Implementing a High-performance Multithreaded Microprocessor: A Case Study in High-level Design and Validation

Eric S. Chung, James C. Hoe
Computer Architecture Lab at Carnegie Mellon
Carnegie Mellon University
Pittsburgh, PA 15213
{echung, jhoe}@ece.cmu.edu

Abstract

We have developed a 16-way multithreaded microprocessor called BlueSPARC. This in-order, high-throughput processor incorporates complex features such as privileged operations, memory management, and a non-blocking cache subsystem. When supported by a hybrid simulation technique that handles rare, unimplemented behaviors in a software host, the BlueSPARC microprocessor runs unmodified UltraSPARC III-based commercial applications on Solaris 8 while hosted on a single Xilinx XCV2P70 FPGA clocked at 90MHz. This significant effort was achieved in under one man-year using a high-level language and a high-level validation approach. In the first part of the paper, we describe our experience in applying the Bluespec SystemVerilog (BSV) language to develop a large hardware design that must meet specific area and performance requirements. In the second part of the paper, we present the FPGA-accelerated validation approach we employed to check the correct execution of real multi-threaded programs running on the BlueSPARC processor. We discuss the challenges and our solutions to validation in the presence of full-system interactions and microarchitectural nondeterminism.

1. Introduction

This paper presents a case study of the high-level design and validation of a 16-way multithreaded microprocessor called BlueSPARC. BlueSPARC is not a microprocessor in the usual sense. Rather, BlueSPARC is an execution engine implemented in FPGA to accelerate an UltraSPARC III architectural-level full-system multiprocessor simulator [1] (with capabilities similar to Simics [2]). To avoid unnecessary development effort, the BlueSPARC processor only implements the common-case ISA behaviors. When supported by a hybrid simulation technique that handles the unimplemented instructions in a nearby software host, BlueSPARC is fully capable of running real operating systems and applications.

Despite its specialized usage in simulation acceleration, BlueSPARC's basic design is typical of an efficient multithreaded in-order pipelined microprocessor and supports the non-trivial subset of UltraSPARC III ISA features necessary to efficiently accelerate simulation of operating systems and commercial SPARC applications. These features include privileged-mode operations, implementation-dependent instructions, and a memory management unit. Further, the implementation is governed by specific area and performance constraints, requiring us to pay careful attention to implementation quality and to pursue aggressive microarchitectural optimizations such as a non-blocking cache subsystem.

Surprisingly, the BlueSPARC microprocessor was developed in a short amount of time—under one man-year. This can be partially attributed to our use of a high-level hardware description language called Bluespec SystemVerilog (BSV) [3]. In BSV, the behavior of a system is described as a collection of "atomic" operations in the form of rules. Throughout the development, rules reduced the amount of effort needed to capture complex concurrency scenarios, such as an implementation of the BlueSPARC processor's non-blocking cache controller. In many cases, rules allowed us to easily describe sharing of port-limited resources and also facilitated incremental design changes without undue complication. In later sections, we will discuss detailed examples of how rules and other BSV features were used to simplify the development and present cases where unexpected difficulties were encountered.

Despite the benefits of using a high-level language, nearly half of our overall development time was still spent in validating the BlueSPARC microprocessor. To develop confidence in our implementation, it was necessary to not only validate individual instructions but to also consider the wide range of interleaved behaviors possible when multiple threads execute and compete for limited resources within a single pipeline. Unfortunately, the slow performance of RTL simulation was a major limitation to fully validating the BlueSPARC processor. To overcome this, we developed an approach that validates the correct execution of multithreaded programs to completion while running on an FPGA prototype. To accommodate nondeterministic behaviors, a full-system replay method was used to maintain agreement between our implementation in FPGA and a "golden" reference simulator. Our validation method allowed us to check billions of instructions and was instrumental in closing the validation "last mile" where RTL simulation was too slow to exercise the rarest design bugs.

During the development of BlueSPARC, the use of BSV did not prevent us from paying careful attention to implementation quality. In comparison to other synthesizable FPGA-based processors based on conventional RTL flows, the BlueSPARC microprocessor achieves comparable clock frequencies without occupying an unreasonable amount of area. For example, when synthesized for a Virtex-5 LX110T FPGA, a 16-way BlueSPARC processor operates at 125MHz, and uses 34% of available logic and 85% of available SRAMs. For an approximate comparison, an OpenSPARC T1 core ported to FPGAs [4, 5] runs at up to 50MHz and consumes 74% of available logic and 78% of available SRAMs in a 4-threaded configuration. Although the T1 supports a complete ISA and implements more advanced microarchitectural features, the BlueSPARC processor shares similarities such as multithreading for throughput and salient features of a 64-bit SPARCV9-based ISA.

Outline. The rest of the paper is organized as follows. Section 2 describes the BlueSPARC microarchitecture, its design rationales, and the hybrid simulation features used to provide full ISA compliance. Sections 3 and 4 describe the high-level design and validation approaches. Section 5 covers related work, and Section 6 offers a summary and concludes.

2. Background

In the following sections, we provide an overview of the BlueSPARC microarchitecture, the hybrid simulation concepts, and a brief introduction to the BSV language. We first explain the context behind the development of BlueSPARC, its microarchitecture, and the major sources of complexity. In addition, we also introduce the reader to BSV and highlight its features most relevant to our efforts.

2.1. FPGA-Accelerated Simulation

The BlueSPARC multithreaded processor was not conceived as a stand-alone processor but as a multiple-context ISA emulation engine at the heart of an FPGA-accelerated full-system multiprocessor simulator [1]. Today's state-of-the-art software-based full-system simulators (e.g., Simics [2]) are too slow to simulate large-scale multiprocessor systems. Our project has developed a novel FPGA-accelerated functional simulator that uses hardware virtualization to attain the benefit of FPGA-acceleration without the complexity and cost of conventional large-scale FPGA prototyping.

Fig. 1 illustrates our FPGA-accelerated simulation concept. One of our first goals in functional-level simulation is to decouple the complexity of the simulated system from the true complexity of the underlying hardware platform. Rather than implementing each logical processor individually, our FPGA-accelerated simulation maps multiple logical processors onto the multithreaded BlueSPARC engine. In this paper, we present a system with only one instance of the BlueSPARC engine, which handles up to sixteen logical processor contexts.

