface design relies on a good partitioning to balance the trade-offs between hardware capacity, communication latency, and communication bandwidth of the host system. The exploration for a good partitioning and an efficient interface is prohibitive due to the costly manual implementation process. Fortunately, SPRI solves this problem through the design automation, and the users of SPRI only need to specify a partitioning decision and select a interface.

## 4.3 SPRI Host Platform

SPRI is designed to produce hybrid microarchitectural simulators which can run on general computer workstations that are connected with FPGAs. The software portions run on the workstations and the hardware portions are downloaded to the FPGAs. The output of SPRI — hybrid structural microarchitectural simulators can be easily ported to different platforms. The simulator partitioning process and VHDL synthesis process are platform-independent, but the interface is not. To make the interface design transplantable, SPRI creates the SystemC/FPGA interface in two separate portions: a platform-independent portion (communication mechanism) and a platform-dependent portion (device APIs). When switching host platforms, only the platform-dependent APIs need to be replaced.

SPRI was tested on the DRC-1000 system [42]. The DRC computer is a integrated system, and provides complete and reliable device APIs to the on-board FPGA, SRAM, and DRAM. The API library saves a massive amount of time spent on writing and testing device API drivers. The DRC-1000 system adopts a CPU socket bus connection for its CPU and FPGA which offers a low-latency and high-bandwidth communication. The FPGA capacity of this system is relatively small and not expandable, but it allows us to clearly demonstrate how hardware capacity, communication latency, and communication bandwidth affect the performance of hybrid simulation when the target microarchitectural model scales.

## 4.4 SPRI Organization

The SPRI infrastructure integrates a chain of tools. Because a hybrid simulator design involves three steps: simulator partitioning, hardware design, and interface construction, SPRI is naturally divided into three parts:

1. a partitioning tool which understands user-input partitioning specifications and automatically splits simulator code into software and hardware portions;

2. a synthesis tool which generates RTL for the hardware portion of the simulator;

3. an interface tool which produces both the software-side and the hardware-side of the interface.

When producing hybrid simulators, SPRI transforms SystemC code into a mix of SystemC code and VHDL code. In this process, SPRI must be capable of manipulating code, optimizing code, and generating code in this transformation. SPRI uses a compiler framework to parse the SystemC code and performs code optimization and code generation based on its intermediate representation (IR). The compiler framework serves as the front-end; the three tools of SPRI perform IR analysis, optimization, and backend code generation.

### 4.4.1 LLVM Compiler Framework

The SPRI infrastructure is created on top of the LLVM compiler framework [43]. SPRI operates directly on LLVM's intermediate representation (IR) rather than on the original source code. Using LLVM has two benefits. First, doing so removes the need to parse SystemC or deal with the fine points of its semantics. LLVM is capable of compiling SystemC code, because SystemC is C++ with a extended library [44]. Second, we can take advantage of LLVM's ability to perform both static and run-time optimization.

LLVM is an open source compilation infrastructure allowing both static and run-time code optimization and generation. It provides complete compiler front-ends for both C and C++. LLVM compiles C and C++ into the LLVM IR. The LLVM IR is composed of RISC-like instructions in a single-static assignment form which combines data and control flows to permit easy static and dynamic optimization. LLVM can produce target machine code for multiple target platforms such as X86, Sparc, and PowerPC. LLVM has a modular structure, and customized transformations and passes can be easily plugged in, and it can also be linked as a library into other programs. A very active and vibrant user community supports the continued development and maintenance of LLVM. Additionally, LLVM uses a non-GPL open source license that does not restrict the release of SPRI.

LLVM supports both static and dynamic compilation and therefore facilitates SPRI to perform transformation and synthesis during a run of the software-only simulation after elaboration is finished. This feature allows SPRI to inspect the data and objects of SystemC modules at run-time after they have been resolved and stored in system memory. As a result, the synthesis methodology of hybrid microarchitectural simulators proposed by SPRI is "hybrid" too, involving both static and dynamic operations. We now discuss this synthesis flow.

### 4.4.2 SPRI Synthesis Flow

SPRI is built upon LLVM and operates directly on the LLVM IR instead of SystemC source code. SPRI performs a series of optimizations and transformations on the LLVM IR of the input software-only simulator, and produces the corresponding hybrid simulator. SPRI synthesizes under the run-time environment of LLVM which permits accesses to the information that can be used to analyze the input microarchitectural models but is not available statically.

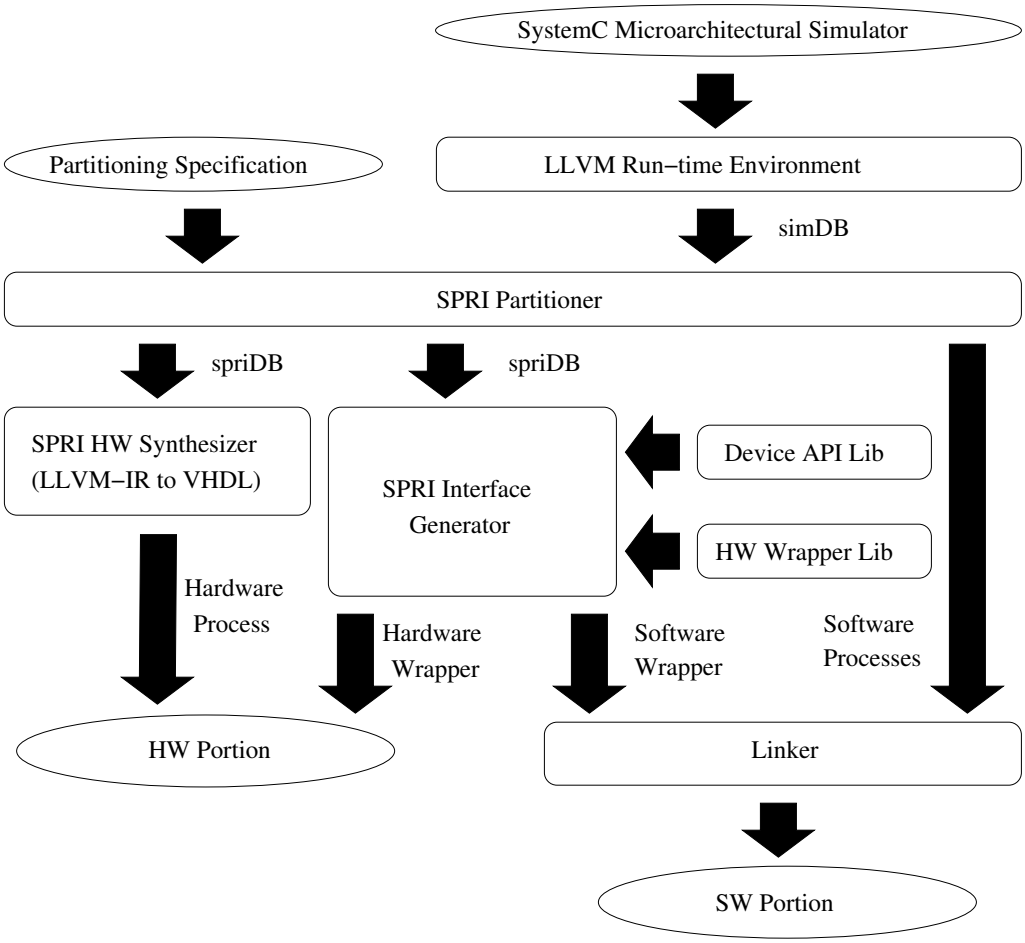



**Figure 4.3:** SPRI synthesis flow

Figure 4.3 displays the synthesis flow of SPRI. SPRI operates during a run of the input software-only simulator after elaboration is finished and the simulation is suspended. After elaboration, SystemC objects have been instantiated and stored in system memory. SPRI collects those

in-memory objects and organizes them in a simulator database. This database is named SIMDB and is provided for the SPRI partitioner. SIMDB maintains the module hierarchy of the input simulator. This database records both object information and hierarchical information. The object information contains SystemC module class, instance, method, port, signal, and event; the hierarchical information describes what methods each module class contains, which instances are instantiated from the same module class, how signals and ports are connected, and what events can be triggered by a signal. The object information and the hierarchical information are obtained by parsing the C++ run-time type information (RTTI) defined in the Application Binary Interface. This parsing relies upon specific naming conventions for C++ mangled names and thus would need to be changed for different operating systems or compilers; we have attempted to isolate these changes to small portions of the code. The SPRI infrastructure is composed of three primary tools: the SPRI partitioner, the SPRI VHDL synthesizer, and the SPRI interface generator. The SPRI partitioner guides the hardware synthesis and the interface generation.

