PROTOFLEX: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs

ERIC S. CHUNG, MICHAEL K. PAPAMICHAEL,
ERIKO NURVITADHI, JAMES C. HOE, and KEN MAI
Computer Architecture Laboratory at Carnegie Mellon
and
BABAK FALSAFI
Parallel Systems Architecture Laboratory
École Polytechnique Fédérale de Lausanne

Functional full-system simulators are powerful and versatile research tools for accelerating architectural exploration and advanced software development. Their main shortcoming is limited throughput when simulating large multiprocessor systems with hundreds or thousands of processors or when instrumentation is introduced. We propose the PROTOFLEX simulation architecture, which uses FPGAs to accelerate full-system multiprocessor simulation and to facilitate high-performance instrumentation. Prior FPGA approaches that prototype a complete system in hardware are either too complex when scaling to large-scale configurations or require significant effort to provide full-system support. In contrast, PROTOFLEX virtualizes the execution of many logical processors onto a consolidated number of multiple-context execution engines on the FPGA. Through virtualization, the number of engines can be judiciously scaled, as needed, to deliver on necessary simulation performance at a large savings in complexity. Further, to achieve low-complexity full-system support, a hybrid simulation technique called transplanting allows implementing in the FPGA only the frequently encountered behaviors, while a software simulator preserves the abstraction of a complete system.

We have created a first instance of the PROTOFLEX simulation architecture, which is an FPGA-based, full-system functional simulator for a 16-way UltraSPARC III symmetric multiprocessor server, hosted on a single Xilinx Virtex-II XCV2P70 FPGA. On average, the simulator achieves a 38x speedup (and as high as 49×) over comparable software simulation across a suite of applications, including OLTP on a commercial database server. We also demonstrate the advantages of

Funding for this work was provided in part by grants from the C2S2 Marco Center, NSF CCF-0811702, NSF CNS-0509356, and SUN. This work was also supported by donations from Xilinx and Bluespec.

Authors' addresses: E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, and K. Mai, Computer Architecture Laboratory at Carnegie Mellon, 5000 Forbes Ave., Pittsburgh, PA 15213; email: echung@ece.cmu.edu; B. Falsafi, Parallel Systems Architecture Laboratory, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

minimal-overhead FPGA-accelerated instrumentation through a CMP cache simulation technique that runs orders-of-magnitude faster than software.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of System—*Modeling techniques*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: FPGA, simulator, emulator, prototype, multicore, multiprocessor

ACM Reference Format:
Chung, E. S., Papamichael, M. K., Nurvitadhi, E., Hoe, J. C., Mai, K., and Falsafi, B. 2009. PROTOFLEX: Towards scalable, full-system multiprocessor simulations using FPGAs. ACM Trans. Reconfig. Techn. Syst. 2, 2, Article 15 (June 2009), 32 pages. DOI = 10.1145/1534916.1534925. http://doi.acm.org/10.1145/1534916.1534925.

1. INTRODUCTION

The rapid adoption of parallel computing in the form of multicore chip multiprocessors has re-energized computer architecture research. The research focus has now shifted toward architectural mechanisms for enhancing programmability and scalability in future, highly concurrent computing platforms. Architectural research of this kind will require close collaborations between hardware and software researchers. To codevelop new systems successfully, two conflicting dependencies must be resolved. First, software researchers cannot afford to wait until the hardware development cycle is complete. Second, developing new hardware requires feedback from software research, which is difficult to attain before a design can be finalized. Fast and flexible functional full-system simulators will play a vital role in helping to resolve these dependences.

In the past and particularly in the uniprocessor setting, full-system functional simulators (e.g., Rosenblum et al. [1995] ,Magnusson et al. [2002], Emer et al. [2002], Bohrer et al. [2004], Bellard [2005], Martin et al. [2005], Wenisch et al. [2006], Binkert et al. [2006], Yourst [2007], Over et al. [2007], AMD [2008]) have stayed within a 100× slowdown relative to real systems. Until recently, this slowdown remained relatively unchanged and was deemed acceptable by many hardware and software researchers. However, when simulating ever-larger multiprocessor systems, the slowdown increases at least linearly in proportion to the number of simulated processors. As a result, existing functional simulators are used to simulate only tens of processors at speeds practical for functional exploration.

Prior attempts to parallelize multiprocessor simulation performance have shown that the scalability of a distributed simulation is limited if the simulated components must interact at an interval below the communication granularity supported by the underlying host system [Reinhardt et al. 1993; Mukherjee et al. 2000; Legedza and Weihl 1996; Chidester and George 2002; Penry et al. 2006]. In the particular case of parallelizing functional simulators, a fundamental tension exists between global synchronization overhead and the instruction interleaving granularity of multiple simulated processors

during round-robin scheduling [Over et al. 2007; Lantz 2008]. While it is desirable to increase the interleaving granularity, which improves cache utilization and parallelism on the host system, increasing it beyond thousands of cycles introduces unrealistic interleavings of memory events [Over et al. 2007].

In addition to the scalability challenges, software-based simulators experience dramatic slowdowns when introducing any form of detailed instrumentation. While stand-alone functional simulators are useful for positioning and setup of workloads (e.g., booting the OS), the lack of timing fidelity makes them unsuitable for architectural studies. To address this, functional simulators are often augmented with more detail, which can range from first-order single-cycle cache simulators to full-blown microarchitectural timing models. Unfortunately, even simple models can introduce significant overheads (up to 10× or more slowdown as observed by Nussbaum et al. [2004]).

1.1 The PROTOFLEX Simulation Architecture

In this article, we present the PROTOFLEX simulation architecture, which aims to leverage fine-grained parallelism in the form of FPGA acceleration to overcome both the scalability and instrumentation performance bottleneck. While FPGAs potentially overcome many of the aforementioned challenges, they also introduce problems on their own, such as extended development time and high cost of ownership. The goal of the PROTOFLEX simulation architecture is to simulate the functional execution of a multiprocessor system using FPGAs at a level of development effort and cost justifiable in a computer architecture research setting.

This goal is distinct from prototyping the accurate structure or timing of a multiprocessor system. Specifically, the requirements for the architecture are to: (1) functionally simulate large-scale multiprocessor systems with an acceptable slowdown (<100x), (2) model full-system fidelity to execute realistic workloads including operating systems, and (3) provide fast, low-overhead instrumentation for architectural and software research. In what follows, we briefly explain how PROTOFLEX achieves these goals.

Addressing Large-Scale System Complexity. Prior FPGA-based simulators–even functional simulators (e.g., Wee et al. [2007])—implemented their models in a way that maintains structural correspondence to the simulated target system and the underlying host.[1] For example, 16 independent cores would be instantiated and integrated together to deliver the functionality of a 16-way multiprocessor. While evidently feasible with small-scale systems, building an FPGA-based simulator with adherence to structural correspondence becomes difficult to justify when studying configurations with hundreds or thousands of processors.

We observe that in the case of providing fast, functional simulation, implementing the target system as-is in an FPGA is unnecessary. In the

[1]We refer to "target" as the simulated computer system being studied while the term "host" refers to the underlying software or hardware simulation infrastructure that carries out the behavior of the target system.

PROTOFLEX simulation architecture, the logical execution and resources of many processors can be virtualized onto a consolidated set of host execution engines on the FPGA. Within each engine, multiple processor contexts are interleaved onto a multiple-context high-throughput instruction pipeline. Through virtualization, the simulator architect is able to judiciously scale or consolidate the number of engines in the FPGA host, as needed, to deliver on necessary simulation performance.

Addressing Full-System Complexity. Apart from limitations in scale, prior FPGA efforts also typically forgo full-system fidelity and are limited to user-level custom-compiled applications (e.g., Öner et al. [1995]). Full-system fidelity enables a simulator to model real machines to a level of functional detail (e.g., devices) such that software, including the BIOS and operating systems, cannot make the distinction between the simulator and a real machine. This capability is supported in software-based simulators today (e.g., Rosenblum et al. [1995], Magnusson et al. [2002], Emer et al. [2002], Bohrer et al. [2004], Bellard [2005], Martin et al. [2005], Wenisch et al. [2006], Binkert et al. [2006], Yourst [2007], Over et al. [2007], AMD [2008]) and is especially important for evaluating applications on real operating systems. Implementing the same capabilities entirely in FPGAs would require detailed hardware design knowledge and effort typically beyond the resources of the simulator architect.

To address this challenge, we observe that the great majority of dynamically encountered behaviors in a full-system simulation make up a very small subset of total system behaviors. It is this small subset of behaviors that determines the overall simulation performance. To exploit this observation, the PROTOFLEX simulation architecture incorporates a hybrid simulation technique called transplanting, which allows a simulated entity such as a processor to be dynamically reassigned from FPGA hardware into software-based simulation. The simulator architect is now given the option to select only a subset of common-case behaviors for acceleration on the FPGA while still retaining the abstraction of a complete system.

Accelerated Instrumentation Using FPGAs.. A major advantage of FPGA-accelerated simulation is the ability to dynamically instrument internal execution state with minimal overheads in performance. As opposed to software-based simulators, which must divide their time between the functional simulator and the instrumentation code, multiple FPGA-resident instrumentation components can be attached to an FPGA-based simulator while operating in parallel. The ability to carry out fast functional instrumentation is beneficial for a wide range of hardware research activities such as trace generation, checkpoint creation, workload characterization, and architectural studies.

In addition to assisting with hardware research, FPGAs are a promising vehicle for fast dynamic monitoring and debugging of software programs. Conventional software techniques that either rely on functional simulators or dynamic binary instrumentation [Srivastava and Eustace 1994; Patil et al. 2004; Nethercote and Seward 2007] are forced to make a delicate trade-off between

the slowdown of the instrumented program versus the level of instrumentation detail. In fact, the extreme overheads of runtime instrumentation mechanisms have led to hardware-based instrumentation proposals such as MemTracker for debugging [Venkataramani et al. 2007], Raksha for security [Dalton et al. 2007], and Log-based Architectures for general-purpose instruction-grain monitoring [Chen et al. 2008].

As a demonstration of high-performance FPGA instrumentation, we developed a functional FPGA-based simulator of a CMP cache hierarchy that runs orders-of-magnitude faster than a comparable software-based simulator. As we describe in Section 4, the fast cache model is used in tandem with the *SMARTS* [Wenisch et al. 2006] statistical sampling technique to significantly reduce the turnaround time of microarchitectural timing studies.