A full-system simulator must be accurate and complete to support the execution of operating systems and unmodified binary executables. However, not all details have to be captured and accelerated in FPGA. Consequently, only the common-case ISA behaviors are implemented in the BlueSPARC processor, with the rare behaviors relegated to software simulation. In our implementation using Xilinx FPGAs [6], the BlueSPARC processor is augmented with a SPARC instruction set simulator running on an embedded PowerPC. To handle memory-mapped I/O instructions, the PowerPC communicates with an off-chip I/O device simulator based on Simics. When

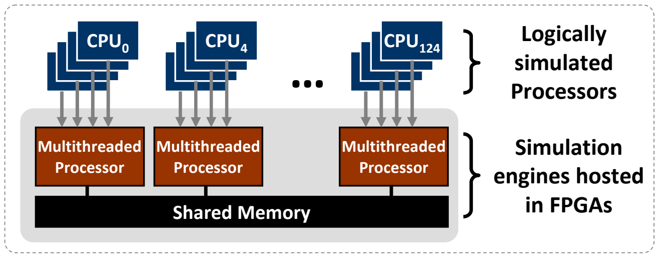


Fig. 1. Multithreaded host interleaving.

TABLE 1. Selected partitioning in hybrid simulation.

BlueSPARC (FPGA)	Integer ALU instructions Register windows General Traps, Device+IPC Interrupts I-/D- memory management unit Standard V9 privileged instructions Memory instructions + atomics
On-chip simulation (On-FPGA PowerPC macro)	Multimedia instructions Floating Point instructions Rare TLB operations (e.g., shootdown)
Off-chip simulation (Simics)	I/O devices

supported by this hybrid simulation approach, BlueSPARC presents a fully-compliant UltraSPARC III abstraction to the OS and applications of the simulated system.

Table 1 summarizes the common-case subset of the UltraSPARC III that is supported natively in BlueSPARC versus the remaining instructions and behaviors relegated to embedded software simulation and off-chip Simics. Notice that the common subset supported by BlueSPARC does include many privileged instructions and a large portion of the memory management unit. Also note that all floating-point-related features are software-simulated in the version of BlueSPARC presented in this paper.

The final implementation statistics are shown in Table 2. While hosted on a Xilinx Virtex-II Pro 70 FPGA on the BEE2 multi-FPGA platform [7], BlueSPARC in combination with hybrid simulation successfully runs a 16-CPU Solaris 8 image, and can execute various multithreaded workloads such as the Oracle TPC-C benchmark. A more detailed workload

TABLE 2. Final BlueSPARC statistics.

Processing Nodes	16 64-bit UltraSPARC III Contexts 14-stage fine-grained multithreaded pipeline
Caches	64KB I-cache, 64KB D-cache, 64B, d-mapped Writeback, Non-blocking, alloc-on-write 16 outstanding misses, 4-entry store buffer
Speed/Capacity	90MHz / 4GB total memory
Resources (XCV2P70)	33,508 LUTs (50%), 222 BRAMs (67%)
With debug+monitors	42,206 LUTs (65%), 238 BRAMs (72%)
EDA tools	Bluespec SystemVerilog, EDK 9.2i, ISE 9.2i
Statistics	25K lines Bluespec, 511 rules, 89 modules

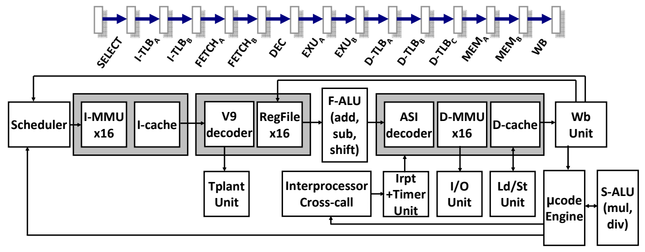


Fig. 2. BlueSPARC: A 16-CPU 14-stage instruction-interleaved multithreaded pipeline.

evaluation can be found in [1]. The BlueSPARC processor runs at 90MHz on the Virtex-II Pro 70 while only consuming 50% of the available logic. As mentioned earlier, our results are comparable to FPGA-optimized soft cores such as the Xilinx MicroBlaze [8] or the OpenSPARC T1 edition for FPGAs [4]. These competitive characteristics were made possible by significant optimizations such as deep pipelining and paying careful attention to area and timing constraints.

2.2. The BlueSPARC Microarchitecture

When considered by itself, BlueSPARC is a highly threaded microprocessor that supports up to 16 independent 64-bit UltraSPARC III V9 processor contexts. Each context maintains its own register file (with eight register windows each) and a complete set of ancillary and privileged registers. Other structures such as the TLBs are also replicated for each context. All of the contexts share and are interleaved onto a single 14-stage pipeline, where only one outstanding instruction from each context is permitted to execute at any given time [9]. To maintain a shared-memory abstraction, all of the threads share a single-level I- and D-cache.

Fig. 2 illustrates the high-level pipeline organization. Starting at the left, a scheduler issues ready instructions into the pipeline using a round-robin scheduling order (to prevent thread starvation). As an instruction passes through the pipeline, a unique context identifier is associated with it and used to index various structures and registers that are replicated (e.g., register files, TLB). In the common-case, simple instructions complete at the writeback stage, and its associated thread is reconsidered for another round of scheduling.

A number of long-latency conditions can cause a thread to be de-scheduled from the pipeline; these operations are non-blocking with respect to other threads, which can continue to utilize the pipeline. These conditions could arise from misses in the I- and D-cache as well as a number of instructions that require complex multi-cycle handling. In the next section, we describe these complex behaviors in more detail.

2.3. Complexities in BlueSPARC

Despite a deliberate attempt to simplify the design, BlueSPARC is still a non-trivial processor implementation due to (1) significant complexities inherent in the subset of the UltraSPARC III ISA implemented and (2) the need to support highly concurrent non-blocking execution of many threads in a pipeline. We highlight the primary sources of design complexities below.

Memory management and non-blocking caches. The UltraSPARC III memory management unit supports complex features such as software-programmable page modes, flexible TLB replacement policies, and selective TLB flush operations. These details are exercised frequently and must be supported natively in BlueSPARC. In a single-threaded pipeline, these procedures can be carried out by serializing the pipeline and handled off the critical path. In our multithreaded pipeline, to avoid severe performance penalties in several target applications, complex multi-cycle instructions used in a software TLB handler must be allowed to operate upon a TLB structure concurrently accessed by other threads.