The output of the SPRI partitioner is a different database named SPRIDB. Based on a PSL specification, the SPRI partitioner extracts the information of the moved-to hardware portion from SIMDB and reorganizes it in SPRIDB. SPRIDB presents the hardware portion as a set of hardware processes; each hardware process has a link to the corresponding LLVM IR. The hierarchy for the hardware processes is completely flattened in that the hierarchical structures (i.e. class, instance, ports) are ignored and the inter-process communication only goes through signals. Flattening the hierarchy of the hardware portion can reduce the complexity of the hardware synthesis and the interface generation, because they don't have to reconstruct data routing through modules and processes in the original hierarchical manner. Figure 4.4 displays an example of the SPRI partitioning in which three hardware processes are extracted from SIMDB and flattened in SPRIDB. During the partitioning process, SPRIDB does not collect SystemC event objects from SIMDB; instead, it summarizes triggering sources (i.e clock, signal, event) and triggering types (i.e level trigger, edge trigger) in a sensitivity list for each hardware process.

SPRIDB represents the hardware portion of the input simulator. This database is passed to the SPRI VHDL synthesizer and the SPRI interface generator. The SPRI VHDL synthesizer operates on the LLVM IR of each process recorded in SPRIDB and produces VHDL code for the hardware process set. The SPRI interface generator creates a software wrapper and a hardware

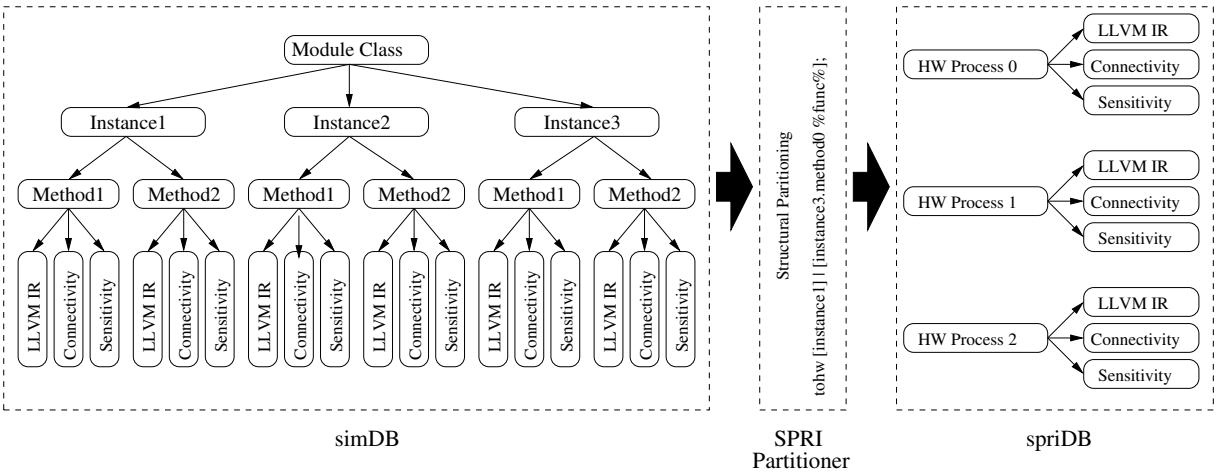



**Figure 4.4:** SPRI simulator partitioning

wrapper. The data transfer that crosses the software/hardware boundary is performed through the communication between these two wrappers. The software wrapper is a mix of the modified LLVM IR and newly-generated C++ code, and it is linked with the software process set to complete the software portion. The hardware wrapper is generated in VHDL as the top level design for the hardware portion. The hardware wrapper connects the VHDL blocks produced by the SPRI VHDL synthesizer, and sets up state machines to transfer data with the software wrapper.

Most commonly, SystemC microarchitectural models are structurally partitioned to form hybrid simulators, which is straight-forward to implement. A structural partitioning is performed by placing only structural objects (i.e process) in the hardware and cutting only structural objects (i.e. signals) on the software/hardware boundary. Because VHDL has similar semantics for the structural portions of SystemC, most SystemC structural objects can be equivalently mapped to VHDL in the synthesis. The structural partitioning results in a clean boundary between the hardware processes and between the software and the hardware where the communication only occurs via signals. However, SystemC allows the use of non-structural constructs in modeling to improve modeling productivity and simulation speed. Thus, SPRI supports certain non-structural partitionings in addition to structural partitionings. The support of non-structural partitionings requires the transformation of SystemC non-structural features to equivalent hardware implementations; the misinterpretation of these features will conduct functional incorrectness and timing mismatch in the hybrid simulation.

# Chapter 5

# SPRI Interface Generation

The SPRI interface generator creates the software/hardware interfaces for the output hybrid microarchitectural simulators. The SPRI interface generator performs two tasks: fixing up the partitioning boundary and producing a software wrapper and a hardware wrapper. The software and hardware wrappers are placed on the partitioning boundary. They redirect signal inputs and outputs between SystemC and FPGA to synchronize the simulation. This software/hardware synchronization introduces an communication overhead, and thereby has a significant impact on the performance of the hybrid simulation. High-overhead interfaces may overshadow the speedup achieved through the FPGA acceleration. Typically, the synchronization overhead can be reduced by two means.

1. Lowering the frequency and data amount of the cross-boundary communication. As stated previously, microarchitectural models are clocked logics, so the software/hardware interface does not have to synchronize at every delta cycle. It can wait till all the combinational outputs become stable and transfer a packet of data once. However, the bandwidth of the physical communication channel may become the bottleneck and limit the packet size, especially when the number of simulated cores scales and the data to different cores is packed for transfer.

2. Overlapping the communication with the computation. This method decouples the software portion and the hardware portion of a hybrid simulator. The software and the hardware execute asynchronously which however does not mean there is no synchronization of the simulator state between the two of them. The asynchronous execution allows the overlap of the communication and the computation. The communication overhead is hidden in the

computation in order to mitigate the impact from the latency of the physical communication channel.

In the previous works, the hybrid transplanting simulator [27] and the hybrid structural simulator [25] both use the coprocessor-style (block and wait) interface; the hybrid functional-timing simulator [14] decouples the software and the hardware through queues to allow asynchronous execution. The simulator designed in [26] even implements a microarchitectural simulator completely in hardware to avoid the communication overhead. Unfortunately, these interfaces are designed by hand, and none of them targets interfacing SystemC and FPGA. Thus this chapter starts with defining the design space of the software/hardware interfaces that can be applied for SystemC/FPGA-based hybrid microarchitectural simulators. Then, it discusses the design and synthesis of different interfaces in the design space, and evaluates the trade-offs between speed, hardware cost, and scalability.

## 5.1 Design Space

The software/hardware interface of a hybrid microarchitectural simulator must coordinate the production of new signal values and the update of state between the software processes and hardware processes. There are two dimensions along which the design space of software/hardware interfaces can be characterized: concurrency and composition.

### 5.1.1 Concurrency

The first dimension is the amount of concurrency provided by the interface. This concurrency may exist between hardware elements and between hardware and software. In general, as concurrency increases, we would expect higher simulator performance. At one extreme, concurrency may be non-existent. When either software or hardware requires an operation of the other, a request is made through the interface and it blocks execution until the request has been handled [25], [27]. At the other extreme, software and hardware can be fully concurrent: either software or hardware may initiate multiple requests to each other and continue on its own work without waiting for the completion of their requests [14].

