High-level Debugging And Verification For FPGA-Based Multicore Architectures

Abstract—**Simulators are key tools for computer architecture research. However, multicore architectures represent a highly complex challenge for software simulators, which may suffer from fidelity loss and long execution times. FPGAs can simulate multicore architectures with scalable performance and high accuracy, but the difficulty of debugging could hinder their adoption.**

In this paper we propose several techniques for inspection, debugging and verification of multicore architectures, both for software-based and FPGA-based simulations. These debugging extensions are cycle-accurate and unobtrusive. As a proof of concept, we have developed a 24-core RISC multiprocessor that runs the Linux Kernel, for which we provide three simulation modes: a fast, functional simulation; a detailed, cycle-accurate simulation; and a FPGA-based simulation. Our platform can run up to 24 cores and perform full-system verification at 17 million instructions per second.

Keywords-**FPGA; debugging; verification; multicore;**

I. Introduction

In the recent years multicore architectures have become ubiquitous, due to the power and heat dissipation implications of ever-increasing uniprocessor frequencies. Since the core mitosis, there have been significant research efforts to effectively exploit the parallelism offered by this new paradigm. However, the complexity of the architectures may exceed the capacity of traditional research tools. In particular, simulating new computer architectures is a key necessity but a highly complex problem.

When the number of cores being simulated grows, it linearly increments the workload, leading to unaffordable simulation times. Speeding up the execution through parallelization usually requires either relaxing the correctness requirements, or replacing the computation of the simulation with approximated statistical models. In both cases, the number of errors grows linearly with the number of cores.

New software simulators have been proposed to face these challenges [1], [2], [3], [4]. These new-generation tools are targeted to tens and hundreds of cores, with simulation errors within 25% and performance ranging from 0.2 to 314 Million Instructions per Second (MIPS). Different techniques have been proposed, from statistical models that elude the computations [1], [2] to interval simulation based on key architectural events [3], [4].

However, such non-cycle-accurate software simulators have several limitations, which we discuss in section VI-B, making them not suitable for novel computer architectures. At the same time, FPGA-based simulators can overcome these limitations.

It is a general consensus in the computer architecture simulation community that FPGA-based simulators are precise and fast, but when discussing this alternative the higher development effort is argued [3], [4]. The complexity of FPGA-based simulator development arises from three main sources. First, the low productivity in FPGA development, which can be addressed with High-Level Synthesis tools and new generation Hardware Description Languages (HDL). Second, long compilation times, which may be reduced from hours to minutes using overlay architectures. **Third, the notorious difficulty of inspecting and debugging hardware models, especially once running on the FPGA.**

We want to address this latter problematic. For that purpose, we developed unobtrusive debugging and verification mechanisms for software-based and FPGA-based simulation of multicore models. As a proof of concept, we developed Bluebox, a RISC multicore system written in Bluespec SystemVerilog. Our platform is highly customizable, boots the Linux kernel and supports the whole MIPS I ISA. We provide three different simulation modes for the three different stages of the development: a fast, functional software simulation for architecture validation; a slow, cycle-accurate software simulation for timing validation; and a fast, cycle-accurate FPGA-based simulation.

In this work we made the following contributions:

- We extend the Bluespec SystemVerilog simulator to implement debugging, checkpointing and verification.
- As we discuss in section V-A, host-assisted verification is not feasible. For that reason, we implement unobtrusive debugging, based on a gated clock domain, and verification on the FPGA. Our techniques respect the cycle-accuracy of the simulation and the I/O, and we use hardware to accelerate 99.5% of the verification.
- We run a 24-core simulation, achieving a full-system verification performance of 17 MIPS. We also show two examples of timing errors in a functionally correct architecture that were detected with our tools.

The rest of the paper is organized as follows. In the next section we present the Bluebox multicore architecture and its three simulation modes. In section III we extend the host-based simulation to implement debugging and checkpointing. In section IV we implement FPGA-based microarchitectural debugging and verification. In section V we evaluate the performance of our solution. In section VI we present the related work, and in section VII we conclude the paper.

II. The Bluebox Synthesizable Multicore

We developed this architecture both as a research platform and as a proof of concept of our debugging and verification techniques. The objectives when designing Bluebox were:

1) To simulate as many cores as possible.
2) To support a popular operating system and a standard Instruction Set Architecture (ISA).
3) To be completely modifiable (i.e., no private units that cannot be seen or modified by the researcher).
4) To be implemented in a high-productivity language.