To provide high-throughput execution, BlueSPARC implements an aggressive and complex memory subsystem where both the I- and D-cache controllers support up to 16 independent misses to main memory. These controllers are further complicated by the need to support different concurrent operations on cache arrays with limited FPGA BlockRAM[1] port bandwidth. For example, in addition to regular loads and stores, operations such as cache fills and invalidations must be supported as independent threads continue to execute.

High number of pipeline stages. To achieve high performance, it was necessary to pipeline the microprocessor into many stages. For example, over 100 BlockRAMs are accessed during TLB lookups, which makes routing at high FPGA clock frequencies a major challenge. We were unable to reduce the TLB sizes because the UltraSPARC III ISA architecturally specifies the sizes exactly [10]. To implement and route these structures at high clock frequencies, a 3-stage D-TLB lookup was used. In addition to pipelined structures, we opted to serialize the TLB and cache lookups to avoid issues associated with virtual indexing. Finally, other stages such as the functional units were heavily pipelined as well to accommodate a 64-bit datapath.

Microcoded and transplanted execution. Fig. 2 shows other components off the critical path that handle instruction behaviors outside the normal pipeline operations. For example, a microcode engine reuses the normal pipeline to carry out a large number of complex, implementation-specific instructions, that either require interaction with other FSMs (e.g., interprocessor interrupt) or require multiple passes in a pipeline to complete (e.g., a 64B BLOCK LOAD instruction which writes to eight consecutive floating-point registers). To reuse resources, the microcode engine decomposes such complex instructions into existing, implemented SPARC instructions, allowing for arbitrary computation and state updates.

One last component that deserves special mention is the transplant unit, which offloads unimplemented behaviors (such

1. BlockRAM is Xilinx terminology for on-chip dual-ported SRAM memories.

as memory-mapped I/O or rarely-used instructions) into software simulation. As noted earlier for the purpose of handling unimplemented instructions, the BlueSPARC microprocessor is augmented with a SPARC instruction set simulator running on a nearby embedded PowerPC. In the event of an unimplemented instruction (e.g., FP instruction or I/O), the transplant unit notifies the embedded PowerPC using an interrupt. Thereafter, the PowerPC will either simulate the unimplemented instruction, or in the case of memory-mapped I/O, send the request to an off-chip I/O device simulator based on Simics. Once the unimplemented behavior is carried out, the updated state is reinserted into BlueSPARC using the microcode engine mentioned earlier. Note that as transplants are being carried out, other independent threads continue to execute in the pipeline. At present, only one outstanding transplant is supported.

2.4. Bluespec Overview

As mentioned in the introduction, the BlueSPARC microprocessor was developed using the Bluespec SystemVerilog (BSV) language. The two BSV features most extensively used in our development were: (1) the ability to formulate atomic state changes using guarded rules, and (2) the ability to leverage high-level software-like abstractions.

In BSV, rules are declared using the "rule" keyword, an uninterpreted name, and a Boolean predicate. When the predicate of a rule evaluates to true, the rule becomes eligible to "fire", in which the state updates declared within its body are atomically executed in a single clock cycle. In BSV, the firing of all rules must appear to execute sequentially without overlaps, i.e., there are no conflicts with other rules. Conflicts potentially occur when two or more rules are eligible to fire in the same clock cycle and specify updates to the same state. To facilitate such shared updates between rules, the BSV compiler is responsible for mapping rules into clock cycles in a synchronous RTL design while obeying the atomicity requirements. When possible, BSV generates hardware that schedules as many possible conflict-free rules to fire in a given cycle. In addition to the scheduling of rules, BSV automatically infers the necessary control logic and datapath to arbitrate concurrent accesses to shared state, for example, by adding a MUX to a register written to by two separate rules.

Finally, in addition to atomic rules, BSV provides a variety of high-level programming language features, such as parameterized types and structs, iterators, containers, and functional programming language structures. The high-level abstraction of the language allows for symbolic and highly parameterizable descriptions of hardware.

2.5. Example of Design with Rules

As mentioned previously, the BlueSPARC microprocessor focuses on high-throughput execution by implementing a non-blocking cache controller, with support for up to 16 independent cache misses. Fig. 3 illustrates a simplified BSV example focusing on the three independent logical operations (*lookup*, *cache fill*, and *flush*) the cache controller must support. The *lookup* operation is executed on most cycles as threads proceed through the memory stages of the BlueSPARC processor's pipeline. The *cache fill* operation occurs when the lookup operation misses in the cache and a request to main memory is initiated. Finally, the *flush* operation is used to handle invalidation of cache blocks.

To activate any of these operations, the cache controller logic asserts various signals shown in each of the rules' predicates (e.g., *doLookup*, *doTagcheck*, *doFlush*, etc.). Note that it is desirable to support these operations concurrently without unnecessarily blocking each others' operation. The operations must also share the bandwidth-limited data and tag arrays, each with only one read port and one write port (a limitation of Xilinx BlockRAMs[2]).

In our cache controller example, memory ports are limited resources. To represent memory ports in our example, we utilize BSV methods. An example of a method is shown in the *TagCheck* rule of Fig. 3 (left) where an entry in the data array is written to using a single cache port (*data_arr.write(address, data)*). While BSV methods appear to look like functions with unclear meaning in hardware, they simply represent interfaces to other modules through a collection of data and control wires. When the *data_arr.write(...)* method is called during the firing of a rule, the *wr_en* signal is asserted and the *wr_addr* and *wr_data* buses will receive the arguments specified by the rule. When two or more rules use the same *data_arr.write(...)* method, the BSV compiler automatically inserts control logic and multiplexers to arbitrate the shared use of *wr_addr* and *wr_data*.