Contributions. We developed an instantiation of a functional full-system, FPGA-based simulator called *BlueSPARC* that incorporates the two key concepts of the proposed PROTOFLEX simulation architecture: (1) time-multiplexed interleaving and (2) hybrid simulation with transplanting. BlueSPARC is our first step toward simulating large-scale multiprocessor systems and currently models the architectural behavior of a 16-CPU UltraSPARC III SMP server. BlueSPARC is hosted on a single 16-way multithreaded instruction-interleaved pipeline running on the BEE2 FPGA platform Chang et al. [2005], while Virtutech Simics [Magnusson et al. 2002] provides the backing software substrate during hybrid simulation. In the future, we envision combining multiple BlueSPARC pipelines to simulate even larger systems. BlueSPARC currently can execute real applications on an unmodified Solaris 8 operating system, including Online Transaction Processing (OLTP) on Oracle. Our performance evaluation shows that we achieve a 38x speedup average over comparable software simulation using Simics. In addition, our FPGA-Accelerated Cache Simulator (FACS) operates in tandem with BlueSPARC to provide fast, functional simulation of a CMP cache hierarchy with negligible overheads in performance (less than 4% on average).

Outline. Section 6 describes work related to FPGA-based simulation and prototyping. Section 2 presents the PROTOFLEX simulation architecture in detail. Section 3 presents an instantiation of the PROTOFLEX simulation architecture called the BlueSPARC simulator. Section 4 describes an example of fast hardware instrumentation through the design and implementation of an FPGA-accelerated CMP cache simulator. Section 5 presents an evaluation and a performance model for the BlueSPARC simulator. We conclude in Section 7.

2. PROTOFLEX

In this section, we expand on the PROTOFLEX simulation architecture for FPGA acceleration of functional full-system simulation. Note that PROTOFLEX is not a specific instance of simulation infrastructure but a set of practical approaches for developing FPGA-accelerated simulators.

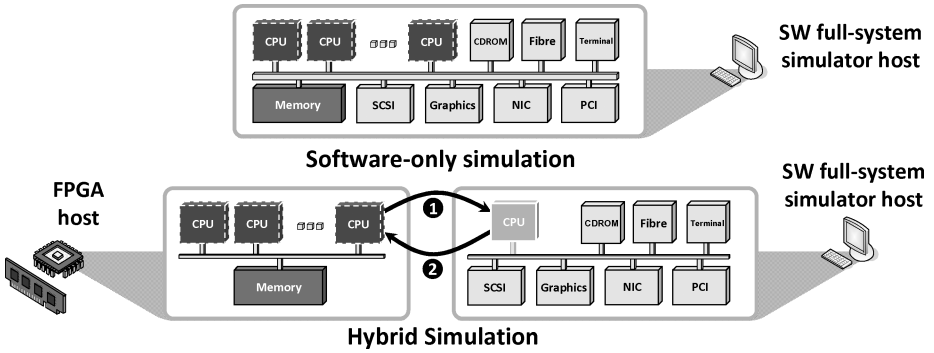

Fig. 1. Partitioning a simulated target system across an FPGA and software simulator during hybrid simulation.

2.1 Hybrid Simulation

To reduce the complexity of full-system support, we developed a new technique called hybrid simulation that aims to reduce the implementation effort of FPGA-based simulators. In hybrid simulation, components in a simulated system are selectively partitioned across both FPGA and software hosts. This technique is motivated by the observation that the great majority of behaviors encountered dynamically in a simulation are contained in a small subset of total system behaviors. It is this small subset of behaviors that determines the overall simulation performance. Thus, to improve software simulation performance while minimizing the hardware development, one should apply FPGA acceleration only to components that exhibit the most frequently encountered behaviors.

Figure 1 offers a high-level example of the hybrid simulation of a multiprocessor system. In the figure, all components are either hosted on the FPGA or simulated by the reference simulator; specifically, the main memory and CPUs are hosted in an FPGA while the remaining components are hosted in a software-based simulator (e.g., disks and network interfaces, etc.). When a software-simulated DMA-capable I/O device accesses memory, it accesses a hardware memory module on the FPGA platform. In a simulation, both the FPGA-hosted and software-simulated components are advanced concurrently to model the progress of a complete system (e.g., simulating disk activity in parallel with processors running on the FPGA).

Transplanting. A component such as the CPU can be partitioned into a small core set of frequent behaviors and an extensive set of complicated or rare behaviors. Assigning the complete set of CPU behaviors statically to software simulation or to FPGA results in either the simulation being too slow or the FPGA development being too complicated. The conflicting goals can be reconciled by supporting "transplantable" components which can be reassigned to the FPGA or software simulation at runtime. In the CPU example, the FPGA would only implement the subset of the most frequently encountered instructions. When the partially implemented CPU encounters an unimplemented

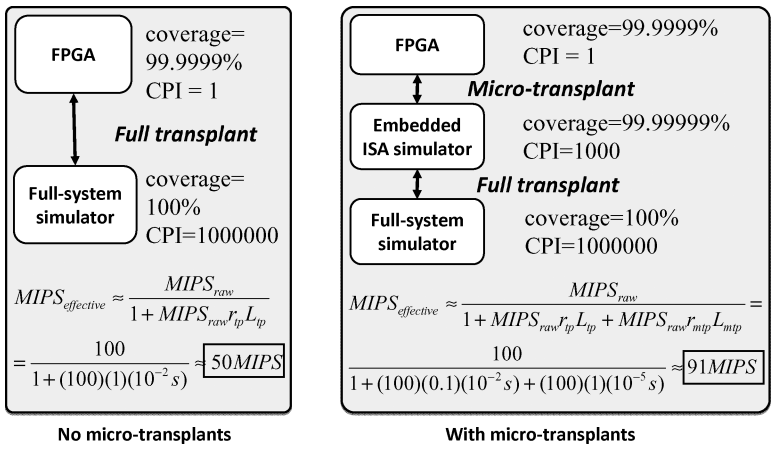

Fig. 2. Improving performance with hierarchical transplanting.

behavior (e.g., a page table walk following a TLB miss), the FPGA-hosted CPU component is suspended. Immediately after, the CPU state is "transplanted" to its corresponding software-simulated component in the reference simulator (number 1 in Figure 1). The software-simulated CPU component is activated to carry out the unimplemented behavior and deactivated again afterwards. Finally, the CPU state is transplanted back to the suspended FPGA-hosted CPU component to resume acceleration on the FPGA (number 2 in Figure 1).

Hierarchical transplanting. While transplanting is a straightforward way to achieve full behavior coverage of a complex component like the CPU, there is a significant performance overhead associated with the hand-off between FPGA and software simulation. This includes both the time to transfer data between the FPGA and the software hosts, and the time for the software simulator to carry out the unimplemented behavior on behalf of the FPGA. This exchange could take several milliseconds if the FPGA and the software simulation hosts are separated far apart (e.g., over Ethernet).

The effective hybrid simulation throughput for a 1-CPU pipeline with transplanting is

$$MIPS_{effective} \approx \frac{MIPS_{raw}}{1 + MIPS_{raw} rate_{tplant\text{-}per\text{-}million} L_{tplant}},$$

where $MIPS_{raw}$ is the FPGA-accelerated simulation throughput discounting transplant cost; $rate_{tplant}$ is the number of transplants per million instructions; and L_{tplant} is the latency cost of a transplant in seconds. Figure 2 (left) illustrates an example calculation assuming a 100 MHz FPGA-hosted CPU with CPI=1. Even if the FPGA-hosted CPU transplants just once every one million instructions, 50% of the 100 MIPS raw throughput potential would be lost due to the high transplant penalty (10msec=1 million cycles). Because $MIPS_{raw}$ also appears in the denominator, attempting to improve performance by increasing $MIPS_{raw}$ would yield diminishing returns. Attempts to raise

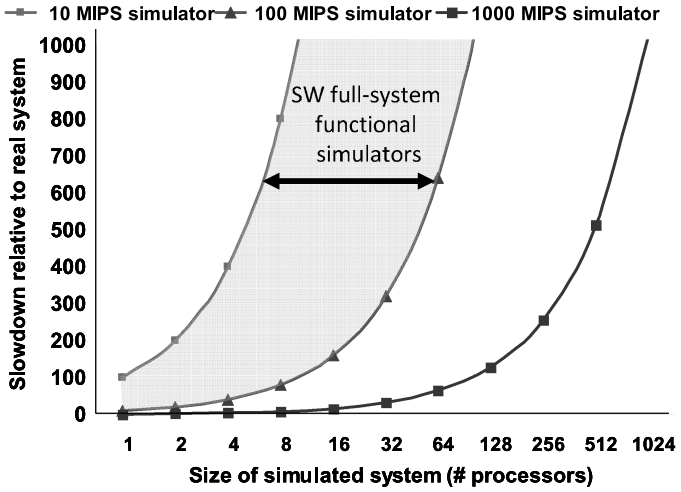

Fig. 3. Simulation slowdown versus system size (assuming a single real processor performs at 1 GIPS).

performance by reducing $rate_{tplant}$ face a different diminishing returns because increasingly more instructions would have to be built in the FPGA-hosted CPU to reduce the transplant rate further.

A better solution is to introduce a hierarchy of CPU transplant hosts with staggered, increasing instruction coverage and performance costs. For example, an embedded processor on the FPGA (e.g., the embedded PPC405 in the Xilinx Virtex architecture or even a synthesized soft core) can support a simple software-based ISA simulation kernel. The simple software-based simulation kernel would still be slow relative to the FPGA-hosted CPU, but would be much faster than a full transplant to the software simulator. The embedded simulation kernel should also be able to reach, with relative ease, a higher instruction coverage in software than in the FPGA-hosted CPU. This reduces the rate of transplants to the reference software simulator.

Returning to the example in Figure 2, suppose we introduce an embedded software-based simulation kernel as an intermediate CPU transplant host with CPI=1000. Further suppose that the intermediate simulation kernel requires transplant to the full reference simulator only once every 10 million instructions. Then, 90% of the 100 MIPS raw throughput potential would be preserved.

2.2 Interleaved Multiprocessor Simulation

In our experience, interactive software research cannot tolerate more than 100x slowdown. Batched performance simulation studies or instrumented functional simulation activites can tolerate as much as 1000x to 10,000x slowdown. Assuming three simulators with aggregate simulation throughputs of 10, 100, and 1000 MIPS, respectively, Figure 3 plots the user-perceived simulation slowdown (y-axis) of the given simulator relative to a real system. As

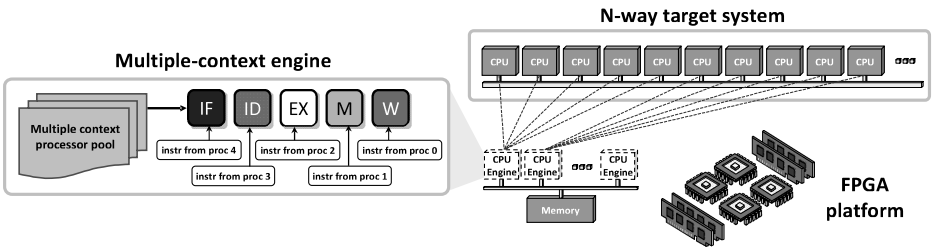

Fig. 4. Large-scale multiprocessor simulation using a small number of interleaved engines.

the system size increases, the aggregate simulation throughput is divided by the number of processors being simulated (x-axis). We assume a "real" multiprocessor system consists of 1000-MIPS processors. As seen in Figure 3, a 100-MIPS simulator represents only a 10x perceived slowdown in a uniprocessor simulation, but is only practical for studying up to tens of processors before the slowdown is no longer acceptable. Figure 3 also suggests that developing a simulator that is faster than 1000 MIPS would exceed the performance needed for effective hardware and software research of a 100-way or even 1000-way multiprocessor system.