### 5.1.2 Composition

Composition is the direct interconnection of a set of hardware processes. Signals internal to the set do not need to be communicated through the interface. In general, as the amount of communication decreases, we should expect higher simulator performance. We consider three degrees of composition:

- **No composition:** All communication between hardware processes takes place via software.
- **Single-FPGA-cycle composition:** Only single-FPGA-cycle hardware processes are composed.
- **Multi-FPGA-cycle composition:** All hardware processes are composed.

The distinction between single-FPGA-cycle and multi-FPGA-cycle composition is an important one. SystemC processes execute instantaneously with respect to simulated time. However, efficient FPGA implementations of those processes may require several FPGA cycles to compute the outputs and next state of one simulated cycle. We call FPGA process implementations which require only a single cycle to compute *single-FPGA-cycle* hardware processes and those that require multiple cycles *multi-FPGA-cycle* hardware processes.

Single-FPGA-cycle hardware processes may be composed directly without affecting their correctness because the processes can directly implement the state machine being modeled. However, as described in [45], multi-FPGA-cycle hardware processes cannot be directly composed because the processes actually implement a different state machine which requires multiple FPGA cycles to *simulate* the modeled state machine. Thus single-FPGA-cycle composition is fundamentally different from multi-FPGA-cycle composition.

Multi-FPGA-cycle composition is possible if the hardware processes are designed to be latency-insensitive. Latency insensitive processes could be designed by the user, however, in order to *synthesize* an arbitrary structural process into a latency-sensitive version, some formal methodology for creating latency-insensitive processes is required. This methodology can be provided by Latency-Insensitive Bounded Dataflow Networks (LI-BDNs), which are formalized in [46]. Hardware processes can be wrapped to form LI-BDN processes using the interface and control logic

**Table 5.1:** The interface design space

| Designation | Concurrency | Composition | Notes |
|---|---|---|---|
| BL-NONE | Blocking | None | |
| BL-SC | Blocking | Single-FPGA-cycle | |
| BL-MC | Blocking | Multi-FPGA-cycle | Nonsensical |
| NB-NONE | Non-blocking | None | |
| NB-SC | Non-blocking | Single-FPGA-cycle | |
| NB-MC | Non-blocking | Multi-FPGA-cycle | |

shown in Figure 5.1. After LI-BDN transformation, LI-BDN processes are connected through FIFOs and simulated time is interpreted as enqueue and dequeue operations on the FIFOs: in one simulated clock cycle, each FIFO is enqueued and dequeued once.[1] LI-BDN processes execute autonomously: when their inputs become available, they produce outputs and update state.

### 5.1.3 Combining the Dimensions: the Interface Design Space

Table 5.1 lists each of the points in the interface design space as well as a name for each interface design style. Note that one design choice – BL-MC – is nonsensical; because LI-BDNs execute autonomously, there can be no notion of software making a request for the hardware to compute and blocking to wait until hardware finishes.

The non-blocking interfaces should have better performance than the blocking interfaces due to their higher concurrency. Furthermore, interfaces with more composition should perform better than interfaces with less composition due to reduced communication overhead. However, multi-FPGA-cycle composition could have higher FPGA utilization caused by the LI-BDN transformation. Note that it is somewhat obvious that BL-NONE should perform much worse than the other interfaces. However, we still describe and evaluate BL-NONE because it corresponds to the natural "co-processor" model of using hardware accelerators: simply request the hardware to do some computation and then wait for it.

[1] Not all models can be composed in this fashion, as cycles of data dependence are not allowed. A detailed description of the LI-BDN implementation of hardware processes for SPRI is given in [47].

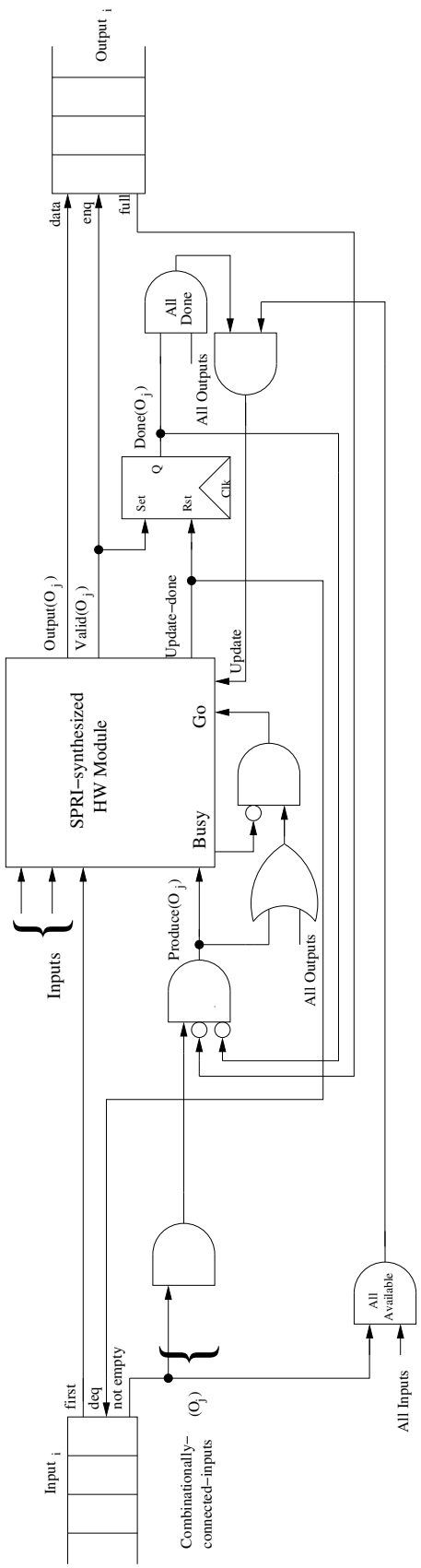



**Figure 5.1:** SPRI-synthesized LI-BDN wrapper [45]

## 5.2 Implementation

The SPRI interface generator can be configured to output any one of the five meaningful interfaces specified in Table 5.1. Rather than implementing five different interface generators, the SPRI interface generator abstracts a top-level interface template as diagrammed in Figure 5.2 and applies it for all the five interfaces where different code may be produced for the interface blocks. This interface template separates the platform-dependent portion and the platform-independent portion, and thereby can be easily ported to different host platforms.

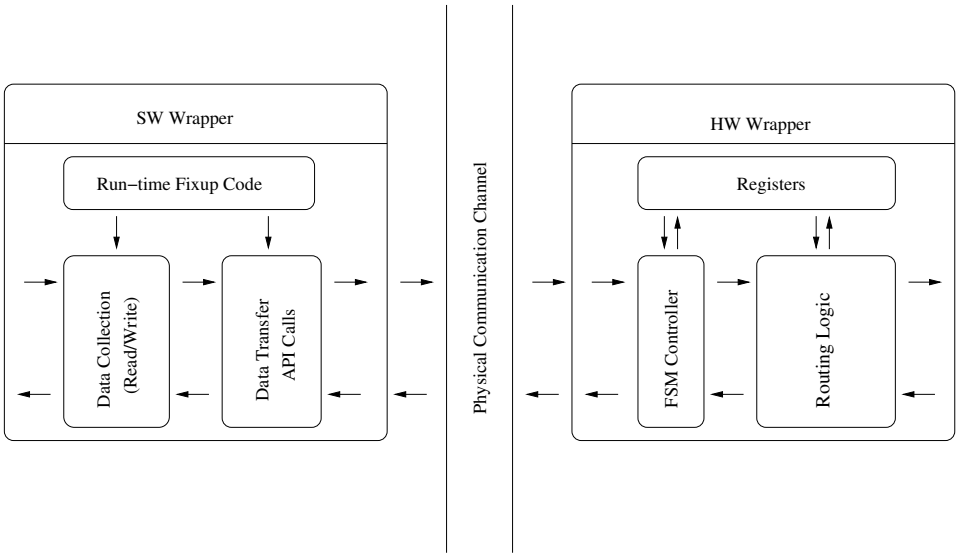



**Figure 5.2:** The abstract form of the SystemC/FPGA interface

In a software wrapper, a mix of SystemC code and LLVM IR is produced for the fix-up portion, and C++ code is produced for the data-collection and device-API portions. The fix-up portion, first of all, disconnects the moved-to-hardware processes from the software simulator by removing the sensitive events for each of these processes so these processes will not be invoked anymore; then it switches the simulation from the software simulation to the hybrid simulation at the exact timestamp where the software simulation is suspended for the VHDL synthesis. After the "switch over", the disconnected processes are replaced with a set of new SystemC processes that communicate with the hardware. These new processes need two fix-ups: time to invoke (i.e sensitivity) and places to load and store data (i.e. addresses of signal objects). The new SystemC

processes and events are carefully constructed and arranged to ensure that the interface code runs at the desired points within the main SystemC simulation loop but without modifying the simulation engine. This is achieved by generating a set of processes and events in SystemC. Some processes fire events directly to invoke other processes, which permits the manipulation of simulation time at the delta-cycle level. When the replacement processes are invoked, they complete the data exchange with the hardware through the data-collection portion and the device-API portion.