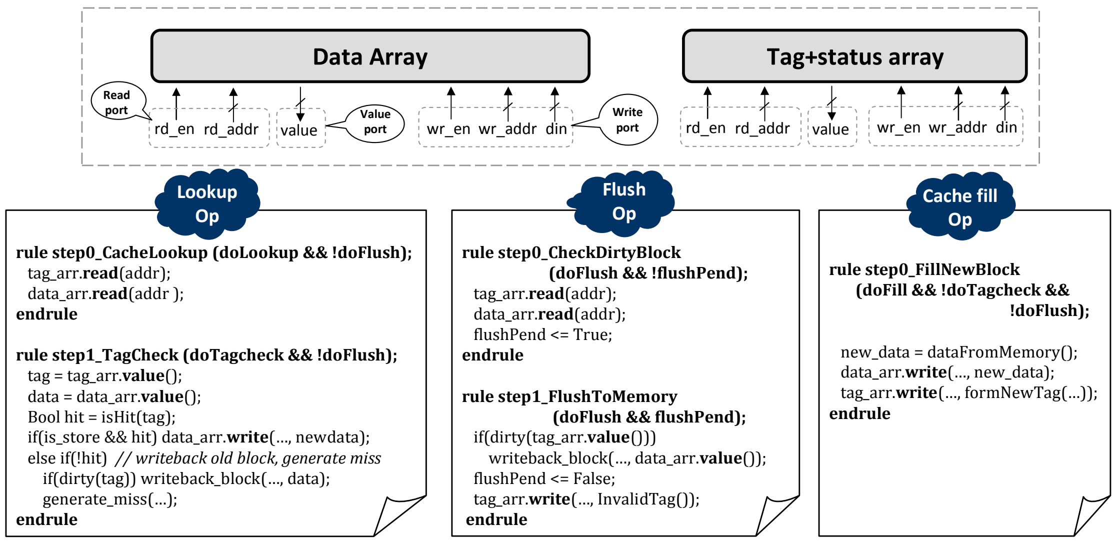
3. BSV Design Methodology

In the following sections, we discuss how rules helped and where they introduced unexpected difficulties.

3.1. Designing with Rules

Improving code readability and flexibility. While developing our cache controller, a benefit we found with using BSV is how methods and rules provided us with a level of abstraction closer to a literal translation of our specification. In the example shown in Fig. 3, note how the body of each rule represents an isolated logical operation, in a manner that is close to the high-level cache specification. For example, the cache *lookup* operation consists of two rules (operating on adjacent clock cycles) where the first rule initiates a synchronous read to the memories, while the second rule performs a tag check (and generates a miss request to main memory, if necessary). In all of the rules, both of the cache tag and data methods

2. Note, since we map the caches onto BlockRAMs, each of the cache reads must be synchronously split across two cycles (i.e., the *value* bus is only valid one cycle after *rd_en* is asserted in Fig. 3). This is an inherent limitation of Xilinx BlockRAMs.



```
rule step0_CacheLookup (doLookup && !doFlush);
  tag_arr.read(addr);
  data_arr.read(addr );
endrule

rule step1_TagCheck (doTagcheck && !doFlush);
  tag = tag_arr.value();
  data = data_arr.value();
  Bool hit = isHit(tag);
  if(is_store && hit) data_arr.write(..., newdata);
  else if(!hit)  // writeback old block, generate miss
     if(dirty(tag)) writeback_block(..., data);
     generate_miss(...);
endrule
```

```
rule step0_CheckDirtyBlock
              (doFlush && !flushPend);
  tag_arr.read(addr);
  data_arr.read(addr);
  flushPend <= True;
endrule

rule step1_FlushToMemory
              (doFlush && flushPend);
  if(dirty(tag_arr.value()))
     writeback_block(..., data_arr.value());
  flushPend <= False;
  tag_arr.write(..., InvalidTag());
endrule
```

```
rule step0_FillNewBlock
     (doFill && !doTagcheck &&
                       !doFlush);

  new_data = dataFromMemory();
  data_arr.write(..., new_data);
  tag_arr.write(..., formNewTag(...));
endrule
```

Fig. 3. Multiple concurrent logical cache operations share a cache memory with one read and one write port.

are accessed as if in isolation. Clearly, however, there are not enough ports (or "methods") that allow all of the rules to fire concurrently. Nevertheless, the rule-based semantics of BSV allow us to describe each of the scenarios independently, as if the rest of the system was "frozen".

Another benefit we found with rules was the ease in which we could add or remove rules with minimal changes to existing code. For example, while developing our cache controller, the additional features developed later in the design cycle (e.g., cache flush) required only minimal changes to the existing lookup and fill operations. Often, the only modifications necessary to existing rules were the predicates that determine the rule-firing conditions. In general, we found this to be true as well for other complex components in the BlueSPARC processor, such as the memory management unit or the microcode engine. Using BSV, the incremental bring-up of new behaviors was facilitated by simply adding new additional rules to an existing module.

Rule overhead cost. An initial concern we had with BSV is whether rules, methods, and high-level features could potentially limit low-level area and timing optimizations. Without fully understanding the abstraction of methods and rules, it was certainly possible to design sub-optimal circuits. However, as mentioned previously, the semantics of methods are clearly defined, and so long as designers remember what they represent (a bundle of data and control wires), then observing their timing and area implications remain clear. Furthermore, in the case of rules, we were able to roughly estimate (and confirm through synthesis) the overhead of automatically generated signals and datapath muxes by simply examining a shared sequential element (e.g., register) and counting the number of rules that independently write to it. We found that to the first order, the size of the multiplexer and the number of control inputs scales with the number of rules writing to that element.

Rule scheduling. Another concern with BSV is that rules could be selected to fire in a nondeterministic order, and that designers give up "control" of the dynamic ordering of events. In a scenario where rules have insufficiently guarded predicates (i.e., rules are not mutually exclusive), BSV does resort to an arbitrary static priority. We found that in most cases, allowing the BSV compiler to infer the scheduling automatically led to unexpected behaviors and bugs. When rules conflict and an arbitrary decision must be made, the BSV compiler will issue warnings that indicate unanticipated hardware behavior. In our cache controller example, we ensured that all rules are conflict-free and clearly define a preferred order of logical operations. For example, in Fig. 3, if both the *doLookup* and *doFlush* signals are asserted, all cache flush rules would take precedence over all cache lookup rules and avoid a possible memory port conflict.