A performance-driven approach. The straightforward approach to construct a large N-way multiprocessor FPGA-based simulator is to replicate N cores and integrate them together in a large-scale interconnection substrate. While this meets the requirement of simulating a large-scale system, developing an FPGA-based simulator with up to hundreds or even thousands of processors would be prohibitive with respect to development effort and required resources. The final aggregate throughput would also be 100 to 1000x faster than needed. Instead, a performance-driven approach would trade the excess simulation performance for a more tractable hardware development effort and cost.

The PROTOFLEX simulation architecture advocates virtualizing the simulation of multiple processor contexts onto a single fast engine through time-multiplexing. Virtualization decouples the scale of the simulated system from the required scale of the FPGA platform and the hardware development effort. The scale of the FPGA platform is only a function of the desired throughput (i.e., achieved by scaling up the number of engines). Figure 4 illustrates the high-level block diagram of a large-scale multiprocessor simulator using a small number of interleaved engines. In the figure, multiple simulated processors in a large-scale target system are shown mapped to share a small number of engines.

Selecting the engine design. When selecting the architecture for a multiple-context engine, there are two desirable requirements: (1) to exploit the concurrency between multiple threads efficiently, and (2) maximize the accuracy of the functional simulation by providing fine-grained interleaving of memory events. These requirements led us towards selecting an instruction-interleaved processor pipeline, which is well-suited for executing multiple

instruction streams efficiently. An instruction-interleaved pipeline switches a processor context on every fetch cycle, allowing for multiple processor contexts to share a single pipeline. This design enjoys several implementation advantages as exemplified in well-known multithreaded architectures such as the CDC Cyber [Thornton 1995] and the HEP barrel processor [Smith 1985]. First and foremost, the logic associated with stalling and internal forwarding can be eliminated entirely if only at most one instruction from each context is allowed in the pipeline at any given time. When doing so, longer pipelines can be used to mitigate critical timing paths, which enables better overall clock frequency. Second, an interleaved pipeline is highly tolerant to long-latency events such as cache misses and transplants since additional threads can execute in the shadow of a long-latency event. Finally, in an implementation with caches, multiple threads, running on a single engine, are automatically kept coherent in shared memory as they interleave onto a single L1 cache; in larger implementations with multiple engines this helps to consolidate the number of nodes in a cache-coherent design.

Apart from simplifying the design in FPGA, a properly scheduled instruction-interleaved pipeline is able to provide a single-cycle round-robin interleaving of memory events from multiple simulated processors without incurring a performance overhead. Software-based multiprocessor functional simulators typically face an inherent tension between the instruction interleaving granularity (called the time-slice quantum) and the performance of the simulator. In the case of single-threaded simulators, increasing the time-slice quantum improves performance by allowing the simulation to benefit from warm cache state in the host's memory hierarchy before switching to another context [Witchel and Rosenblum 1996]. In the case of parallel simulators, the time-slice quantum also dictates the interval at which multiple simulated threads are required to synchronize [Reinhardt et al. 1993; Mukherjee et al. 2000; Over et al. 2007; Lantz 2008].

Prior work has shown that parallelized simulators are scalable only when the quantum is set to sufficiently large sizes [Reinhardt et al. 1993; Mukherjee et al. 2000; Over et al. 2007; Lantz 2008]. While this still produces a functionally correct simulation and can be suitable for workload positioning, various studies [Witchel and Rosenblum 1996; Over et al. 2007] have reported unrealistic interleavings of memory events when the time-slice quantum exceeds hundreds or thousands of instructions. For example, a large quantum may result in a simulated processor acquiring a lock for an unrealistic amount of time. Unfortunately, while single-cycle interleaving is desirable, reducing the quantum limits the scalability of software-based parallelized functional simulators as well as the performance of single-threaded simulators. In the case of FPGAs, increasing the amount of instruction interleaving actually improves the efficiency of the hardware as well as overall throughput.

Scaling to multiple engines. The integration of multiple interleaved engines will be necessary to achieve scalable simulation throughput. To support shared-memory architectures, implementing cache coherency and interconnect mechanisms across multiple engines and possibly across multiple FPGAs is

Table I. BlueSPARC Interleaved Engine Characteristics

Processing Nodes	16 64-bit UltraSPARC III Contexts 14-stage interleaved pipeline
Caches	64KB I-cache, 64KB D-cache, 64B, direct-mapped Writeback, Non-blocking loads/stores, allocate-on-write 16 outstanding misses, 4-entry store buffer
Clock frequency	90MHz
Main memory	4GB total memory
Resources used on Xilinx V2P70	33,508 LUTs (50%), 222 BRAMs (67%)
Resources used with debugging+monitors	42,206 LUTs (65%), 238 BRAMs (72%)
EDA tools	Bluespec System Verilog Xilinx EDK 9.2i, XST 9.2i
Statistics	25K lines Bluespec, 511 rules, 89 module types

necessary for modeling of large-scale configurations. From estimations shown earlier, we anticipate that tens of interleaved engines running at 100 MIPS would be sufficient to achieve minimal slowdown practical for 100-way or 1000-way architecture research. Depending on available FPGA host and topology options, bus-based coherency mechanisms may be sufficient within a single FPGA. For simulation configurations requiring a large number of FPGAs, a distributed shared-memory or hierarchical bus-based architecture would be more appropriate.

In similar vein to parallelized software-based simulators, care must be taken during scaling of multiple engines to avoid having distributed engines execute unrealistic memory interleavings. As opposed to software, FPGAs allow for extremely fine-grained communication mechanisms (e.g., sending or processing a message using a state machine). This could be used to facilitate low-overhead synchronization between multiple engines. Furthermore, while multiple engines may gradually drift out of sync (for example, if different FPGAs derive their clocks from independent clock crystals), resynchronization would be relatively inexpensive and infrequent since FPGAs do not suffer the same types of workload imbalance issues that parallel software-based simulators have.

3. THE BLUESPARC SIMULATOR

3.1 Overview

In this section, we describe BlueSPARC, our first instantiation of the PROTOFLEX simulation architecture. BlueSPARC is one of our first steps towards simulating large-scale multiprocessor systems. BlueSPARC models a 16-way symmetric multiprocessor (SMP) UltraSPARC III server (Sun Fire 3800). The BlueSPARC simulator combines Virtutech Simics [Magnusson et al. 2002] on a standard PC and a single 16-context interleaved engine on an FPGA for acceleration. Table I summarizes the high-level characteristics of the BlueSPARC interleaved engine (which we explain in more detail in the coming pages). The FPGA portion is hosted on a Berkeley Emulation Engine 2

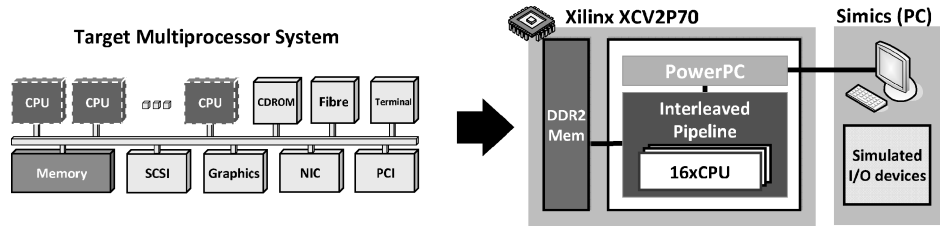

Fig. 5. Allocating components for hybrid simulation in the BlueSPARC simulator.

(BEE2) FPGA platform [Chang et al. 2005]. The BlueSPARC simulator embodies the two key concepts of the PROTOFLEX simulation architecture: (1) hybrid simulation to achieve both performance and full-system fidelity with reasonable effort and (2) decoupling the logical system size from the physical design complexity. Future work will develop the cache coherency support needed for combining multiple engines to support simulation of larger multiprocessor systems.

The BlueSPARC simulator can serve the same functionality currently provided by simulators such as Simics. In fact, the BlueSPARC simulator uses the same simulated console as Simics, so a user can even "interact" with the simulated server. Furthermore, BlueSPARC is fully capable of generating and loading Simics-compatible checkpoints of the entire simulated system state.

3.2 High-Level Organization

Figure 5 shows a high-level block diagram of how we map the functionality of the target 16-way SMP server onto software simulation and hardware hosts. First, the main memory system is hosted directly by DDR2 DRAM on the BEE2. Next, all 16 target SPARC processors are mapped onto a single interleaved engine contained on one Xilinx Virtex-II XC2VP70 FPGA. The SPARC processors are transplant-capable components. In addition to the interleaved engine, the PPC405 embedded processors serve as an intermediate transplant host for the target SPARC processors when they encounter unimplemented behaviors in the interleaved engine. The reference Simics simulator running on a PC provides the third hosting option for the target SPARC processors. Last but not least, all remaining components of the simulated system are hosted by the reference Simics simulator.

3.3 Implementation on BEE2

Figure 6 shows a structural view of our entire system mapped onto the BEE2 FPGA platform. The BEE2 board holds five Xilinx V2P70 FPGAs, which are connected to each other using low-latency, high-bandwidth 200 MHz links. The FPGA portion of the BlueSPARC simulator is hosted on two FPGAs. The center FPGA is used to host the BlueSPARC interleaved engine. This pipeline is connected over a high-speed interchip link to another FPGA, which hosts four DDR2 controllers for up to 4GBytes of main memory. In the center FPGA, the PPC405 interacts with the BlueSPARC engine over the Processor Local Bus (PLB) for servicing both transplants and I/O transactions. In addition to serving as an intermediate transplant host for the target SPARC processors,

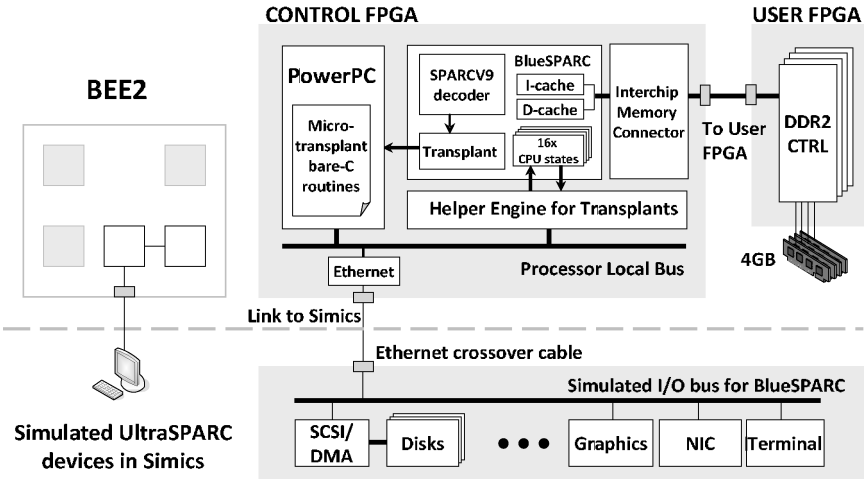

Fig. 6. High-level organization of the BlueSPARC simulator.

the PPC405 embedded processor also manages external communication with the simulation PC over Ethernet. The PC and the BEE2 are connected over a crossover Ethernet cable.