Bluebox is a multicore architecture that supports the whole MIPS I ISA. Each CPU is a 5-stage, 32-bit RISC processor with arithmetic and multiplication/division units and a unified Memory Magagement Unit (MMU). These minimal characteristics ensure a reduced footprint while still being able to run any standard application and boot the Linux Kernel 2.6. At this moment the core does not include a floating point unit, because such units consume a significant amount of resources in FPGAs, but this functionality is emulated by Linux natively. We are considering implementing area-efficient solutions such as shared FPUs between cores.

Each CPU has independent instruction and data caches. The memory hierarchy implements a MSI (Modified-Shared-Invalid) coherency protocol. The CPU and the caches conform one core, as shown in Figure 1. A level 2 cache manages the coherency. All the cores are interconnected using a ring bus. We chose this topology because its short interconnects from core to core are better suited for FPGAs, increase the frequency and ease placing and routing.

One important aspect of Bluebox is that most of its characteristics are parameterizable. Most of the instruction subsets of the CPU can be disabled to reduce the footprint, such as multiply/divide; byte, half-word and unaligned memory accesses; virtual memory; and exceptions and interrupts. The pipeline supports several optimizations which can be enabled, like branch delays (a characteristic of the MIPS ISA), a branch predictor, and bypassing values between stages, which reduces the execution cycles but decreases the frequency. The size of the caches, the size of the cache lines and the number of ways of the level 2 cache are configurable as well.

In section VI-C we discuss other FPGA-based softcores, and their suitability to our purposes.

We implemented the architecture with the commercial language Bluespec SystemVerilog (BSV) [5]. It was convenient for our purposes because its rule-based, data-flow nature gives the designer a tight control over the hardware model, and it allows extending the simulation with software as we will see in next section.

We also considered other alternatives, like C-based High-Level Synthesis (HLS) languages [6], [7], which provide very high productivity and fast adoption. But it is difficult to express the cycle model with these tools.

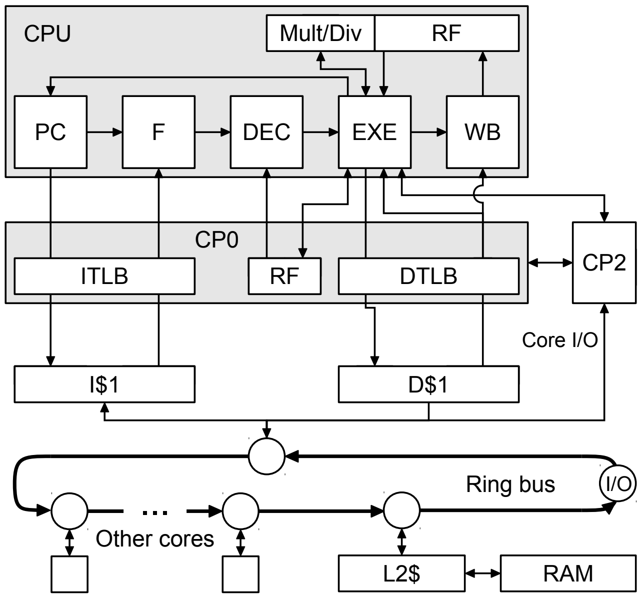


Figure 1: A Bluebox core with the CPU, instruction and data caches. In the MIPS I ISA, Coprocessor 0 manages the exceptions and virtual memory. We chose to dedicate Coprocessor 2 to timers, I/O and inter-processor messaging.

Our platform can be simulated in three modes, representing the three stages of the development process:

- `funcsim`: A functional and fast software simulator, for verification and validation.
- `timesim`: A detailed but slow simulation on host.
- `fpgasim`: A fast and detailed simulation on the FPGA.

In the next section we will detail the first two modes, consisting of software simulation on a host machine. FPGA-based simulation will be covered in section IV.

The `funcsim` model is a 1-stage CPU without memory caches. We used this functional simulator to validate the subsequent stages of the development. `funcsim` can be used for fast functional verification of new research hardware and software features.

III. Software Extensions For High-Level Debugging

Once the architecture was validated with `funcsim`, we implemented a BSV model of Bluebox. BSV can be mapped to Verilog and implemented in the FPGA. The BSV compiler is also able to generate a cycle-accurate C++ simulation model of the hardware. However, the BSV simulator does not allow to directly inspect the state of the hardware, like memories or registers. It supports `$display` commands like Verilog, which can print information during software simulations in a similar fashion as `printf` in C programs. This debugging level is not sufficient for complex hardware systems like multicore architectures.