The exact formation of rule predicates has implications for both functional correctness and performance. Consider the example in Fig. 3 where all of the lookup rules are prioritized over the cache fill rules. When load or store instructions are in the tag check stage of the pipeline, cache fills are automatically suppressed. While this opportunistically avoids stalling the pipeline to satisfy a fill, pathological code sequences could arbitrarily delay cache fills. A correct implementation must guarantee that the cache fill operation will eventually have an opportunity to fire its rules. In our example, we resolve this by having the pipeline detect long-waiting memory acknowledgements and temporarily de-asserting cache lookups to allow the fill to complete.

Retaining microarchitectural control. Although rules provide a higher level of abstraction over conventional RTL languages, we did not find this to be a limiting factor when developing the specific microarchitecture we had in mind. In BSV, all resources and datapath elements (e.g., pipeline stages) are explicitly instantiated and managed through single-cycle rules. This level of abstraction allowed us to capture fine-grained, low-level details, such as the lockstep transfer of data between pipeline stages. The use of rules also did not preclude the description of purely synchronous activity. For example, while interfacing with non-BSV components such as existing Xilinx FPGA IP blocks (e.g., DDR2 controller), it was necessary to hand-shake with existing interfaces using specific synchronous protocols. By writing our rules in an FSM-like manner, we were able to develop interfaces to non-BSV components without any major limitations.

3.2. BSV Limitations

Single-cycle rules. In BSV, the firing of a rule occurs within the boundary of a single clock cycle. While this abstraction is conceptually intuitive, not all high-level logical operations map efficiently into a single clock cycle. For example, writing a 64-byte cache block into a data array using a single rule may be simpler to reason about, but could result in prohibitive wiring and logic costs. In contrast, fragmenting the block into smaller parts may yield a more area-efficient implementation but at the complexity of having to use multiple rules operating over multiple adjacent clock cycles to complete the logical transaction.

In our experience, we found that there was a tension between the desire to simplify the hardware using fewer rules but at the cost of increased area or timing overheads. In addition, when high-level operations are spread over multiple rules, the atomicity guarantees of such high-level operations must be explicitly enforced by the designer, which in some cases, becomes similar to the manual effort needed when using conventional HDLs. Using the example in Fig. 3, we illustrate a possible atomicity violation. If the signals *doLookup*, *doFlush*, and *doTagcheck* are asserted in sequence over consecutive cycles, it would be possible to execute rules in this order: *step0_CacheLookup*, *step0_CheckDirtyBlock*, and *step1_TagCheck*. The *step1_TagCheck* rule would receive an incorrect tag value. To maintain atomicity of the high-level lookup operation, the cache controller logic must be designed in mind to prevent such scenarios. Nevertheless, despite these issues, we found that managing atomicity for multiple rules was never more difficult than having to achieve the same guarantees using a conventional HDL. Furthermore, managing multiple rules still requires less effort than micro-managing the control logic for individual hardware resources.

Obscurity of generated Verilog. One drawback we encountered was the difficulty in relating the generated Verilog code back to components in the original BSV description. As our design increased in size, the generated Verilog became more difficult to read. This made critical path timing analysis in downstream FPGA tools challenging and time-consuming. This issue was partially addressed by enabling special compiler flags to retain certain signal names, but further compiler improvements are needed to avoid the tedious inspection of generated Verilog.

Coping with long compile times. The last drawback we encountered was long compile times for large designs. A clean compilation that takes our BSV description all the way to Verilog requires 30 minutes on a 3GHz Core 2 Duo. From a debugging perspective, which requires simulation at the Verilog-level, this posed a significant overhead during the edit-simulate-debug cycle. To reduce this, we restructured the code base to facilitate incremental BSV compilation using (*synthesize*) pragmas. Such pragmas allow sub-modules to be compiled into Verilog without requiring a complete re-build. We generally found that the compiler struggles when the number of rules is large (hundreds) or when rules have weak predicates (requiring checking of mutual exclusion). One way to overcome this was to reduce the compiler's rule scheduling effort level and to improve predicates such that rules were mutually exclusive.

3.3. Other Features of BSV

Apart from rule-based synthesis, we also made extensive use of other language features to assist in our development.

Type-checking. One of our most important uses of BSV was to take advantage of the advanced type-checking system to avoid common errors such as the incorrect assignment of wires of different widths. As a result, from the beginning, we made a conscious effort to describe as much of the internal state as possible using meaningful types and structs that correspond exactly to the UltraSPARC III reference manual. In the end, we had well over 100 unique types to capture all possible state elements in the design.

This decision increased significantly the readability of the code and allowed our design to be more maintainable for future architectural additions or adjustments. For example, our SPARC V9 decoder is entirely symbolic and is as readable and easy-to-understand as the comparable sections in a software simulator. This allowed us to make changes easily to instruction behaviors and to even add special customized instructions with ease (e.g., hardware breakpoints).

Advanced uses of static elaboration. BSV supports extensive parameterization and advanced static elaboration, which allows for convenient ways to both represent and manipulate structures in hardware. Through the use of parameterizable container objects such as *Lists* and *Vectors*, we were able to instantiate and manipulate a large number of structures with very few lines of code. These containers were used extensively in both the MMU and the cache controllers to describe large collections of BlockRAMs.

We were also able to define generic containers for common datapath elements and conveniently pass their references around to modules corresponding to pipeline stages. For example, we could define a new *struct* called *Datapath*, which includes interfaces to all of the datapath components in BlueSPARC (e.g., data caches, register files, TLBs, etc.). In various modules corresponding to pipeline stages, any particular stage can conveniently access the interface belonging to a desired component. For example, the decoder module could access the register file entirely through the *Datapath* object. When introducing new datapath components, only the *Datapath* type requires modification.

Mixed-language designs. On occasion, downstream synthesis tools such as Xilinx XST [11] had difficulty identifying non-logic macro blocks (e.g., BlockRAMs) from Bluespec-generated Verilog. To address this problem, we utilized BSV's ability to import external Verilog code into BSV modules. This allowed us to re-write certain primitives (e.g., BlockRAMs, Distributed RAMs) using XST-friendly Verilog and to use them within our BSV designs. Unfortunately, the major drawback of importing Verilog is losing the ability to simulate our designs using the Bluesim simulator, which is a faster-than-RTL, rule-based simulator produced by the BSV compiler.