3.4 Hybrid Simulation and Transplanting

The BlueSPARC simulator fully incorporates hybrid simulation and transplanting to reduce implementation complexity. The BlueSPARC engine only supports the selected subset of the UltraSPARC III ISA required to achieve acceptable performance. We select instructions for inclusion in the FPGA based on a quantitative profile of our commercial and integer benchmarks. A general rule of thumb in the selection was to implement in FPGA the set of behaviors to limit the rate of transplants to no more than 5 times per 10,000 instructions in any of the software workloads. This rule of thumb was determined by first-order modeling of an interleaved pipeline augmented with transplanting.

In the current BlueSPARC simulator, the in-FPGA software-based simulation kernel can capture the complete behavior of the UltraSPARC III processor. The only time execution is transplanted to Virtutech Simics on the remote PC is when the CPUs interact with I/O devices. For clarity, we refer to a transplant to the in-FPGA software-based simulation kernel as a microtransplant, and we refer to a transplant to Simics as an I/O-transplant. Table II shows a detailed breakdown of how we partitioned the required behaviors of an UltraSPARC III processor for direct simulation in FPGA, microtransplant, and I/O-transplant.

As shown in Table II, the BlueSPARC engine is designed to support all of the user-level instructions as well as a subset of privileged and implementation-dependent behaviors. We found that the frequent behaviors that must be included tend also to be the simpler ones, leaving the most complicated instructions to the embedded software-based simulation kernel. It is worth mentioning that some complex behaviors do occur sufficiently frequently to

Table II. Selected Partitioning in Hybrid Simulation

BlueSPARC (FPGA)	add/sub/shift/logical/multiply/divide instructions register windows and associated traps 38/103 SPARC ASI instructions interprocessor interrupt cross-calls device and software interrupts I- and D-MMU (including software TLB) Loads, Stores, Atomics (plus inverse-endian modes) VIS block memory instructions
Micro-transplant (on-chip simulation)	65/103 SPARC ASI instructions VIS I/II multimedia instructions Floating point add/sub/mul/div and associated traps Floating point to/from integer conversion Alignment instructions Fixed-point arithmetic instructions TLB/cache diagnostic instructions TLB de-mapping operations
Transplant (off-chip simulation)	PCI bus, ISP2200 Fibre Channel, i21152 PCI Bridge IRQ Bus, Text console, SBBC PCI device Serengeti IO PROM, Cheerio-hme NIC Fibre Channel SCSI Disk, SCSI cdrom, SCSI bus

warrant implementation in the simulation engine. Despite being a RISC ISA, the UltraSPARC III superset of SPARCV9 has a large number of complex implementation-dependent and legacy behaviors (e.g., SPARC ASIs).[2] Despite these challenges, we found that our implementation was still much simpler than a full-blown prototype. Without microtransplants or I/O transplants, implementing all of the behaviors in Table II in hardware would have required a full design team (instead of just one graduate student in one year).

3.5 Interleaved Pipeline

The BlueSPARC engine is a 14-stage, instruction-interleaved pipeline that supports the multithreaded execution of up to 16 UltraSPARC III processor contexts (see Figure 7). The maximum retirement rate of our pipeline is one instruction per cycle, which in combination with the clock frequency dictates the peak simulation throughput. The primary goals in developing the BlueSPARC engine are: (1) to ensure correctness, (2) to maximize maintainability for future design exploration and (3) to minimize effort. In many cases we forgo complex performance optimizations in favor of a simpler, more maintainable design.

Pipeline operation. In Figure 7, instruction processing begins at the Context Select Unit (left), which is a simple fair scheduler that issues in round-robin order from different processor contexts. The scheduler is responsible for ensuring that all processor contexts make equal progress on average. This

[2]ASIs or "Address Space Identifiers" are used to modify the behavior of SPARC memory instructions. For example, ASIs can alter the endianness of a memory access or convert a normal memory instruction into a block transfer. ASIs instructions are also used to read and write registers in the Memory Management Unit for software TLB operations.

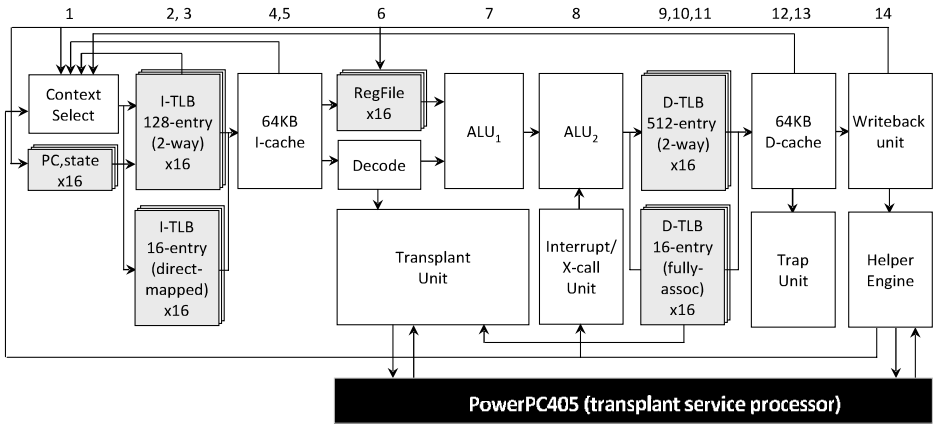

Fig. 7. The BlueSPARC engine: a 14-stage instruction-interleaved pipeline.

requirement is important in order to avoid unrealistic system behaviors (e.g., a processor holding a lock indefinitely). After being selected, each issued instruction is tagged with a unique context identifier used to index various structures and registers replicated for each processor context throughput the pipeline (see shaded components). Once the instruction is selected, a subset of its architectural registers (e.g., pstate, current window pointer) are scanned from a state register file and passed into the pipeline. After fetch, the decoding stage determines whether an instruction requires transplanting. Supported instructions in the common case proceed through the pipeline stages until reaching the writeback stage. At writeback, the processor context re-enters the Context Select Unit for the next round of scheduling. A processor context can only have one instruction in the pipeline at a time, therefore register file read-after-write hazards cannot arise.

Microtransplants and I/O-transplants. In the event a transplant operation is identified (either through the SPARC decoder or through I/O accesses in the MMU), the unimplemented instruction word and the processor context ID are queued into the Transplant unit (see Figure 7) for service by the embedded PPC405 processor running a small UltraSPARC III ISA simulation kernel. With 16 processor contexts and 14 pipeline stages, up to 2 transplanted contexts can be serviced while the remaining contexts keep the pipeline fully utilized. Only one outstanding microtransplant and one outstanding I/O-transplant are currently supported. Any additional I/O- or microtransplant requests from other contexts would be queued in the Transplant unit.

Figure 8 illustrates step-by-step how the interleaved pipeline interacts with the embedded PPC405 during a microtransplant. After an unimplemented instruction is detected and queued in the Transplant unit, the PPC405 is interrupted (1) and begins to read a minimum amount of state from the requesting context to decode the unimplemented instruction (2). After decoding, the PPC405 continues to read from the pipeline the state required to complete the unimplemented instruction. Reading and writing processor state (e.g.,

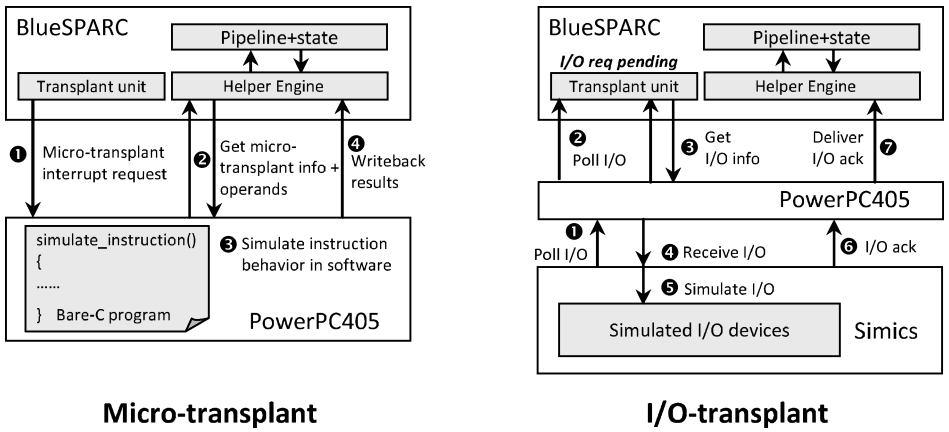

Fig. 8. Detailed operation of microtransplant and I/O-transplants.

register file and cache reads) are carried out by issuing synthetic "helper" instructions into the engine. Once the input state is acquired, the ISA simulation kernel computes the state updates (3) and writes the changed state values back to the engine for the requesting context (4)—again with the help of helper instructions—before returning the blocked processor context back to scheduling.

Figure 8 also shows how I/O-transplants are handled. During simulation, Simics continuously polls the embedded PPC405 to examine for I/O activity queued in the transplant unit (1)/(2). If an I/O transaction (e.g., a memory-mapped load or store instruction) is pending, the PPC405 acquires the necessary state information from the engine and forwards the state to Simics in response to the polling (3)/(4). For I/O transplants, Simics simulates the memory-mapped I/O bus transaction to the target simulated devices (5) and returns the acknowledgement (6)/(7).

Other sources of serialization and latency. In addition to transplants, a variety of long-latency events can deter the progress of an instruction in the pipeline. The most frequently encountered cases are both instruction and data cache misses (due to the high degree of cache sharing among 16 processor contexts). In our memory system, the caches are nonblocking, and up to sixteen misses to independent cache sets are allowed at any given time. Other contexts may continue to utilize the pipeline while memory accesses are pending. To avoid conflicts between multiple contexts for cache sets in a transient state, each cache set is augmented with a pending bit in the tag array. Any subsequent context that attempts to access a pending cache set is removed from the pipeline and forced to retry from the scheduler at a later time. To hide store misses, a 4-entry store buffer allows processor contexts to retire past a store miss; we currently do not implement store-to-load forwarding and thus any conflicting loads will stall until pending stores are drained into memory.

A number of special instructions cannot finish in a single pass through the pipeline. For example, atomics (e.g., ldstub) require a load followed by a store.