In a hardware wrapper, VHDL code is generated to bridge the communication channel of the host platform with the hardware processes. The hardware wrapper controls inbound and outbound data through a FSM controller on the signals of the communication channel. Cross-boundary data are registered and routed to the corresponding signals for each hardware process.

The device API and the communication-channel FSM are the platform-dependent portions. These two portions can be easily replaced through the device-API library and the FSM-template library of SPRI. The capability allows SPRI-generated hybrid simulators to be easily transplanted to different host platforms without changing the interface code.

### 5.2.1 Blocking, No Composition (BL-NONE)

This interface blocks after making requests of the hardware and allows no direct communication between hardware processes. The software side of the interface simply replaces the software process corresponding to each hardware-implemented process with a *proxy* process sensitive to the same set of signals.[2] As shown in Figure 5.3, each proxy process first reads input signals from SystemC and sends them to the hardware. It then waits for the hardware to finish execution, reads the values of the output signals from hardware, and writes them to the corresponding SystemC signals. The hardware automatically begins execution when the last input signal is written; if the process is sensitive to the clock, the hardware updates state as part of its execution.

On the software side, better performance is achieved by staging the signal values through a memory buffer, transferring the entire buffer at once rather than each signal value individually. We speculatively read the hardware outputs at the same time we poll the status register; in the common

[2] A process sensitive to no signals will have a proxy which is never fired; this is correct, as the process must be driving a constant value which will be driven during the hybrid simulator's initialization.

```
proxy() { // sensitive to input signals or clock edge
   foreach input signal s
      WriteDataToHW(s.read());
   // hardware begins execution and updates state
   WaitForHWFinish();

   foreach output signal s
      s.write(ReadDataFromHW());
}
```

**Figure 5.3:** BL-NONE interface: software side

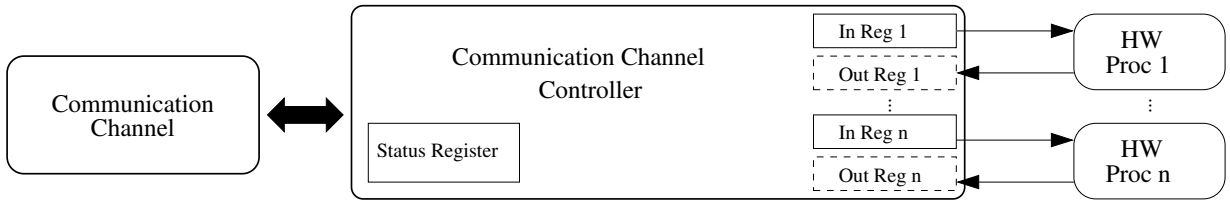



**Figure 5.4:** BL-NONE interface: hardware side

case of hardware finishing before the poll, the poll and data transfer occur with only a single device driver call.

The hardware side of the interface (Figure 5.4) has a communication channel controller (CCC) which manages the interface. Signal values are stored in distributed registers which are memory-mapped into the communication channel's address space. These registers are connected directly to the hardware processes. There is also a memory-mapped completion status register. As a size optimization, output signals driven by single-FPGA-cycle hardware processes do not have output registers, because such processes produce their outputs continuously. The CCC triggers the execution of a hardware process when the hardware process's last input signal is written; the CCC sets a completion bit in the status register when it receives a "done" signal from a hardware process. The software must read the status register to determine data availability before it reads and the CCC must set the completion bit only after all the output signal registers for a process are valid. The status register is mapped to address 0 so that a sequential read of addresses will produce the correct ordering.

```
proxy() { // sensitive to input signals or clock edge
   foreach cross-boundary input signal s
      copytoBuffer(s.read());
   if (sensitive to clock edge)
      clockedTransferEv.notify();
   else
      nonclockedTransferEv.notify();
}

transfer() { // sensitive to a transfer event
   WriteDataToHW(data_from_buffer);
   // hardware begins execution and updates state
   WaitForHWFinish();
   foreach cross-boundary output signal s of the process set
      s.write(ReadDataFromHW());
}
```

**Figure 5.5:** BL-SC interface: software side

### 5.2.2 Blocking, Single-FPGA-cycle Composition (BL-SC)

This interface blocks after making requests of the hardware but permits direct composition of single-FPGA-cycle hardware processes. All single-FPGA-cycle processes are composed to internalize communication.

The software side of the interface has a proxy process for each process that has input signals across the software/hardware interface. All single-FPGA-cycle processes are grouped into a single *process set*. Two *transfer* processes are required to transfer data and update values, one for the clock-sensitive processes in the process set and another for the non-clock sensitive processes in the process set. The two-step proxy/transfer flow is motivated by a desire to efficiently transfer data for many signals with a single device driver call. Figure 5.5 gives pseudo-code for these processes.

A proxy process copies its cross-boundary input signal data into a buffer and triggers a transfer process to run using a SystemC immediate notification. Each transfer process writes the data from the buffer to the hardware and starts hardware execution. After the hardware finishes, the transfer process reads the cross-boundary output signals from hardware and writes them to the corresponding SystemC signals. Output signals are polled speculatively as in BL-NONE.

The transfer processes for clock-sensitive and non-clock-sensitive processes are different. First, the clock-sensitive transfer process writes one additional word of data to the hardware which the hardware interprets as an UPDATE STATE command. Second, while the non-clock-sensitive process transfers all the cross-boundary outputs of non-clock-sensitive processes, the clock-sensitive

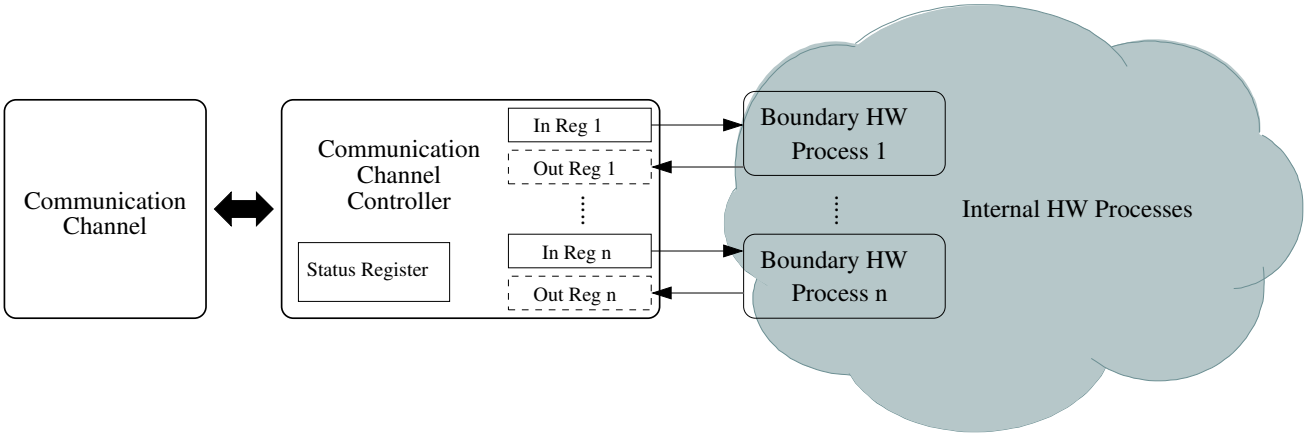



**Figure 5.6:** BL-SC interface: hardware side

process must transfer the cross-boundary outputs of all processes. This difference occurs because the state update in the hardware may have changed signal values which are combinationally dependent upon the state and thus affected the outputs of non-clock-sensitive processes.