Assertions. A critical debugging strategy we adopted was to implement hardware-based assertions to validate invariants both in simulation and at runtime on the FPGA. While BSV provides a set of simulation-only assertions (Assert package), we developed an assertion package (HwAssert) that is synthesizable into hardware and can be serially monitored by an in-system synthesizable logic analyzer such as Xilinx Chipscope [12]. In BSV, assertions are clean in description because they exist only within rules (unlike assertions in conventional HDLs). Because of this, an assertion is only valid when its associated rule is firing. Therefore, any predicate associated with the rule is implicitly AND'ed with the assertion's Boolean expression, which avoids having the designer to describe under all circumstances when an assertion is valid. Our implementation also allowed us to group assertions into different categories (e.g., Cache, MMU) and to monitor them independently during runtime.

The assertions we used proved invaluable in detecting a significant number of bugs (ranging from functional bugs to ISA specification errors). For example, the types of assertions checked include:

- Queue overflows or underflows
- Mismatches in request-acknowledgement pairs (e.g., memory lookup/ack)
- Boundary checking of physical addresses
- Address misalignment
- Use of reserved instruction fields
- Validity of opcode combinations
- Forward progress for all the threads
- Suspicious exceptions (e.g., illegal instruction trap)
- Ensuring certain tag and status bit combinations never occur in the caches
- Logical invariants in the pipeline (e.g., the total sum of all threads at any given time must equal to 16)

Multiple clock domains. While debugging hardware, operating the BlueSPARC processor at a lower clock frequency was useful in reducing the amount of time needed to place and route the design on the FPGA. To facilitate this, the BlueSPARC processor was placed in a separate clock domain from all other components, including the memory controllers, the embedded PowerPC, as well as the processor bus. BSV allowed us to add quickly multiple clock domains without introducing unwanted synchronization errors, because the notion of clock domains is explicitly built into the type system of the BSV compiler. When partitioning rules into multiple clock domains, the compiler can then statically detect cross clock-domain violations when there is unsynchronized communication between rules.

4. High-Level Validation Approach

While the use of a high-level language such as BSV limited certain classes of common design errors that could occur, fully validating the BlueSPARC design in itself was still necessary. Functional design bugs such as specification errors, deadlocks, or accidental queue overflows were all still possible. Below, we discuss two challenges we faced while initially developing our validation strategy.

First, one of the key requirements for the BlueSPARC microprocessor (when supported by hybrid simulation) is the ability to reliably execute unmodified, commercial binaries in a full-system environment. Developing confidence in the correctness of our full-system implementation requires us to consider not only the microprocessor individually but also its interactions with other complex components such as peripheral devices and the hybrid simulation mechanisms themselves. Many full-system interactions (e.g., DMA, multiprocessor TLB shootdown) can be difficult to test with confidence, especially through manual assembly programs that only validate individual instructions. Furthermore, individual instruction validation may not be able to expose unexpected concurrency bugs when multiple threads execute and compete for limited resources within a single pipeline.

Second, a major obstacle we faced is the slow performance of detailed RTL simulations. Using Verilog RTL simulations, we were unable to check more than 100s of instructions per second while simulating the BlueSPARC processor. Ideally, simulating and validating our target workloads to completion would allow us to achieve higher confidence in the implementation. However, long benchmark lengths of up to tens or even hundreds of billions of instructions make software-based simulations impractical.

In the next section, we discuss our solution to these challenges using a fast, full-system validation technique to exercise and test the full range of behaviors observable in BlueSPARC.

4.1. In-System Validation Using FPGAs

To meet our functional validation goals while achieving acceptable performance, we developed a technique that uses the FPGA directly to accelerate the testing of multithreaded applications running on the BlueSPARC processor. To verify the correct execution of programs, a model of an UltraSPARC III-based multiprocessor system from Virtutech Simics is used as a "golden" software reference model to generate the correct output. By accelerating our testing on an FPGA and running our multithreaded workloads to completion, we expect to increase confidence in our implementation by fully exercising a wide range of behaviors possible when multiple threads interact within a processor pipeline, and in the presence of full-system interactions. (Note: conventional RTL simulation was still used during the early stage of development and when simulating small test cases for validating individual instructions.)

Before describing our technique in full detail, we first point out the key challenges in implementing this FPGA-accelerated validation approach:

- For multithreaded validation purposes, it is insufficient to run simply the same multithreaded application in both Simics and the BlueSPARC processor and to compare their outcomes. While running a multithreaded program on the FPGA-hosted BlueSPARC processor, nondeterministic sources such as uninitialized state, multiple clock domains, variability in DRAM access latencies, and asynchronous interrupts all contribute to an unpredictable interleaving of threads. To validate the output against Simics, the same thread interleavings must be duplicated by the reference multiprocessor model.
- The nondeterministic behavior also makes it difficult to reproduce reliably design bugs that only occur after long periods of execution.
- When a bug is detected, there must be sufficient information to determine the location of the error and its cause. This can be difficult given the low visibility of FPGA hardware and not knowing a priori when to capture internal signals (e.g., using a limited-window in-system logic analyzer) for post-mortem debugging.

Fig. 4 illustrates the infrastructure and flow we developed to facilitate nondeterministic multithreaded validation. Our approach employs an in-FPGA Trace Recorder (TR) that captures detailed full-system execution traces while a multithreaded application executes on an FPGA-hosted BlueSPARC processor (see Fig. 4, left). To validate against the software model, the execution traces are replayed in the reference architectural simulator to generate the correct output. Although not shown in the Figure, the BlueSPARC processor is connected to a full memory system and is also supported by the hybrid simulation mechanisms discussed in Section 2.1 used to support full-system workloads and operating systems. This allows us to test the BlueSPARC processor (and its peripheral components) under the most realistic possible conditions by running real applications on a real operating system. The actual applications we tested ranged from 16-CPU multiprogrammed SPEC2000 applications to multithreaded commercial workloads such as the TPC-C benchmark (Online Transaction Processing) in both IBM/DB2 and Oracle. More details on the applications we used can be found in [1].