To prevent access to a locked cache line during this operation, any processor context that attempts to load or store to the same cache line is cancelled and reenters the context select unit for another round of scheduling. Other examples of multipass instructions are Quad LDDs (atomic load of two doublewords), or block load/store instructions (64B loads into 8 consecutive registers). We implement a helper engine (see Figure 7) that issues helper instructions to facilitate these rare multipass instructions. As noted earlier, the helper engine also facilitates state access during microtransplanting. Lastly, cache flushes and pipeline "freezes" can also introduce serialization into the pipeline. Because the i- and d-caches are incoherent, we conservatively flush the caches on any possibility of staleness in the i-cache (i.e., when FLUSH instructions are encountered). The caches are also flushed when DMA operations are carried out. Pipeline "freezes" occur when the entire pipeline is stalled for a temporarily owned resource (e.g., during TLB replacement).

Nondeterminism. In the current instance, BlueSPARC is a nondeterministic simulator (i.e., repeated multithreaded simulations do not necessarily yield the same interleaving order of threads). In particular, the interhost communication mechanism for hybrid simulation uses standard Ethernet across the BEE2 and a PC workstation, which is unable to provide deterministic delivery of messages. This has wide-ranging nondeterministic effects, such as the unpredictable delivery of interrupts and DMA. Future work will examine mechanisms that can provide deterministic delivery of interhost communication.

Clock frequency and area. As shown in Table I, BlueSPARC runs at a peak clock frequency of 90 MHz on a Virtex-II Pro 70. In our current design, the major critical timing paths arise from long FPGA interconnect delays when accessing the large architecturally defined I- and D-TLB structures. At present, over 100 BRAMs alone are involved during TLB lookups, which makes routing at higher clock frequencies a major challenge. In the future, we will examine techniques that reduce the on-chip storage requirements by simply caching the TLB entries of all processor contexts within a small and shared host TLB cache. We also intend to migrate towards newer FPGA families such as the Virtex-5, which offer higher logic density and lower interconnect delay using 6-input LUTs.

At present, BlueSPARC consumes a relatively large number of LUTs compared to most other available FPGA soft cores. As mentioned previously, our initial emphasis in design was correctness, and in some cases, we did not exploit area-efficient FPGA design methods. With additional effort in the future, we expect to reduce our logic consumption by a factor of two or more.

3.6 Validation

Our validation methodology involves a combination of mixed hardware/software cosimulation and testing directly on the BEE2 system. We rely on Simics to provide us with a complete and correct reference model for the UltraSPARC III server, including the prevalidated device models. In general, all of our testing (either in RTL simulation or running directly on FPGAs) must pass

validation by matching up exactly with the architectural state that Simics generates. Our validation test suite comprises a subset of ported diagnostics from the OpenSPARC framework [Vahia and Hartke 2007], the SPARC International V9 compliance tests, autogenerated stress-test programs, hand-written test cases, and real applications/benchmarks.

We relied on a variety of strategies to validate our design in the presence of nondeterminism in the interleaved pipeline. In order to match architectural state with Simics, we implement a hardware "event recorder" that records the global commit order of instructions within the pipeline for each processor context. The event recorder also captures the timing of asynchronous events during runtime (e.g., the exact point of delivery of an interrupt). This ordering is scanned from the FPGA on the BEE2 and replayed in Simics to attain identical architectural and memory state for comparison. Another validation strategy we adopted was to implement over 200 synthesizable assertion instances in our design. These assertions were monitored using the Chipscope built-in logic analyzer and responsible for detecting over 50% of the design bugs. We have successfully passed all of our test suites and have been able to validate billions of instructions in all of our workloads using the event recorder. We also have been able to interact with the hardware using a simulated console in Solaris and have been able to run console applications correctly without assertions firing or software crashes occurring. In all, we have high confidence that BlueSPARC is validated to a level suitable for active use by end-users.

In the following section, we extend the BlueSPARC implementation to support low-overhead instrumentation to accelerate microarchitectural exploration.

4. FPGA-ACCELERATED INSTRUMENTATION

A key advantage of FPGA-accelerated simulation over conventional software simulation is the support for very low overhead instrumentation, which can range from a simple set of passive counters that monitor interesting processor- or instruction-level events up to simulating a complete multiprocessor cache model. The BlueSPARC design presented previously is particularly well suited for trace-based instrumentation, where the processor pipeline generates a stream of trace information that can be processed by an independent instrumentation component. While software simulators experience dramatic slowdown during the trace generation process, an FPGA-accelerated simulator, such as BlueSPARC, is able to deliver traces to another hardware instrumentation component, possibly residing on a separate FPGA, with minimal overhead. It is worth noting that if the FPGA instrumentation component is unable to handle traces at full rate, the functional simulator must be appropriately backpressured and stalled to avoid losing trace information; this necessates careful design of the instrumentation component to avoid a performance bottleneck.

In the following section, we present an example application of FPGA-accelerated instrumentation that uses FPGA-based cache simulation to reduce the turnaround time of simulation sampling methodology.

4.1 Accelerating Simulation Sampling Using FPGAs

While fast functional simulation is an essential ingredient for accelerating architectural studies, improving microarchitectural simulation performance is also of major importance. As opposed to ongoing work such as Chiou et al. [2007] and Pellauer et al. [2008] that implement cycle-accurate timing models directly in FPGAs, we present an alternative approach that combines software simulation sampling methodology with FPGA-accelerated instrumentation, which gives accurate performance results for homogeneous multithreaded workloads. For brevity, we refer the reader to Wenisch et al. [2006] for a more detailed description of the sampling methodology and its limitations.

The sampling methodology uses statistical sampling theory to estimate the performance of applications running in software-based microarchitecture simulators. In a sampling approach, an instrumented functional simulator executes a benchmark from beginning to end while generating checkpoints of machine state at selected intervals. From each of these checkpoints, independent cycle-accurate simulations are launched in parallel and execute for a short duration. By combining together a sufficiently large number of samples, it is possible to attain an accurate mean value estimation of a chosen metric (e.g., IPC).

The main disadvantage to the sampling approach is the long turnaround times associated with instrumented functional simulation. During checkpoint creation, a functional simulator must also be instrumented to carry out warming of long-term microarchitectural structures such as caches and branch predictors. When instrumented for warming activity, full-system functional simulators typically run at around 1–2 MIPS [Wenisch and Wunderlich 2005]. For benchmarks consisting of hundreds of billions or trillions of instructions, this process can take tens of hours or up to days.

Supporting fast functional warming with FACS. To demonstrate FPGA-accelerated functional warming, we have developed an FPGA-Accelerated Cache Simulator (FACS), which functionally models a two-level Piranha-like chip multiprocessor cache with private L1 I&D caches and a shared non-inclusive L2 [Barroso et al. 2000]. This particular cache model is identical to the software-based one used for functional warming simulation in the SimFlex toolkit [Wenisch and Wunderlich 2005], which uses the aforementioned sampling-based methodology. FACS is an alternative to software-based CMP cache warming and is able to execute at nearly one or two orders of magnitude faster than its software counterpart (tens of MIPS as opposed to approximately one MIPS).

A key advantage of functional simulation is that it is only required to replicate the higher-level behaviors of the simulated target structure instead of trying to mimic low-level timing details. The implementation details of FACS that follow demonstrate how this less-constrained setting allows for simpler and more optimized designs that take full advantage of hardware-specific features, such as concurrent lookup of multiple memory blocks.

Through these practical techniques, FACS is able to run on a separate BEE2 FPGA and can receive and process dynamically generated memory address

Table III. FACS Implementation Characteristics

L1 Caches	16 private 2-way split I&D (64B blocks) (2 stage pipeline, 1 cycle/ref)
L2 Cache	8-way single shared victim cache (64B blocks) (4 stage pipeline, 2 cycles/ref)
Clock frequency	100MHz
Resources used	7483 LUTs (11%), 134 BRAMs (40%) (when configured with 64KB L1s, 4MB L2) 7277 LUTs (11%), 292 BRAMS (89%) (when configured with 128KB L1s, 16MB L2)
Statistics	2500 lines of Verilog, 8 modules

traces over a high-bandwidth inter-FPGA link from BlueSPARC at nearly full speed. For each incoming memory reference in the trace, FACS updates its L1 and L2 cache tags and maintains coherence among the private L1 caches. It is important to note that FACS has no actual functionality (i.e., there is no data stored in the caches) but simply tracks the long-term coherence state of each simulated cache block (to later export into a software cycle-accurate simulator as per the sampling methodology).

Architecture and Implementation. Before presenting the actual implementation of FACS, we first discuss additional details relevant to the target cache model. In the Piranha-like cache hierarchy, processors are connected to private split L1 I&D caches and share a common L2, which acts as a large victim cache, that is, only blocks that are evicted from L1 get inserted into the L2. Blocks are also inserted into the L2 when transferring blocks between private L1s. In addition to storing its own cache blocks, the L2 maintains a directory for all of the L1 private caches to preserve cache coherence. The L1 and L2 uses the same pseudo-LRU replacement strategies implemented in the software-based reference model.

The cache architecture described here belongs to the target simulation model and does not reflect the actual FACS implementation. FACS is a pure functional cache model, which means that it requires only storing and updating tag and status bits for each cache-line. Not storing the actual data in each cache-line greatly reduces the amount of required memory. For instance, modeling 16 64KB L1 caches, along with a single shared 4MB L2 cache requires less than 300KB of memory, which easily fits in on-chip FPGA memory resources (e.g., BlockRAMs). Moreover, FACS is fully parameterizable in the number of nodes, L1/L2 block sizes, L1/L2 cache sizes, and L2 associativity. The default configuration of FACS, which was also used to generate experimental results presented in Section 5, consists of 16 private 64KB split L1 I&D caches and a single shared 4MB L2 cache. Table III summarizes the high-level characteristics of FACS.