The structure of the hardware side of the interface is shown in Figure 5.6. It is quite similar to that of BL-NONE, except that the CCC need only consider cross-boundary inputs and outputs. The CCC asserts an UPDATE STATE signal to all processes in a process set when a dummy address just beyond the last cross-boundary signal value's register for that set is written. This extra dummy address provides easy control over whether state updates occur. Only a single completion status bit is maintained for each process set.

### 5.2.3 Non-blocking, No Composition (NB-NONE)

This interface does not block after making requests of the hardware but allows no direct communication between hardware processes. To achieve a non-blocking interface, the software side of the interface must separate the requests it makes to hardware from the checks it makes for the completion of those requests. It must also execute any runnable software processes without waiting for hardware completion. The software side has a proxy process for every hardware-implemented process, two transfer processes, and two *completion* processes. The pseudo-code for these processes is shown in Figure 5.7.

The proxy process is similar to that of BL-SC. As in BL-SC, the transfer processes write data to hardware, however, they do not wait for the hardware to complete. Instead, they schedule

a completion process to run later by firing an event to which the completion process is sensitive. There is one transfer process for all clock-sensitive proxy processes and one transfer process for all non-clock-sensitive proxy processes; the clock-sensitive transfer process writes an additional word of data to indicate UPDATE STATE to the hardware, as in the BL-SC interface.

```
proxy() { // sensitive to input signals or clock edge
   this.request = true;
   foreach input signal s
      copytoBuffer(s.read());
   if sensitive to clock edge
      clockedTransferEv.notify();
   else
      nonclockedTransferEv.notify();
}

transfer() { // sensitive to a transfer event
   if !busy or this is the clocked transfer process
      WriteDataToHW(data_from_buffer);
      // hardware begins execution and updates state
      busy = true;
      if this is the clocked transfer process
         clockedCompletionEvent.notify(SC_ZERO_TIME);
      else
         nonclockedCompletionEvent.notify(SC_ZERO_TIME);
   else
      mustRedo = true;
}

completion() { // sensitive to a completion event
   checkHWstatus();
   foreach proxy process p such that the
            corresponding hardware process is not active
            and p.request is true
      foreach output signal s of p
          s.write(ReadDataFromHW());
      p.request = false;
   if no hardware processes are active
      busy = false;
      if mustRedo
         mustRedo = false;
         nonclockedTransferEv.notify(SC_ZERO_TIME);
   else
      if this is the clocked completion process
         clockedCompletionEvent.notify(SC_ZERO_TIME);
      else
         nonclockedCompletionEvent.notify(SC_ZERO_TIME);
}
```

**Figure 5.7:** NB-NONE interface: software side

There is a completion process for each transfer process. Each completion process begins by determining which hardware processes are executing by reading a hardware status register. The completion process then reads the output signals of each non-executing process whose proxy

process has requested hardware process execution and writes them to the corresponding SystemC signals, clearing the proxy process' request flag. Finally, if any of the corresponding hardware processes are still executing, the completion process schedules itself to run again.

The hardware side of the interface is very similar to that of the BL-NONE interface. Indeed, the same hardware could be used. However, we optimize this hardware slightly by using only two trigger signals, one for clock-sensitive processes and one for non-clock-sensitive processes. The memory mapping of the input signal registers is arranged such that the input signals of each of these two groups of processes are located at consecutive addresses and thus the last signal to be written to in each group may be detected. Reducing the number of trigger signals slightly reduces the FPGA resources required.

### 5.2.4 Non-blocking, Single-FPGA-cycle Composition (NB-SC)

This interface does not block after making requests of the hardware and permits direct communication between hardware processes which execute in a single FPGA cycle. Processes are grouped into process sets such that single-FPGA-cycle processes which share an input or output signal are in the same set. The software side of the interface is structured identically to the software side of the NB-NONE interface, with four differences:

1. Processes whose input/output signals are completely contained within hardware have no proxy processes.

2. There are two transfer and two completion processes for each process set. There is a pair of busy and redo flags for each process set.

3. Proxy, transfer, and completion processes only read, transfer, and write signals which are on the boundary between hardware and software. Transfer and completion processes only transfer and write those signals which are destined for or produced by their process set.

4. There are no individual request flags for the proxy processes. The non-clock-sensitive completion process writes all boundary signals produced by non-clock-sensitive processes in its process set. The clock-sensitive completion process writes all boundary signals produced by all processes in its process set.

```
clocked() { // sensitive to clock edge
   transferEvent.notify(SC_ZERO_TIME);
   completionEvent.notify(SC_ZERO_TIME);
}

transfer() { // sensitive to transferEvent
             //   and all boundary input signals
   foreach boundary input-to-software signal s
      if s.available
         copytoBuffer(s.read());
      set/clear available flag for s in buffer
   if any signal is available
      WriteDataToHW(data_from_buffer);
}

completion() { // sensitive to completionEvent
   foreach boundary output-to-hardware signal s
      if s.availableInHW();
         s.write(ReadDataFromHW());
         s.available = true;
   if any boundary output signal is not yet available in hardware
      completionEvent.notify(SC_ZERO_TIME);
}
```

**Figure 5.8:** NB-MC interface: software side

The hardware side of the interface is identical to that of the BL-SC interface.

### 5.2.5 Non-blocking, Multi-FPGA-cycle Composition (NB-MC)

The final interface does not block after making requests of the hardware and permits composition of multi-FPGA-cycle hardware processes. Such composition requires the use of a latency-insensitive hardware implementation method such as LI-BDNs and a rather different interface. Two issues must be addressed. First, because time in LI-BDNs is measured in enqueue and dequeue operations, the interface must ensure that exactly one value per boundary input signal to the hardware is enqueued per simulated cycle. Second, LI-BDNs do not have a centralized notion of time as the software does; individual LI-BDN elements may "slip" in time from one another. Therefore, there must be a mechanism to allow the boundary LI-BDNs and software to coordinate updates of simulation time.

The software side of the interface is shown in Figure 5.8. There is one clocked process, one transfer process, and one completion process. The clocked process is made sensitive to the clock edge and serves solely to schedule the completion and transfer processes to run later.

The transfer process is made sensitive to every boundary input signal to the hardware as well as the event which triggers it at the start of the clock cycle. This process checks whether each

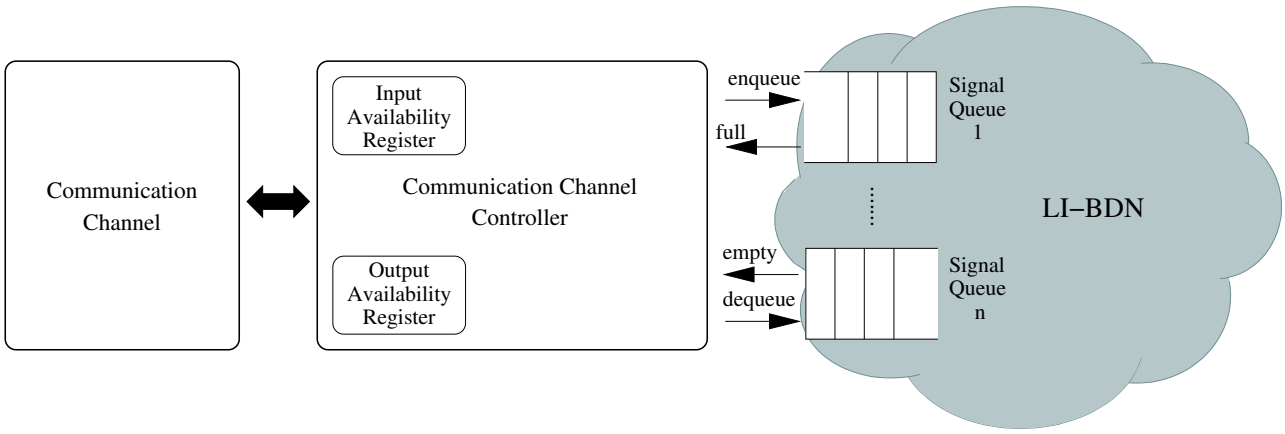



**Figure 5.9:** NB-MC interface: hardware side

signal is marked as *available*; if it is, its value is copied into a buffer and its availability flag in the buffer is marked. Then the buffer is written to hardware, which will enqueue the available signals which have not yet been enqueued in this cycle.