To avoid the inconvenience of having to boot an operating system and having to position a workload each time for testing, we utilize the built-in mechanisms of Simics to generate checkpoints of pre-positioned workloads in the form of device

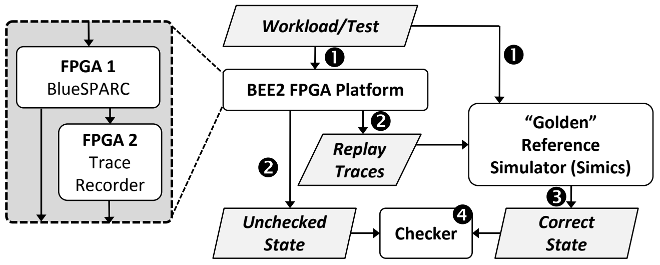


Fig. 4. Validation for the BlueSPARC processor.

state, a memory image, and 16 processors' worth of register files, TLBs, and CPU registers.

Beginning with step 1 in Fig. 4, a selected workload (in the form of a Simics checkpoint) is loaded directly into the BlueSPARC processor and its memory system to begin execution on the BEE2 FPGA platform. (The same state is also used to initialize a 16-CPU multiprocessor model hosted within a reference full-system architectural simulator for comparison purposes later). Once the state is loaded, the BlueSPARC processor is instructed to execute a finite number of instructions while delivering a continuous trace of its execution over a high-bandwidth link to a secondary FPGA (Fig. 4, left). On the secondary FPGA, TR records a runtime trace of the global instruction retirement order, including the occurrence of asynchronous events (e.g., interrupts). Overall, the type of events necessary for deterministic replay were: device interrupts, interprocessor cross-calls, timer interrupts, the value of accessed timer registers, and device-to-memory transactions (e.g., DMA). To buffer and store the instruction traces and events, the secondary FPGA maintains a circular buffer in DDR2 memory. One consequence of using DDR2 memory to store the traces is limited bandwidth during traced executions. To avoid unrealistic backpressure, the BlueSPARC processor's clock was set to a lower value (e.g., 10MHz) during validation.

Once the TR buffer reaches capacity, the BlueSPARC microprocessor pauses its execution while the trace is serially read from memory into a software test harness (step 2 in Fig. 4). To facilitate output comparison against the BlueSPARC processor, the multithreaded program running on Simics is forced to execute the interleaving called for by the traces to attain identical memory states (step 3 in Fig. 4). This was achieved by using the built-in Simics APIs for manipulating the order in which processors in a simulated 16-CPU system should execute. Other events such as interrupts or DMA were also delivered according to timestamps in the trace. After the replay period, the architectural states of both the BlueSPARC processor and the Simics simulator are compared (step 4 in Fig. 4); this includes all 16 register files, TLBs, and processor states. During traced executions, TR can also record the addresses and values for all memory instructions, which allows us to check explicitly every memory operation during validation. For a given workload, this entire process of running, replaying, and checking is repeated over and over in quantums of instructions until an application runs to completion. In general, we found that a quantum of 5000–10000 instructions between comparisons was sufficient to capture coarse-grained full-system interactions such as a full DMA operation (i.e., beginning from programmed I/O setup to the final device interrupt acknowledgement).

There is a tradeoff between (1) the quantum granularity, (2) the bug detection coverage, (3) performance, and (4) the ease of isolating a bug. Ideally, larger quantums lead to lower performance overhead, because more instructions can be checked per trace scan-out operation. However, making a quantum too large can lead to masked architectural errors, which affects the bug detection coverage. Lastly, a large quantum interval (e.g., one million instructions) means finding the actual error during mismatch can be difficult compared to a much shorter interval (e.g., 10 instructions). In the next section, we describe how we go about reliably detecting and isolating bugs when architectural mismatches occur.

4.2. Diagnosing Errors

During checking operation, if an architectural state mismatch is detected, it is necessary to identify the location and the cause of the error within the instruction stream. This process is often the most challenging and time-consuming aspect of validation because the bug may not always reliably be reproduced across multiple runs, due to nondeterminism. Therefore, our TR component includes extra information to help with "single-pass" debugging, in which enough information is extracted during the original run to facilitate post-mortem debugging.

Conventional FPGA debugging approaches often rely on in-system logic analyzers such as Chipscope to capture internal signals based off internal programmable triggers. Unfortunately, in our situation, when an architectural mismatch occurs, not only is the error difficult to reproduce, the source of the error may have occurred many thousands of cycles ago. Because tools like Chipscope are limited in sampling size (subject to the constraint of on-chip memory capacity), there is no easy way to capture the entire sequence of events that led to the erroneous behavior.

TR addresses some of these problems by recording a large amount of information per instruction during traced execution. When an architectural mismatch occurs, this extra amount of tracing information can be read out and used in post-mortem analysis, where instructions are verified individually and checked in terms of consistency with respect to other instructions (instead of just checking the final architectural state). For example, we can verify from a detailed trace that every load to a physical address returns the correct value from the prior store to the same address. We can also attempt to reproduce error conditions in Verilog simulation by recording the actual instruction words and the timestamps of when the instruction enters a given pipeline stage. These techniques allowed us to detect long-running bugs that would have been difficult to find using slower RTL simulation.

TABLE 3. Simulating 1M instructions (VCS vs. FPGA).

Synopsys VCS verilog simulator	
Traced execution	1383.9s (91.3%)
Trace extraction	0s (0.0%)
Simics replay	14.6s (1%)
Comparison	116.3s (7.7%)
Total time	1515s (25m15s)
FPGA-accelerated validation	
Traced execution	0.2s (0.9%)
Trace extraction	6.9s (30.3%)
Simics replay	15.4s (67.9%)
Comparison	0.2s (0.9%)
Total time	22.7s
Speedup over VCS	66.7X

4.3. Coverage achieved using TR

TR accelerated our validation efforts significantly when used in place of conventional software simulation debugging practices. Table 3 shows a simple experiment comparing the performance of our FPGA-accelerated validation technique (running at 10MHz) against conventional software-based RTL simulation using a commercial Verilog simulator, Synopsys VCS [13]. To simulate and validate a total of 1 million instructions for the 16-way BlueSPARC, the VCS simulator running on a 3GHz Core 2 Duo requires over 25 minutes. By comparison, the FPGA-accelerated approach is able to validate the same number of instructions in 22.7 seconds, yielding a speedup of 66.7X.