FACS is structured as a 6-stage pipeline, which processes the memory address trace generated from BlueSPARC through an inter-FPGA FIFO interface. The architecture of FACS is depicted in Figure 9. Internally, it consists of two core modules; one that simulates the set of all private L1s and one that simulates the larger shared L2, which are implemented as a 2-stage and a 4-stage pipeline, respectively. The L1 module can sustain the processing rate

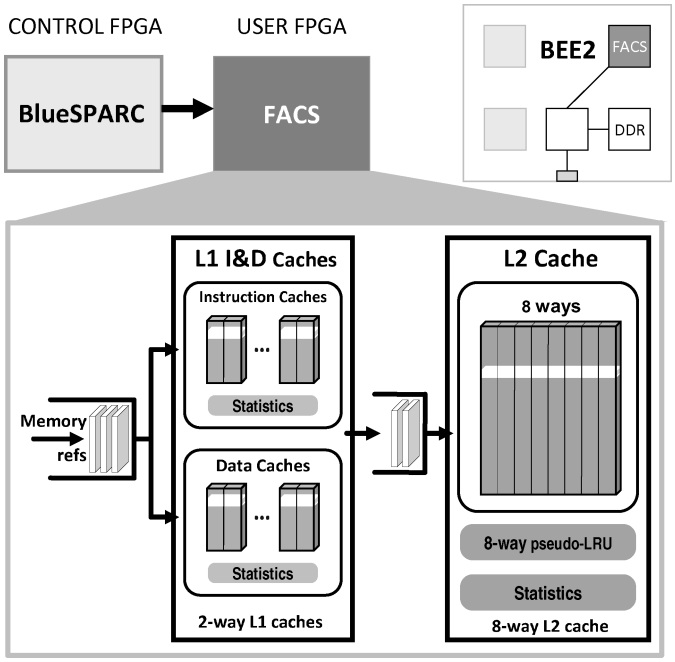

Fig. 9. FACS internal pipeline organization.

of one memory reference per cycle, while the L2 module is limited to accepting one memory reference once every two cycles. Each memory reference that is processed by the L1 may produce up to two corresponding memory references that are forwarded to the L2: one for the actual reference and one for the potentially evicted victim cache-line. However, as we show in Section 5, the rate difference between the L1 and L2 modules as well as the additionally generated L2-specific memory references have negligible impact on overall simulation performance. On the occasion that the input FIFO to FACS is full, inter-FPGA flow control measures ensure that the BlueSPARC pipeline stalls to avoid dropping any memory references.

In the target Piranha-like cache model, processors can issue multiple concurrent independent accesses to their respective private L1s, while coherence is maintained through a shared directory that resides in the L2. By taking advantage of the memory reference serialization that occurs due to instruction interleaving in the BlueSPARC simulation engine, FACS follows a different approach that is simpler, achieves high performance, and still retains functional correctness. Instead of preserving coherence by maintaining a directory in the L2, FACS concurrently accesses all of the private L1s for each memory reference, in a similar manner to snoop-based coherence protocols. This allows coherence conflicts to be resolved instantaneously, without requiring propagation to the L2 and potential subsequently generated coherence messages (e.g., invalidations) that travel back to one or more L1s. Although this approach limits the throughput of FACS, it greatly simplifies the design and completely decouples the L1s from the L2 by eliminating any potential L2-to-L1 feedback.

The L1s can process references at double the rate of the L2 module. Thus more than half of the total memory references would have to miss in the L1s on average for the L2 module to become the throughput bottleneck. In all of our workloads, the average L1 miss-rate is well below the 50% threshold; additionally the FIFO connecting the L1s and L2 modules can smooth out short bursts of consecutive L1 misses.

5. EVALUATION

This section reports an evaluation of the 16-way BlueSPARC simulator described in Section 3 as well as the FACS component described in Section 4. The performance evaluation includes a comparison to the Simics simulator. Furthermore, we present a first-order model to understand the performance of multithreaded interleaving.

5.1 Simulation Throughput

Benchmarks. The evaluations in this section are based on simulating a 16-way symmetrical multiprocessing (SMP) UltraSPARC III server (Sun Fire 3800) using BlueSPARC (with and without the FACS CMP cache simulation). The simulated server is running an unmodified Solaris 8 operating system. We exercise the simulated server using 16-way software workloads for saturating the pipeline. These consist of six SPEC2000 integer benchmarks (bzip2, crafty, gcc, gzip, parser, vortex) and a TPC-C Online Transaction Processing (OLTP) benchmark. For the SPECint workloads, 16 independent copies of the benchmark program are executed concurrently on the simulated 16-way server; we measure simulation throughput for 100 billion aggregate instructions (after the program initialization phase). Due to limited physical memory on our target configuration, it was necessary to run with test inputs and to omit benchmarks that exhibited excessive disk paging activity after program initialization (gap, mcf). Other SPECint benchmarks (eon, perlbmk, twolf, and vpr) were omitted due to a high rate (>1 in 1000) of unimplemented instructions (e.g., floating point instructions). For the OLTP benchmark, the simulated server is running a commercial multithreaded database server (Oracle 10g Enterprise Database Server) configured with 100 warehouses (10GB), 16 clients, and 1.4GB SGA as in Wenisch et al. [2006]. We measure simulation throughput for 100 billion aggregate instructions in a steady-state region (where database transactions are committing steadily). Due to nondeterminism of performance in our current implementation,[3] each bar graph we report is the average over four samples for each workload. We omit all of the standard error bars, which have values of 1% or less. As we will see next, different workload behaviors have a large impact on the simulation throughput of both Simics and BlueSPARC.

[3]Several components in our current implementation of BlueSPARC are nondeterministic. One example is the Ethernet component, which is used to facilitate I/O-transplants and DMA transfers.

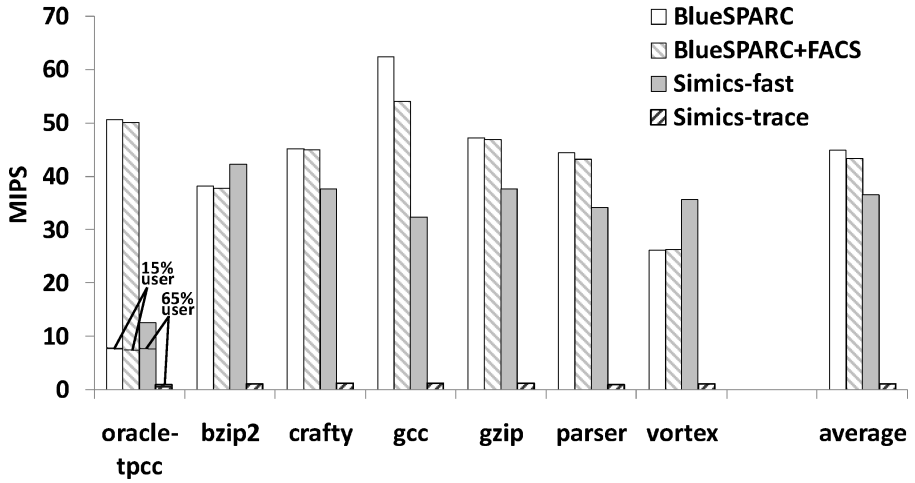

Fig. 10. Performance comparison of BlueSPARC. The height of the lines shown inside the Oracle TPC-C bars indicate the performance of Oracle TPC-C in User MIPS. The listed percentages also indicate the overall fraction of user instructions. For all remaining SPECint workloads in both BlueSPARC and Simics, the rate of privileged mode instructions is less than 5%.

Simics performance. When Simics is invoked with the default "fast" option, it can achieve many tens of MIPS in simulation throughput. However, the fast mode is a purely black-box system; that is, it does not support instrumentation (e.g., memory address tracing) or augmentation of behavior (e.g., adding a new opcode). There is at least a factor of 10 reduction in simulation throughput when Simics is enabled with trace callbacks for instrumentation (this observation is also noted by Nussbaum et al. [2004]). In Figure 10, the bars labeled Simics-fast and Simics-trace report our measured Simics simulation throughput (total simulated instructions per second) for the simulated 16-way SMP server. Simics simulations were run on a Linux PC with a 2.0 GHz Core 2 Duo and 8GBytes of memory. The performance more relevant to architecture research activities is represented by Simics-trace.

BlueSPARC performance. The simulation throughput of the BlueSPARC simulator is reported in the leftmost bars in Figure 10. The BlueSPARC simulator with acceleration from a single Virtex II XCV2P70 FPGA clocked at 90 MHz achieves speed comparable to Simics-fast on the SPECint and Oracle TPC-C workload. In comparison to the more relevant Simics-trace performance, the speedup is more dramatic, an average speedup of 38x and as high as 49x on GCC. It is worth noting that some applications such as vortex and bzip2 run slower under BlueSPARC than in Simics-fast. These applications in particular experience a high rate of memory misses under BlueSPARC. It is also worth noting that the user-to-privileged instruction ratio changes significantly in Oracle TPC-C when comparing BlueSPARC to Simics (as noted in the percentages above the Oracle TPC-C bars in Figure 10). In Section 5.2, we explain these results in detail and also discuss the major sources of performance overhead that prevent BlueSPARC from attaining ideal performance.

FACS performance. As mentioned in Section 6, fast functional instrumentation is a key capability in accelerating cycle-accurate simulation sampling activities. To demonstrate this, the second-to-leftmost bars in Figure 10 report the performance of BlueSPARC executing with FACS running in parallel on a second FPGA at 100 MHz. Overall, BlueSPARC+FACS achieves a speedup of 37x over Simics-trace. The results we report here are lower-bound speedups because we do not include a software cache simulator when measuring the performance of Simics-trace. In practice, the introduction of a CMP cache simulator (not shown in this article) further reduces the performance of Simics-trace by over a factor of two, which means BlueSPARC+FACS potentially achieves up to nearly two orders-of-magnitude performance improvement.

In the configuration described in Section 4, FACS simulates 16 independent 64KB L1 I&D caches and a shared 4MB L2 cache. At 100 MHz, FACS is able to consume up to $100M$ memory references per second. Figure 10 illustrates for the majority of our applications that FACS does not backpressure BlueSPARC's rate of progress. In the special case of GCC, a 14% reduction in performance is observed. When BlueSPARC without FACS executes GCC, the IPC of the BlueSPARC pipeline reaches 0.95 during long phases of program execution (billions of instructions). In addition, nearly 40% of all instructions are memory accesses. As a result, BlueSPARC generates $0.95 * 90MHz * 1.4 = 120M$ memory traces per second during these phases, which exceeds the peak processing rate of FACS. When FACS is introduced, BlueSPARC is backpressured during these phases and is unable to reach the original peak IPCs of 0.95, resulting in the overall 14% reduction in performance. In Vortex, BlueSPARC+FACS appears to be slightly better in performance than BlueSPARC alone. However, the error bars (not shown in the figure) for BlueSPARC and BlueSPARC+FACS overlap due to variability in runtime, and no significant difference in performance is observable.

5.2 BlueSPARC Performance Analysis

The performance model we present is based on a simple accounting of pipeline utilization by the interleaved processor contexts. Under ideal conditions, the instructions from multiple processor contexts would be issued into the pipeline in round-robin order, without stall or interference. Let N be the number of interleaved processor contexts and P be the depth of the interleaved pipeline. (In the case of the BlueSPARC design, $N = 14$ and $P = 16$.)

Let I be the average interval (in cycles) between the instructions issued from one processor context into the pipeline. The pipeline utilization by one context is then $\frac{1}{I}$, and the total utilization by N contexts is $\frac{N}{I}$. When the pipeline is 100% utilized (IPC=1), I is equal to N. With full pipeline utilization, the average instruction interval I can be broken into $P + S$ cycles. The first component (P) represents the number of cycles traversed in a P-stage pipeline. The second component (S) is the amount of cycles spent queued up in the scheduler. The S component is nonzero due to the fact that only a subset of contexts at any given moment can occupy the pipeline when $N > P$.