The completion process reads the boundary output signals as well as their availability. If a signal is available from the hardware and has not yet been made available to SystemC in this clock cycle, the signal is written to SystemC and set to be available. The completion process reschedules itself to run again if any output signals are not yet available in the hardware. [3] This rescheduling allows the interface to be non-blocking. As an optimization, all signal values and availability are speculatively read from the hardware in one device driver call.

The hardware side of the interface is shown in Figure 5.9. The CCC is connected to the LI-BDN through FIFOs. The CCC contains input signal value registers but no output signal value registers. It also contains availability registers for both input and output signals. LI-BDNs fire autonomously as their inputs become available, thereby requiring no process control signals.

The CCC sets a bit in the output availability register when an output FIFO is not empty. The heads of the output FIFOs can be read without dequeuing. The CCC also maintains a simple state machine for each boundary-crossing input signal; this state machine ensures that the signal is only enqueued once per simulated clock cycle.

[3] The SystemC processes which remain in software require modifications to propagate signal availability. Details of how these modifications are made within SPRI are beyond the scope of this work.

The end of a simulated clock cycle is detected when all of the input FIFOs have been enqueued, all of the output signals are available, and the CCC has serviced a read of the output signals in which all signals were available. When this occurs, all input state machines are reset, the output signal availability register is reset, and all output FIFOs are dequeued. To prevent race conditions, software must ensure that it writes the input signal availability after writing the values. Likewise, it must read the output signal values after reading the output availability. To make the interface implementation easy to achieve this execution order in block writes and reads, the input signal availability register is memory mapped at an address just after the input signal values and the output signal availability register is mapped at address 0.

## 5.3 Evaluation

We evaluate the performance and hardware resource utilization of the five explored interface designs for hybrid microarchitectural simulators. The hybrid simulators are synthesized by the extended version of SPRI from a microarchitectural model of a chip multiprocessor which contains a parameterizable number of simple, in-order PowerPC cores and a simplistic cache hierarchy. We chose a speculative-functional-first [48] simulator organization: all instruction-set functional behavior is separated from the timing behavior. SPRI keeps the functional behavior processes in software and synthesizes the timing behavior processes into hardware. This organization, partitioning, and simple model were chosen to highlight the differences between the five interfaces and demonstrate their scalability. Each core model in hardware has independent communication with the software; the size of data to be communicated grows linearly with the size of the model. The cores are simple so that we may fit large numbers of them on the FPGA and so that interface overhead is more pronounced. Because the core models are so simple, little internal parallelism can be exploited in hardware. As a result, we expect hybrid-simulator speedup to be mainly due to exposing the parallelism between cores. This parallelism is limited by the number of cores, so we expect speedups to not to be particularly high, however, there will be significant differences between the interfaces.

Hybrid simulators with the five interfaces are synthesized for six different core counts. The experiments are carried out on the DRC 1000 system and the VHDL-to-bitstream synthesis was done by ISE 8.2. The simulator is run on the FFT and Radix kernels from the SPLASH-2 [49]

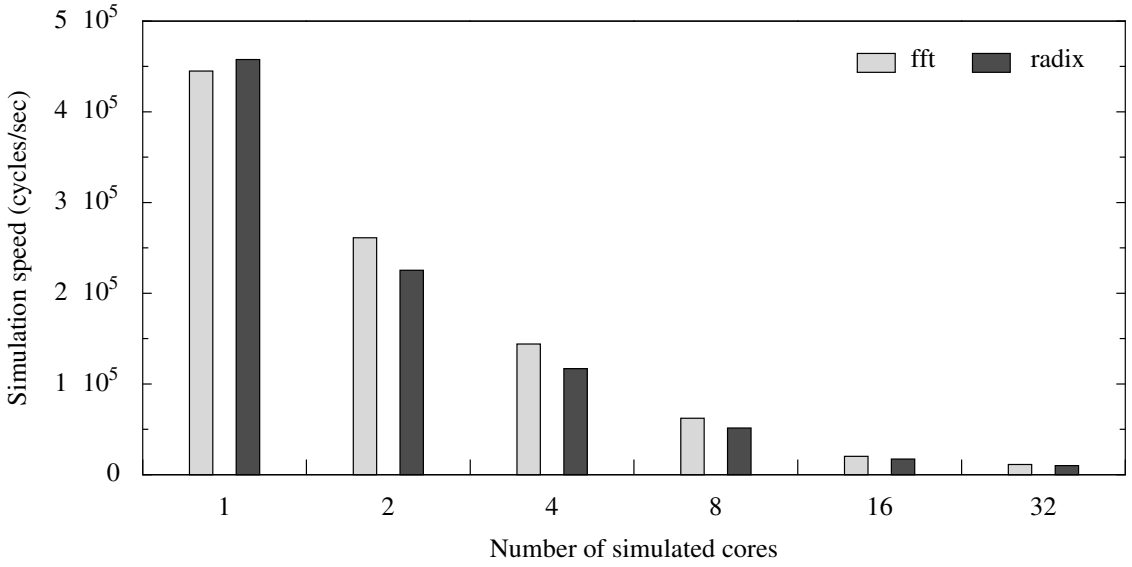



**Figure 5.10:** Software-only simulation speed

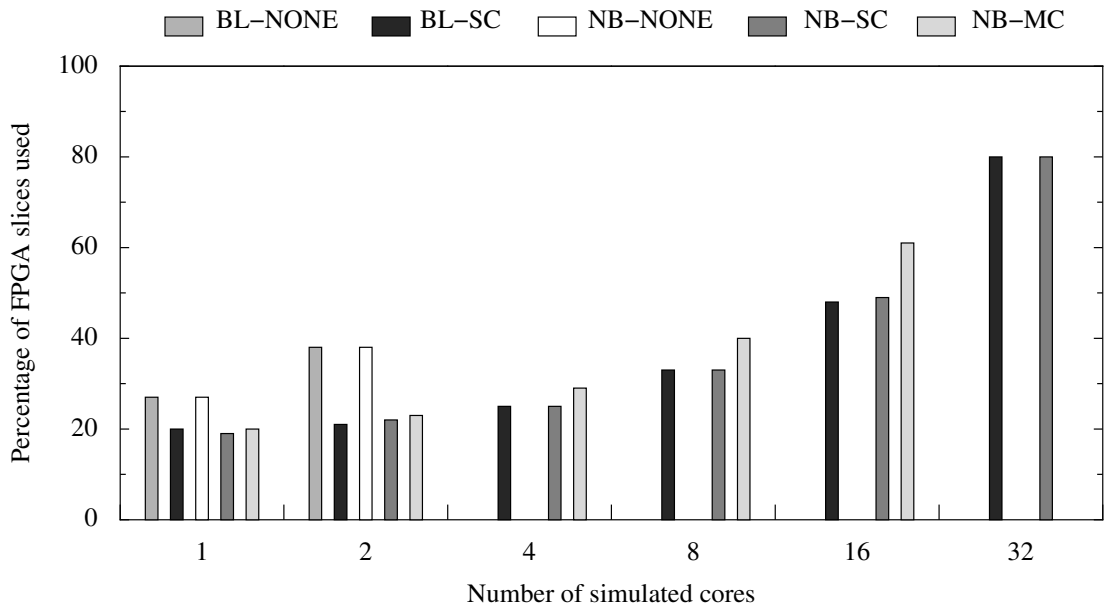
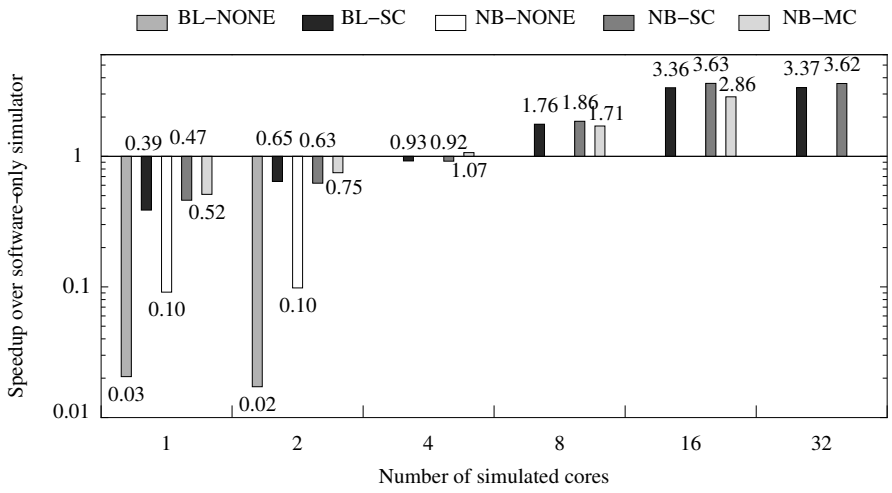