It is worth noting that in the case of FPGA-accelerated validation, the dominant component in validation time is mostly spent replaying and simulating the traced execution within Virtutech Simics (67.9% of the time). It is also worth noting that at 10MHz, very little time is actually spent executing the real application on the FPGA (0.9%) compared to the trace extraction process itself, which entails streaming of execution traces from the BEE2 FPGA to a remote workstation (30.3%).

Nevertheless, using a single BEE2 FPGA platform, we are able to validate more instructions than the combined throughput of our 50-PC cluster in the same amount of time. This technique allowed us to reach the "last mile" when even months of simulations would not be able to uncover additional corner cases caused by rare interleaving combinations and uncommon interactions involving devices.

One notable success case was the attempt to diagnose a rare memory system bug that would only occur once after running 250 million instructions in an IBM DB2 TPC-H benchmark. This memory bug would only occur if back-to-back stores to two different words in a double-word aligned address happened on consecutive cycles. TR provided enough clues and information to reliably reproduce this behavior in simulation. In total, TR allowed us to validate over 100 billion instructions, which gives us high confidence in our implementation.

4.4. Limitations of TR

A limitation of TR is that it cannot detect microarchitectural bugs that lead to processor hangs—for example, if the scheduler enters an illegal state and cannot issue any more instructions. Instead, we rely on synthesizable BSV assertions to identify these types of bugs. Another limitation of TR is the assumption of a sequentially consistent memory system. In a more relaxed memory model, it is more difficult to acquire matching architectural states between the hardware and a reference software model since correct load values may not be easily determined. Race recording techniques such as [14] could overcome this limitation.

5. Related Work

In this paper, we described our experience in applying a high-level hardware description language (BSV) to design and synthesize a working processor that must meet specific performance and area constraints. Although a number of processor designs have been implemented using BSV [15, 16, 17, 18], BlueSPARC is the first to be both built using BSV and validated to the extent of being able to run real applications and commercial operating systems.

There are many languages and design flows that support a higher level of hardware design abstraction beyond conventional register-transfer-level and behavior-level synthesis from Verilog or VHDL. For example, languages like SystemC [19], Catapult-C [20], and SpecC [21] retain the familiarity of sequential programming languages while generating hardware through automatic refinement and synthesis flows. In the processor synthesis domain, there are frameworks that can synthesize even more abstract ISA-level descriptions to hardware implementations based on specific microarchitectural templates [22, 23, 24].

The capability of these high-level approaches do not match the requirements of our effort, where a high degree of control and optimization of the datapath microarchitecture and operational timing is needed to meet our performance and area constraints. While low-level languages such as Verilog offer the finest degree of control, BSV still allowed us to maintain control of the microarchitecture while retaining a style of high-level description and synthesis. As noted in Section 2, the quality of our implementation, in terms of performance and cost, is on par with other examples of FPGA-based processors developed using conventional RTL synthesis flows [4].

6. Conclusion

In this paper, we presented the development of a 16-way multithreaded microprocessor called the BlueSPARC. When targeting FPGAs, BlueSPARC is competitive with commercial soft cores based on conventional RTL flows. In this paper, we presented our experiences in using both BSV and a high-level, full-system validation technique to develop the BlueSPARC processor. Our experience showed that a high-level methodology can be effective in the development of high-quality synthesizable soft cores.

Acknowledgments

Funding for this work was provided in part by grants from NSF CCF-0811702, NSF CNS-0509356, C2S2 Marco Center, and SUN. We thank Xilinx for their generous FPGA and tool donations. We also thank Bluespec for their tool donations and support. We thank our colleagues in the RAMP and TRUSS projects for their interaction and feedback. We also thank SPARC International for providing their compliance tests.

References

[1] E. S. Chung *et al.*, “A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations using FPGAs,” in *FPGA’08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2008, pp. 77–86.

[2] P. Magnusson *et al.*, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[3] Bluespec, Inc. http://www.bluespec.com/products/bsc.htm.

[4] Microelectronics Group, Sun Microsystems, Inc., “OpenSPARC T1 FPGA implementation, Release 1.6 update,” 2008.

[5] T. Thatcher *et al.*, “OpenSPARC T1 on Xilinx FPGAs–Updates,” 2008.

[6] Xilinx, Inc. http://www.xilinx.com.

[7] C. Chang *et al.*, “BEE2: A High-End Reconfigurable Computing System,” *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.

[8] Xilinx, “MicroBlaze Processor Reference Guide,” 2009.

[9] B. J. Smith, “Architecture and applications of the HEP multiprocessor computer system.” in *Proceedings of SPIE - Real-Time Signal Processing IV, pages 241-248*, 1981.

[10] “UltraSPARC III Cu User’s Manual Version 2.2,” 2003.

[11] Xilinx, “Xilinx XST User Guide 9.2,” 2008.

[12] Xilinx, “ChipScope Pro.” http://www.xilinx.com/tools/cspro.htm.

[13] Synopsys. (2009) VCS. http://www.synopsys.com/Tools/Verification.

[14] M. Xu *et al.*, “A regulated transitive reduction (RTR) for longer memory race recording,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 49–60, 2006.

[15] R. E. Wunderlich *et al.*, “In-System FPGA Prototyping of an Itanium Microarchitecture,” in *FPGA’04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2004, pp. 255–255.

[16] N. Dave, “Designing a Processor in Bluespec,” *Master’s Thesis, EECS, MIT*, Jan 2005.

[17] K. Ekanadham *et al.*, “IBM PowerPC Design in Bluespec,” in *Technical Report RC24706*, 2008.

[18] F. Gruian *et al.*, “VHDL vs. Bluespec system verilog: a case study on a Java embedded architecture,” in *SAC’08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008, pp. 1492–1497.

[19] “SystemC: The Open SystemC Initiative.” http://www.systemc.org.

[20] Mentor Graphics. (2009) Catapult C. http://www.mentor.com/esl.

[21] D. D. Gajski *et al.*, “SpecC: Specification Language and Methodology,” 2000.

[22] “PEAS-III: An ASIP Design Environment,” in *ICCD’00: Proceedings of the 2000 IEEE International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2000, p. 430.

[23] O. Schliebusch *et al.*, “RTL Processor Synthesis for Architecture Exploration and Implementation,” in *DATE’04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 30156.

[24] P. Yiannacouras *et al.*, “The microarchitecture of fpga-based soft processors,” in *CASES’05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2005, pp. 202–212.