Table IV. Long-Latency Event Class Summary

Flush	Pipeline is halted while the caches are being flushed
Retry	A context is suspended due to a resource conflict
I/O stall	A context waits until a memory-mapped I/O operation completes
Micro-tplant	A context blocks until a micro-transplant is carried out
Load stall	A context misses in the D-cache & waits until the fill is completed
I-Fetch stall	A context misses in the I-cache & waits until the fill is completed
Scheduler	A context queueing up & being re-scheduled for another round of execution
Pipeline	The minimum amount of time needed for any instruction to complete

Consider the example of BlueSPARC. If only a single processor context is executing, the average instruction interval I is equal to 14 and the average number of cycles queued up in the scheduler S is zero. The pipeline utilization is thus $\frac{N}{I} = \frac{1}{14}$. When 16 contexts are running, each context's average instruction interval increases to $I = 16$ since each context must wait its turn for two cycles in the scheduler before running in the pipeline again. With 16 contexts, the pipeline utilization is $\frac{N}{I} = \frac{N}{P+S} = \frac{16}{14+2} = 1$.

Under real conditions, events such as cache misses, microtransplants and I/O-transplants increase the average interval I between the instructions issued from one processor context. We can express this as $I_{avg} = P + S + r_1L_1 + r_2L_2 + r_3L_3 + ... + r_iL_i$ where r_i is the rate (event per instruction) of a specific long-latency event class that delays the completion of an instruction by L_i cycles. During long-latency events, the blocked processor context is removed from the pipeline and does not consume pipeline issue slots. Thus, a high pipeline utilization can be sustained even in the presence of these long-latency events, provided: (1) $N > P$ and (2) the number of ready-to-execute contexts is at least P. A summary of long-latency event classes can be found in Table IV.

Table V gives the rate r_i (events per instruction) and average latency L_i (in 90 MHz cycles) for the event classes that impact I_{avg}. All of these statistics are collected using performance counters built into the BlueSPARC pipeline. Using the measured data as input, the I_{avg} model agrees well with wall-clock measurements and is able to predict pipeline utilizations for all of the workloads within 1% error. Figure 11 illustrates the measured contribution of each event class to the overall I_{avg}. The contribution of each component from an event class is computed as r_iL_i. The components also include contributions from time spent in the pipeline (14 cycles) and time in the scheduler (S).

Note that for workloads such as bzip2 or gcc, the time spent in the scheduler can be less than two cycles per instruction. In an otherwise perfectly scheduled pipeline with no long-latency events, this would not be possible; however, in the case of real workloads, processor contexts experiencing long-latency events can free up scheduling opportunities for others. It is also possible for contexts to experience more than two cycles per instruction in the scheduler due to stolen scheduling opportunities for retried instructions or when pipeline freezes occur (e.g., when resource hazards occur).

Based on Figure 11, the three most dominant sources of performance overhead arise from fetch and load misses, retries, and memory-mapped I/O. For the majority of the workloads, fetch and load misses contribute a significant fraction of the average instruction interval; this is expected as we interleave

Table V. Long-Latency Event Rate and Latency

	Micro-tplant	I/O	Load miss	Fetch miss	Scheduler	Retry	Flushes
Oracle	0.012% (7554)	0.0005% (668895)	2.95% (65)	1.097% (55)	100% (1.8)	10.5% (3.0)	0.0029% (3355)
Bzip2	0.000055% (3787)	0.000065% (145676)	8.6% (87)	0.173% (104)	100% (1.0)	37.3% (14.3)	0.0039% (3257)
Crafty	0.0000300% (4046)	0.000001% (162498)	7.01% (127)	2.067% (120)	100% (0.6)	13.9% (4.8)	0.00084% (3371)
GCC	0.0050% (9977)	0.000066% (195679)	1.84% (92)	0.936% (99)	100% (2.3)	12.8% (3.1)	0.0061% (3309)
Gzip	0.000046% (22628)	0.000071% (130331)	6.63% (106)	0.23% (118)	100% (1.2)	22.9% (7.3)	0.0038% (3274)
Parser	0.000173% (2990)	0.000093% (124179)	5.34% (116)	1.66% (110)	100% (2.0)	23.5% (8.0)	0.0098% (3238)
Vortex	0.00664% (2800)	0.0002682% (120747)	8.65% (153)	3.03% (114)	100% (2.2)	30.0% (15.7)	0.0095% (3243)

Percentages represent the rate of events per instruction. Numbers in parentheses represent the latency per event in units of 90 MHz cycles.

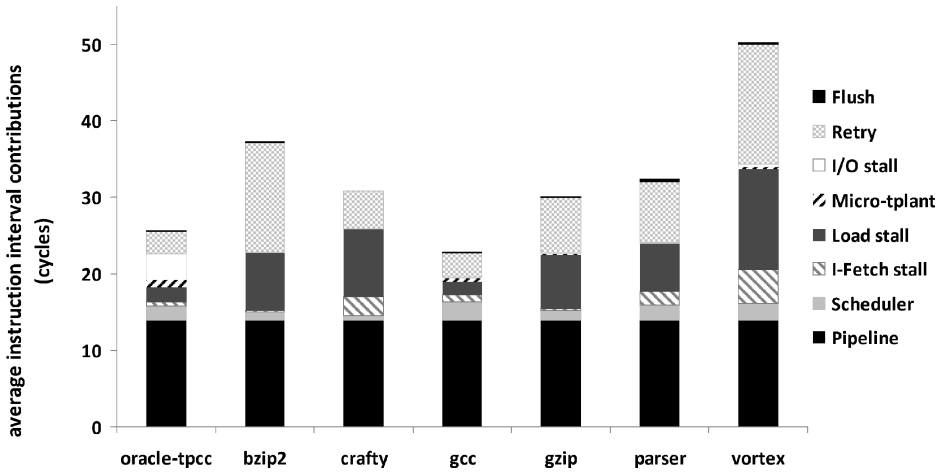

Fig. 11. Average contribution of each long-latency event class to the average instruction interval (I_{avg}). Each subcomponent within each bar is the product of the event rate (event per instruction) and the average measured latency of the event. An explanation of each of the event classes can be found in Table IV. Table V also lists the actual rates and latencies used to generate the subcomponents in each bar.

16 processor contexts onto a small, direct-mapped cache. Vortex and crafty experience the highest rate of cache misses and are also additionally penalized by increased queuing latency in the memory system. Recall that our memory controllers are hosted on a second FPGA; the interchip link between the two FPGAs limits the rate at which multiple outstanding memory requests can be serviced.

In all of the workloads, retried instructions (including discarded computation) contribute a noticeable fraction to the average instruction interval. In our measurements, the majority of retries are caused by conflicting accesses to the L1 data cache sets in a transient state (e.g., pending cache fill). Due to the direct-mapped configuration of the caches and a large window of vulnerability (e.g., long-latency cache fills), this event occurs frequently enough to contribute a significant overhead. Note how in Figure 11 that workloads with higher load and fetch stalls also experience a larger amount of retries.

Microtransplants contribute little if no overhead in almost all of the workloads. This outcome is due to the fact that we were conservative in our initial chosen instruction coverage in the FPGA; these results show that more opportunity exists to omit even further behaviors from the hardware.

As expected, none of the user-dominated SPECint benchmarks experiences any noticeable I/O. In contrast, Oracle TPC-C experiences a high level of disk activity and increased latency of I/O-transplants due to queuing of multiple outstanding I/O requests (note the higher latency in Table V).

Performance analysis of Oracle TPC-C. Despite a large I/O overhead, Oracle TPC-C running in BlueSPARC appears to have a significantly higher MIPS rate than Simics (50.5 MIPS compared to 12.5 MIPS). Further

investigation into this unexpected result reveals that the instruction-level behavior of Oracle changes dramatically when BlueSPARC is used instead of Simics (as shown above the bars in Figure 10). In particular, we observe that the rate of privileged-mode instructions when running under BlueSPARC increases to nearly 85% of all instructions. In contrast, Simics spends only up to 35% of all instructions in privileged mode. In prior work shown in Hankins et al. [2003], the user IPC is directly correlated to the rate at which database transactions are committed.

These divergences in Oracle's behavior are due to the fact that BlueSPARC introduces positive delay into the system with respect to memory-mapped I/O and DMAs. It is worth noting that Simics does not model any notion of accurate timing. All system-level events such as I/O, interrupts, or DMA occur in zero cycles. In the case of BlueSPARC, the introduction of positive I/O delay causes additional queuing for device resources in the kernel and results in increased lock contention. Increased amount of spinning in the kernel improves the IPC of Oracle running under BlueSPARC despite a lower overall database transaction throughput. Nevertheless, the verdict on whether BlueSPARC or Simics provides the more "representative" behavior of Oracle TPC-C remains unclear. In future work, we plan to compare our results against real multiprocessor systems, which will offer better insight on how representative our simulators are with respect to Oracle's runtime behavior.

6. RELATED WORK

FPGA-based simulators. Recent FPGA efforts such as Lu et al. [2007] and Vahia and Hartke [2007] have produced full-system prototypes involving a CPU implemented in an FPGA, while real motherboards and PC components are used to provide the CPU-external full-system environment. In both examples, the CPU designs are commandeered from existing RTL models developed originally for standard cell technologies. While RTL produces the most accurate timing model, the source code is generally more difficult to modify than simulation models crafted in mind for design exploration.

Other approaches such as Öner et al. [1995] and Wee et al. [2007] prototype functional multiprocessor models with approximated timing fidelity; that is, they lack cycle-accuracy but retain timing similarities to the target system. These approaches forgo full-system fidelity and also maintain structural correspondence to the target system in the number of processors implemented in the FPGA.

In similar spirit toward reducing FPGA implementation complexity, UT-FAST [Chiou et al. 2007] is a form of hybrid simulation that uses FPGAs to accelerate cycle-accurate simulation of uniprocessor microarchitectures. UT-FAST is currently divided into a software functional partition [Bellard 2005] and a timing model implemented in the FPGA. Similar work by HASIM [Pellauer et al. 2008] implements both the functional and the timing partitions within the FPGA.

In addition, there have also been a number of mature projects in the RAMP project [Wawrzynek et al. 2007], including RAMP Blue [Krasnov et al. 2007] and RAMP Red [Wee et al. 2007], both of which follow a prototyping-like methodology for modeling large-scale message-passing machines and transactional memory, respectively. More recently, the RAMP Gold project is a new effort to develop a large-scale FPGA-based simulator for manycore and data center research [Tan et al. 2008]. Currently, the RAMP Gold strategy adopts the interleaving virtualization technique demonstrated in Chung et al. [2008] and primarily focuses on efficiently maximizing thread and core count per FPGA.