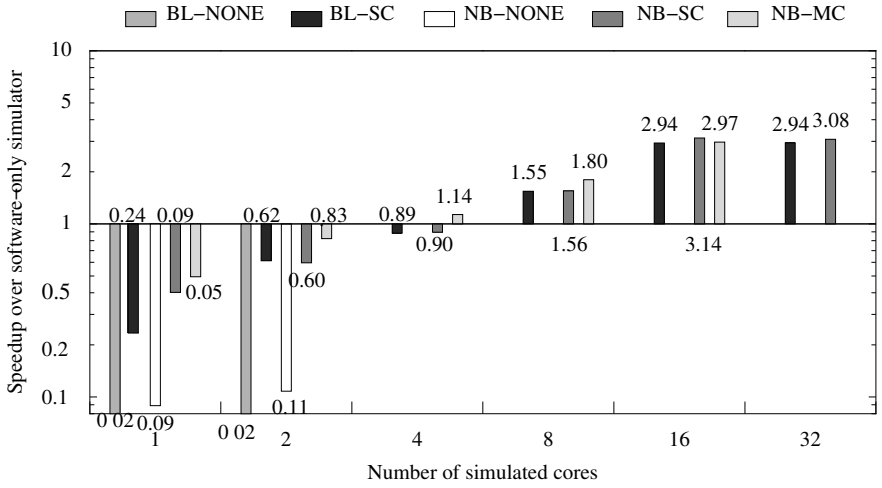

**Figure 5.11:** FPGA slice utilization

benchmark suite. LLVM 2.9 was used for compilation. Figure 5.10 shows the simulation speed for the software-only simulators using the reference implementation of SystemC 2.0.1.

Figure 5.11 displays the FPGA slice utilization for the synthesized hybrid PowerPC microarchitectural simulators. Missing bars correspond to hybrid simulators which could not be created due to FPGA capacity problems of two kinds. First, routing resources are limited for the




(a) Running FFT

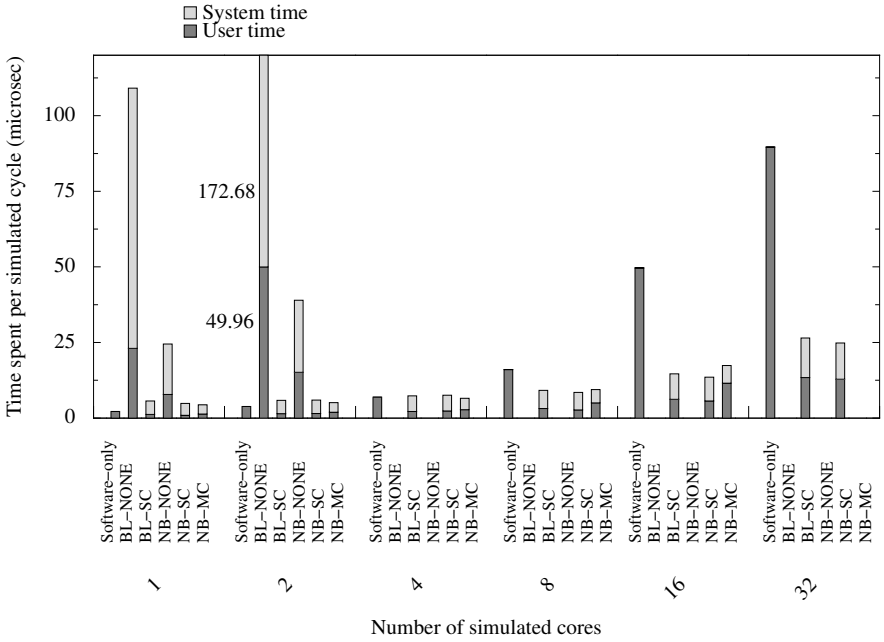



(b) Running Radix

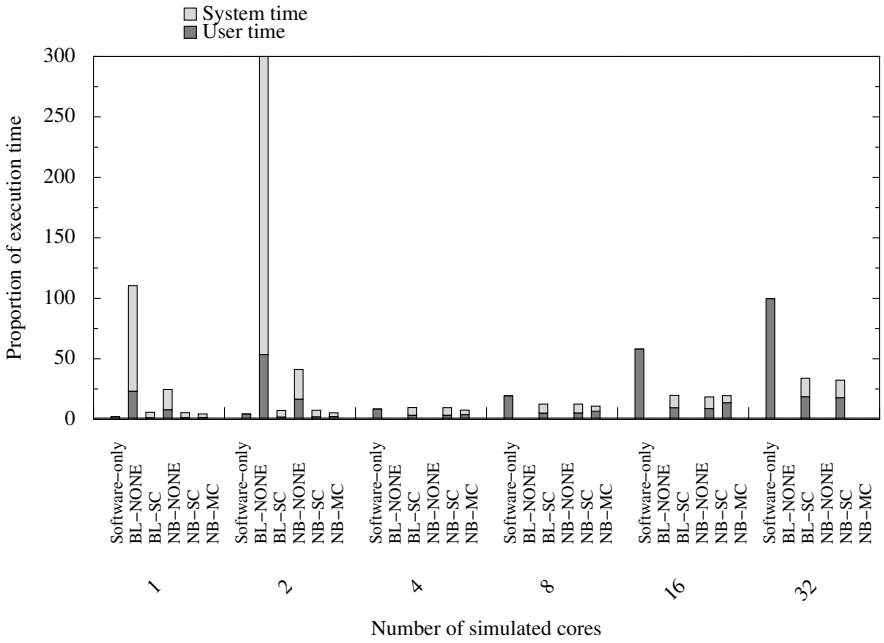
**Figure 5.12:** Speedup of hybrid over software-only simulation

non-composition interfaces. More than two cores for BL-NONE and NB-NONE failed to route, because large number of output signals can't be muxed together within the CCC. Second, multi-FPGA-cycle composition requires more slice resources because of the FIFOs and control logic between processes used by LI-BDN [45].

Figure 5.12 shows the speedup over the baseline software-only simulator achieved by the hybrid FAME simulators with the five interfaces. The FFT and Radix benchmarks have similar results. Missing bars indicate that the FPGA runs out of resources for the corresponding simulators.




(a) FFT




(b) Radix

**Figure 5.13:** Execution time breakdowns

As expected, even the best speedups for this model are not large, due to the simplicity and lack of internal concurrency of this model. However, there is an enormous difference between interfaces:

up to a 34x difference in speed. This is caused primarily by reducing the number of round-trip communications and interface overhead due to composition and nonblockingness.

Additionally, a certain amount of time is spent in the operating system. When the device driver is called for the software/hardware communication, it spins and waits for operations to complete. Figure 5.13 shows the amount of time spent per cycle in both system and user modes. The software-only model bars indicate how the amount of work which the baseline simulator must do per cycle grows rapidly as the models increase in size. The two non-composing interfaces spend the vast majority of their time in communication, because of the many round-trip communications in each cycle. In user mode, they both do 10 to 15 times more work which stems from copying signal data from place to place in the interface code. Thus both interfaces result in significant slowdowns. On the other hand, the amount of time spent in communication grows very slowly for the composing interfaces and the amount of time spent executing processes in software increases far less rapidly than it does for the software-only model. Thus, as the number of cores increases, the increasing amounts of offloaded concurrent work lead to higher speedups.