For this reason, we decided to extend the BSV simulator with new features to make the development and research experience closer to the software design. Verilog and BSV support a programming interface to call software functions from within the simulation. We used this interface to extend the BSV simulator and produce `timesim`, a set of C++ extensions over the BSV simulator.

A. Externalizing The State

The state of the simulation at any moment is determined by several architectural elements. This elements are stored in BSV primitives, such as memories and registers, which cannot be queried by external software. We identified such elements and moved them to the C++ extensions of `timesim`. Then we replaced the BSV state primitives, such as registers and memories, by wrapper modules with compatible ports. Accessing these wrappers generates C++ calls to the external, software-managed state.

The elements managed by `timesim` are:

- The CPU register file.
- The MIPS multiply/divide unit registers.
- The MIPS Coprocessor 0 register file, used for virtual memory, exceptions and interrupts.
- The Translation Look-aside Buffer (TLB) entries.
- The main RAM.

We excluded other elements like the contents of the caches in this version of Bluebox.

B. Debugging And Profiling

In this section we will describe how we make use of these extensions. The external C++ calls from within the simulation have another side effect: the BSV simulator yields the execution to the external function, which pauses the execution because it is single-threaded. We used this effect to implement debugging functionalities to the BSV simulator.

The main debugging break point is in the execution stage of the CPU, where the changes of the current instruction are applied to the state. Before this happens, an external C++ function is called, which yields the control to the user through an interface. The user can inspect the contents of the state managed by `timesim`, and send commands like advance one instruction, run freely or stop on a given address. This kind of environment is familiar to programmers, and allows the designer to abstract from the low level details and focus on the architectural elements such as instructions and CPU registers.

Currently, the `funcsim` and `timesim` modes of Bluebox support several kinds of interfaces, like terminal, ncurses (graphical terminal), a Python shell, and a GDB server. The user interfaces show helpful information like the disassembly of the current instructions and the contents of the register file. The GDB interface allows the popular GDB debugger to connect to `funcsim` and `timesim` and control the execution of the hardware model as if it was a multithreaded application (each core appears to GDB as if it was a software thread). Some debuggers allow source-code-level debugging, abstracting from the machine code of the simulation. This feature is very helpful when debugging applications with complex macros and build systems like the Linux Kernel.

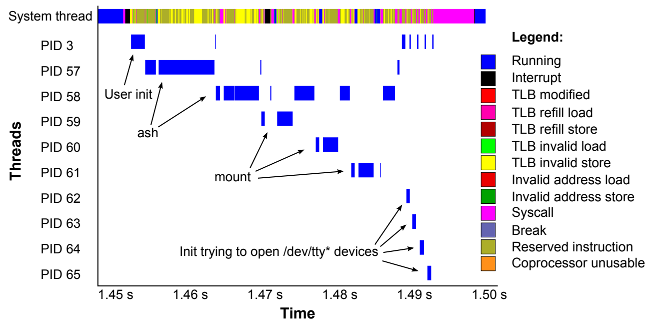


Figure 2: Execution trace using Paraver, displaying Linux threads and the execution mode of the CPU.

All the events collected by the extensions can also be saved into traces and visualized in a graphical way. `funcsim` and `timesim` allow to store special events and later visualize the trace using Paraver [8], a tool to graphically visualize supercomputing execution traces. Bluebox can generate different Paraver traces, including:

- The software threads created by Linux. This trace required software collaboration, which represented a single line of code in the Linux scheduler and an overhead of 1 cycle each time a software process is scheduled.
- The execution mode of a CPU over time, distinguishing between normal execution and special modes like interrupt serving or virtual memory management.
- The exact software function that was executed each cycle. This trace is very detailed, and helps the designer to understand the execution path of the hardware until an error appears.

In Figure 2 we show an example Paraver trace of a Bluebox core booting the Linux Kernel.

We also implemented some features which simplify the debugging of the multicore, such as the capacity of loading binary applications. Traditionally, hardware designers use Verilog commands to load memory contents from plain text files, which requires some simulation cycles and transforming the data into the desired hardware state. In some FPGA design environments the contents of the memories can be hardcoded in the description of the hardware, which requires recompiling the model each time the contents are modified. `funcsim` and `timesim` can preload the memories, like the RAM or the register file, at the beginning of the simulation from complex formats such as ELF files, the Linux executable format.