Software-based functional simulators. The need for full-system simulators has led to a large number of available implementations today, including SimOS [Rosenblum et al. 1995], Virtutech Simics [Magnusson et al. 2002], ASIM [Emer et al. 2002], Mambo [Bohrer et al. 2004], QEMU [Bellard 2005], GEMS [Martin et al. 2005], SimFlex [Wenisch et al. 2006], M5 [Binkert et al. 2006], PTLSim [Yourst 2007], Sulima [Over et al. 2007], and AMD SimNow [AMD 2008].

There have been a number of approaches for parallelizing machine simulators by Reinhardt et al. [1993], Mukherjee et al. [2000], Legedza and Weihl [1996], Chidester and George [2002], Penry et al. [2006], Over et al. [2007], Lantz [2008], and Wang et al. [2008]. The Wisconsin Wind Tunnel [Reinhardt et al. 1993; Mukherjee et al. 2000], in particular, uses a time quantum approach to synchronize simulated processors across multiple host processors. The original work on Embra in Rosenblum et al. [1995] includes a parallel mode which allows translated code to execute across multiple hosts. The recent work on Parallel Embra [Lantz 2008] extends Embra to support multiple simulated processors per host processor and also improves the overall scalability of the simulator by parallelizing internal data structures. The Sparc-Sulima work [Over et al. 2007] introduces two styles of parallel simulation: an "active backplane" mode for parallelized cycle-accurate simulation and a "passive backplane," which supports a functional cache simulation using nondeterministic execution bounded by periodic global synchronization.

In general, the parallel approaches presented here all face a delicate tradeoff between scalability and accuracy. Furthermore, as mentioned previously, any software-based approach to instrumentation introduces significant slowdowns, regardless of how much parallelism is exploited between simulated processors. In terms of scalability and low-overhead instrumentation, we believe that FPGAs offer unique capabilities that overcome these common limitations.

7. CONCLUSIONS

In this article, we presented the PROTOFLEX simulation architecture for FPGA-accelerated functional full-system multiprocessor simulation. The BlueSPARC simulator is the first instantiation of this architecture that simulates a 16-way UltraSPARC III SMP server. The resulting simulator embodies the two key concepts of PROTOFLEX: hybrid simulation with transplanting and multiprocessor interleaving. Our evaluation of BlueSPARC

showed a simulation throughput as high as 62 MIPS. The evaluation further showed a significant speedup, 38x on average and 49x best case, relative to Simics with instrumentation enabled. We also demonstrated the FACS architecture, which functionally simulates a Piranha-like CMP cache model that operates in tandem with BlueSPARC while introducing less than 4% overhead in performance on average. FACS provides accelerated checkpoint generation for cycle-accurate simulation sampling methodologies and can also generate real-time cache statistics for the end-user.

In the long run, we plan to scale up the BlueSPARC simulator by combining tens of interleaved engines in order to achieve an aggregate of 1000 MIPS to support the simulations of large (>100-way) multiprocessor systems. To succeed at that scale, we also need to extend the PROTOFLEX simulation architecture with means to virtualize the required main memory capacity expected of a multiprocessor system at that scale. We are investigating the feasibility and performance impact of employing demand-paging against a larger but slower backing storage (e.g., disk or FLASH memory) to provide the illusion of the required main memory capacity. When combining multiple interleaved engines, coherence mechanisms must also be introduced to maintain the abstraction of shared-memory. We are currently investigating bus-based and distributed cache coherence protocols to achieve this.

ACKNOWLEDGMENTS

We thank our colleagues in the RAMP and TRUSS projects for their interaction and feedback. We also thank SPARC International for the use of the SPARCV9 compliance test.

REFERENCES

AMD. 2008. *Advanced Micro Devices, SimNow Simulator 4.4.3.* User's manual.

BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. *SIGARCH Comput. Archit. News 28*, 2, 282–293.

BELLARD, F. 2005. QEMU, A fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'05)*. USENIX Association, 41–41.

BINKERT, N. L., DRESLINSKI, R. G., HSU, L. R., LIM, K. T., SAIDI, A. G., AND REINHARDT, S. K. 2006. The M5 simulator: Modeling networked systems. *IEEE Micro 26*, 4, 52–60.

BOHRER, P., PETERSON, J., ELNOZAHY, M., RAJAMONY, R., GHEITH, A., ROCKHOLD, R., LEFURGY, C., SHAFI, H., NAKRA, T., SIMPSON, R., SPEIGHT, E., SUDEEP, K., HENSBERGEN, E., AND ZHANG, L. 2004. Mambo: A full system simulator for the PowerPC architecture. *ACM SIGMETRICS Perform. Eval. Rev. 31*, *4*, 8–12.

CHANG, C., WAWRZYNEK, J., AND BRODERSEN, R. W. 2005. BEE2: A high-end reconfigurable computing system. *IEEE Des. Test Comput. 22*, 2, 114–125.

CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., AND VLACHOS, E. 2008. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, 377–388.

CHIDESTER, M. AND GEORGE, A. 2002. Parallel simulation of chip-multiprocessor architectures. *ACM Trans. Model. Comput. Simul. 12*, 3, 176–200.

CHIOU, D., SUNWOO, D., KIM, J., PATIL, N. A., REINHART, W., JOHNSON, D. E., KEEFE, J., AND ANGEPAT, H. 2007. FPGA-Accelerated simulation technologies (FAST): Fast, full-system, cycle-

accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. IEEE Computer Society, 249–261.

CHUNG, E. S., NURVITADHI, E., HOE, J. C., FALSAFI, B., AND MAI, K. 2008. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA'08)*. ACM, New York, 77–86.

DALTON, M., KANNAN, H., AND KOZYRAKIS, C. 2007. Raksha: A flexible information flow architecture for software security. *SIGARCH Comput. Archit. News 35*, 2, 482–493.

EMER, J., AHUJA, P., BORCH, E., KLAUSER, A., LUK, C.-K., MANNE, S., MUKHERJEE, S. S., PATIL, H., WALLACE, S., BINKERT, N., ESPASA, R., AND JUAN, T. 2002. Asim: A performance model framework. *Comput. 35*, 2, 68–76.

HANKINS, R., DIEP, T., ANNAVARAM, M., HIRANO, B., ERI, H., NUECKEL, H., AND SHEN, J. 2003. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 151–162.

KRASNOV, A., SCHULTZ, A., WAWRZYNEK, J., GIBELING, G., AND DROZ, P. 2007. RAMP Blue: A message-passing manycore system in FPGAs. In *Proceedings of the Conference on Field Programmable Logic and Applications*.

LANTZ, R. 2008. Fast functional simulation with parallel Embra. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*.

LEGEDZA, U. AND WEIHL, W. E. 1996. Reducing synchronization overhead in parallel simulation. *SIGSIM Simul. Digest 26*, 1, 86–95.

LU, S.-L. L., YIANNACOURAS, P., KASSA, R., KONOW, M., AND SUH, T. 2007. An FPGA-based Pentium®in a complete desktop system. In *Proceedings of the ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA'07)*. ACM, New York, 53–59.

MAGNUSSON, P., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Comput. 35*, 2, 50–58.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News 33*, 4, 92–99.

MUKHERJEE, S., REINHARDT, S., FALSAFI, B., LITZKOW, M., HILL, M., WOOD, D., HUSS-LEDERMAN, S., AND LARUS, J. 2000. Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator. *Concurr. IEEE 8*, 4, 12–20.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, 89–100.

NUSSBAUM, F., FEDOROVA, A., AND SMALL, C. 2004. An overview of the Sam CMT simulator kit. Tech. rep. TR-2004-133, Sun Microsystems Research Labs.

ÖNER, K., BARROSO, L. A., IMAN, S., JEONG, J., RAMAMURTHY, K., AND DUBOIS, M. 1995. The design of RPM: An FPGA-based multiprocessor emulator. In *Proceedings of the ACM 3rd International Symposium on Field Programmable Gate Arrays (FPGA'95)*. ACM, New York, 60–66.

OVER, A., CLARKE, B., AND STRAZDINS, P. 2007. A comparison of two approaches to parallel simulation of multiprocessors. *ispass 0*, 12–22.

PATIL, H., COHN, R., CHARNEY, M., KAPOOR, R., SUN, A., AND KARUNANIDHI, A. 2004. Pinpointing representative portions of large Intel®Itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*. IEEE Computer Society, 81–92.

PELLAUER, M., VIJAYARAGHAVAN, M., ADLER, M., AND EMER, J. 2008. Quick performance models quickly: Timing-Directed simulation on FPGAs. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

PENRY, D., FAY, D., HODGDON, D., WELLS, R., SCHELLE, G., AUGUST, D., AND CONNORS, D. 2006. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors.

In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 29–40.

REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. 1993. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. *ACM SIGMETRICS Perform. Eval. Rev. 21*, 1, 48–60.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel Distrib. Technol. 3*, 4, 34–43.

SMITH, B. 1985. In *The Architecture of HEP on Parallel MIMD Computation: HEP Supercomputer and its Applications*. Massachusetts Institute of Technology, Cambridge, MA, 41–55.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, New York, 196–205.

TAN, Z., ASANOVIĆ, K., AND PATTERSON, D. 2008. An FPGA host-multithreaded functional model for SPARC v8. In *Proceedings of the 3rd Workshop on Architectural Research Prototyping*.

THORNTON, J. E. 1995. Parallel operation in the control data 6600. 5–12.

VAHIA, D. AND HARTKE, P. 2007. OpenSPARC T1 on Xilinx FPGAs–Updates. *June 2007 RAMP Retreat*.

VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y., AND PRVULOVIC, M. 2007. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, 273–284.

WANG, K., ZHANG, Y., WANG, H., AND SHEN, X. 2008. Parallelization of IBM mambo system simulator in functional modes. *SIGOPS Oper. Syst. Rev. 42*, 1, 71–76.

WAWRZYNEK, J., PATTERSON, D., OSKIN, M., LU, S.-L., KOZYRAKIS, C., HOE, J. C., CHIOU, D., AND ASANOVIĆ, K. 2007. RAMP: Research accelerator for multiple processors. *IEEE Micro 27*, 2, 46–57.

WEE, S., CASPER, J., NJOROGE, N., TESYLAR, Y., GE, D., KOZYRAKIS, C., AND OLUKOTUN, K. 2007. A practical FPGA-based framework for novel CMP research. In *Proceedings of the ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA'07)*. ACM, New York, 116–125.

WENISCH, T. AND WUNDERLICH, R. 2005. SimFlex: Fast, accurate and flexible simulation of computer systems. In *Proceedings of the Tutorial in the International Symposium on Microarchitecture (MICRO-38)*.

WENISCH, T. F., WUNDERLICH, R. E., FERDMAN, M., AILAMAKI, A., FALSAFI, B., AND HOE, J. C. 2006. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro 26*, 4, 18–31.

WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. *ACM SIGMETRICS Perform. Eval. Rev. 24*, 1, 68–79.

YOURST, M. 2007. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 23–34.

Received June 2008; revised September 2008; accepted November 2008