The speedups of hybrid simulation are affected by two aspects of the communication overhead: the number of round-trip communications and the amount of data transferred across the software/hardware boundary. First, as a result of the chosen partitioning and organization of this model, the NB-MC interface has only one round-trip communication per cycle, the lowest communication overhead; whereas the BL-SC and NB-SC have two: one for clock-sensitive processes and one for non-clock-sensitive processes. In this case, BL-SC has similar performance as NB-SC; with other partitionings, the blocking behavior would affect speeds more. Overall, the NB-MC interface should expect to encounter these one-trip scenarios more frequently. However, the NB-MC interface shows additional overhead in the user time; this overhead is spent in computing signal availability for software processes. Second, increasing the number of simulated cores in hardware causes more signals to cross between software and hardware. The increasing amount of transferred data eventually saturates the CPU-to-FPGA communication bandwidth, which thereby becomes the performance bottleneck: when scaling to 32 simulated cores with the NB-SC and BL-SC, the speedups of simulating 32 cores are not bigger than those of 16 cores.

To summarize, the combination of module composition and non-blocking communication overcomes the speed limitations by permitting hardware concurrency, reducing the communication

overhead, and overlapping communication and computation. The two preferred interfaces are NB-SC and NB-MC. NB-SC should be used whenever single-FPGA-cycle composition is possible, as it is both faster and less resource-intensive. NB-MC should be used when there is a large number of multi-FPGA-cycle processes, as this will reduce communication overhead.

# Chapter 6

# Conclusions and Future Directions

Microprocessors are applied everywhere in our life. When designing products, architects must face the challenge of planning, evaluating, and implementing multiple heterogeneous microprocessors that are capable to meet different processing requirements. To rapidly validate their ideas, microarchitectural models must be easy to construct, flexible to change, and fast to execute. The synthesis of hybrid structural microarchitectural simulators offers an opportunity to achieve all these goals. The work of this thesis takes the first step towards the synthesis of hybrid structural microarchitectural simulators and contributes a complete infrastructure (SPRI) which perfoms simulator partitioning, hardware synthesis, and interface synthesis. This thesis has researched on the interface design and synthesis techniques for synthesized hybrid structural microarchitectural simulators. It has also thoroughly evaluated the interface design space by implementing five interface generators for SPRI and comparing a set of the automatically-generated interfaces. This analysis and study have demonstrated the important design trade-offs and performance factors (i.e. hardware capacity, interface latency, interface bandwidth, simulation speed, and design scalability to simulate multiple cores) involved in choosing an efficient interface. The insights of this reseach are essential. They can lead to better decisions on how to organize a simulator, how to partition a simulator, and how to choose an interface, no matter whether architects are planning on applying SPRI to generate hybrid structural microarchitectural simulators or manually creating them from scratch.

In the future, we will continue to improve the non-blocking interface by separating communications with the hardware into a separate thread, allowing SW to continue execution during communication and by transforming portions of the SW into an LI-BDN-compatible form. We are also looking to supports other hybrid simulation platforms which have FPGAs with large capacity and scale to hybrid simulators for hundreds of cores.

## Bibliography


[1] M. Vachharajani, "Microarchitectural modeling for design-space exploration," Ph.D. dissertation, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, United States, November 2004. 2, 12

[2] *IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual*. IEEE, 2005. 2, 5, 12, 33

[3] 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. 2, 12

[4] 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. 2, 12

[5] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 249 – 261. vii, 2

[6] 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. 4

[7] *IEEE Std 1076-2000: IEEE Standard VHDL Language Reference Manual*. IEEE, 2000. 6, 21

[8] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, April 2003. 7

[9] D. Burger and T. M. Austin, "The SimpleScalar tool set version 2.0," Department of Computer Science, University of Wisconsin-Madison, Tech. Rep. 97-1342, June 1997. viii, 10, 11

[10] D. A. Penry, "The acceleration of structural microarchitectural simulation via scheduling," Ph.D. dissertation, Princeton University, Princeton, NJ, November 2006. 11, 16

[11] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The Liberty Simulation Environment: A deliberate approach to high-level system modeling," *ACM Transactions on Computer Systems*, vol. 24, no. 3, August 2006. 12

[12] D. Penry and D. I. August, "Optimizations for a simulator construction system supporting reusable components," in *Proceedings of the 40th Design Automation Conference (DAC)*, June 2003, pp. 926–931. 13

[13] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002, pp. 108–116. viii, 14

[14] 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. 14, 18, 26, 27, 43

[15] L. Eeckhout and K. De Bosschere, "Early design phase power/performance modeling through statistical simulation," in *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, 2001. 15

[16] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003. 15

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57. 15

[18] J. Namkung, D. Kim, R. Gupta, I. Kozintsev, J. Bouget, and C. Dulong, "Phase guided sampling for efficient parallel application simulation," in *Proceedings of the 3rd Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 187–192. 15

[19] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003, pp. 84–97. 15

[20] M. Chidister and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 3, pp. 176–200, July 2002. 16

[21] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood, "Wisconsin Wind Tunnel II: A fast and portable architecture simulator," in *Workshop on Performance Analysis and its Impact on Design (PAID)*, June 1997. 16

[22] A. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael, "Accuracy and speed-up of parallel trace-driven architectural simulation," in *Proceedings of the 11th International Parallel Processing Symposium*, 1997. 16

[23] P. Coe, F. Howell, R. Ibbett, and L. Williams, "A hierarchical computer architecture design and simulation environment," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 4, pp. 431–446, October 1998. 16

[24] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 330–335. 16

[25] 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. 17, 18, 29, 43

[26] 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. 17, 24, 43

[27] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsfi, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 2, June 2009. 18, 28, 43

[28] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007. 20

[29] S. Hauck and A. DeHon, *Reconfigurable Computing*. Morgan Kaufmann, 2007. 21

[30] *IEEE Std 1364-2001: IEEE Standard Verilog Hardware Description Language*. IEEE, 2001. 21

[31] "Xilinx XUP-V5 development system home page," http://www.xilinx.com/univ/xupv5-lx110t.htm. 24

[32] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanović, "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, March-April 2007. 24

[33] "DRC coprocessor system user guide," 2006. 24

[34] "Intel Xeon FSB FPGA accelerator module," http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html. 24, 26

[35] "FPGA-accelerated simulation technologies (FAST)," http://users.ece.utexas.edu/ derek/-FAST.html. 27

[36] E. C. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using memoization," in *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998, pp. 283–294. 27

[37] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: A performance model framework," *IEEE Computer*, vol. 0018-9162, pp. 68–76, February 2002. 27

[38] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt, "Network-oriented full-system simulation using m5," in *the 6th Workshop on Computer Architecture Evaluation Using Commerical Workloads*, 2003. 27

[39] 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. 28

[40] 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. 28

[41] M. Vachharajani, N. Vachharajani, and D. I. August, "The Liberty Structural Specification Language: A high-level modeling language for component reuse," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, June 2004, pp. 195–206. 29

[42] "DRC computer home page," http://www.drccomputer.com. 37

[43] 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. 38

[44] Z. Ruan, K. Rehme, and D. A. Penry, "SPRI: Simulator Partitioning Research Infrastructure," in *3rd Workshop on Architectural Research Prototyping*, 2008. 38

[45] T. S. Harris, Z. Ruan, and D. A. Penry, "Techniques for LI-BDN synthesis for hybrid microarchitectural simulation," in *Proceedings of the 2011 International Conference on Computer Design*, 2011, pp. 253–260. viii, 44, 46, 58

[46] 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. 44

[47] T. S. Harris, "Techniques for LI-BDN synthesis for hybrid microarchitectural simulation," Master's thesis, Brigham Young University, Provo, UT, May 2013. 45

[48] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 64–67, July 2009. 56

[49] 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. 56