The ELF files may also contain debugging information, which helps funcsim and timesim to offer advanced information to the user. Examples of such information include the name of the current function being executed, or the name of the data variable being accessed by load and store instructions. Moreover, we implemented a call stack decoder that shows the source-code lines which lead to the current instruction.

C. Checkpointing

The detailed simulation with timesim is faster than the detailed simulation of Verilog code thanks to the rule-based nature of BSV. But being a cycle-accurate simulation of hardware which can be synthesized, can become impractical. timesim can execute up to 0.1 MIPS when simulating one core. The performance falls dramatically when the number of cores grows.

To avoid long waiting times, we implemented a checkpointing mechanism in timesim which allows to save the state of the simulation at any given point. This state is saved to a file, and can be loaded later and resume the simulation from that point with no functional differences. Moreover, the hardware model can be modified between simulation sessions. For instance, if a bug is detected in a late moment of the simulation, a checkpoint can be saved before arriving to the failing cycle, and different hardware models can be tested with the same initial state. Another possibility is avoiding tedious simulations, like booting the Linux Kernel, and resuming the simulation directly when applications can be executed.

This is possible because most of the state is managed by timesim. But during the execution of the hardware model there are intermediate stages where part of the state is in hardware units not managed by our extensions. For instance, the instructions in the pipeline and the dirty data lines in the data caches, which have been modified but not committed to memory yet. For this reason, the simulation must be driven to a new stable epoch where all the state in intermediate hardware has been flushed to units managed by timesim.

For this purpose, timesim can instruct the CPU pipeline to finish the execution of the current instructions and stop issuing more requests. After doing so, the CPU starts sending special commands to the L1 and L2 data caches to flush any dirty data lines to the main RAM. This behavior required small hardware extensions to the CPU and the caches, which are not present in fpgasim in this version.

After this preparation, the state managed by timesim is coherent with the state of the simulation, and it is saved into a file. Later on, the user can load this checkpoint and the state is preloaded into the hardware elements, which allows resuming the simulation from the previous functional point. Flushing the pipeline and the data caches when saving the simulation, and refilling when resuming it, requires some simulation cycles because this mechanisms are implemented in hardware. This timing differences are minimal, but perfect checkpointing would require timesim to manage all the state of the simulation.

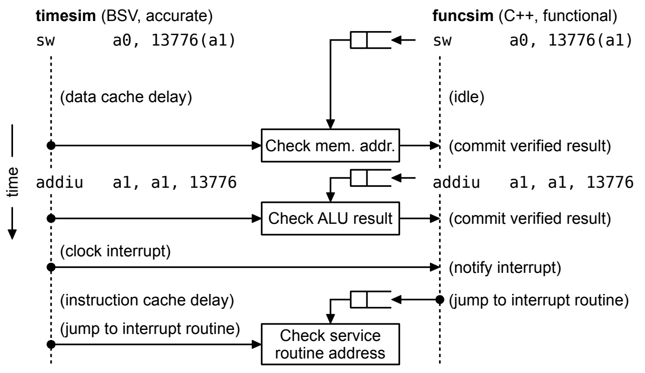


Figure 3: Verification of timesim using funcsim. Functional results are buffered and checked when timing-aware results are available.

To optimize the size of the checkpoint file, the timesim RAM manager keeps a registry of modified 4-KB pages. When saving the memory, only such pages are stored. When loading a checkpoint, only the pages in the checkpoint file are loaded to the memory.

The hardware mechanisms that allow flushing the buffers of the system could also be synthesized as hardware in the FPGA, but in this version of Bluebox this is not possible yet, especially because saving the contents of the DDR RAM can be challenging.

D. Verification

Errors in the software or the hardware may not exhibit symptoms immediately. For example, the result of a buggy arithmetic operation in the CPU may be sent to the memory. After a few million cycles, when this result is retrieved and used to calculate an instruction address, it will cause a TLB exception. Debugging the origin of this error can be slow and tedious, especially in a multicore running an operating system.

To implement fine-grain, instruction-level verification, we integrated funcsim into timesim as a functional verifier. In this simulation mode, funcsim and timesim run in parallel and the result of each instruction is compared. The much lower simulation cost of funcsim only represented a modest time overhead for timesim.

Both simulation modes run independently, but funcsim does not have notion of time and must be synchronized with the timing model of timesim. For that purpose, we implemented the elastic verification mechanism shown in Figure 3, where the immediate results of funcsim are queued, and checked when the timing-aware results of timesim are generated. If both versions match, the result is committed to the shared state managed by timesim.

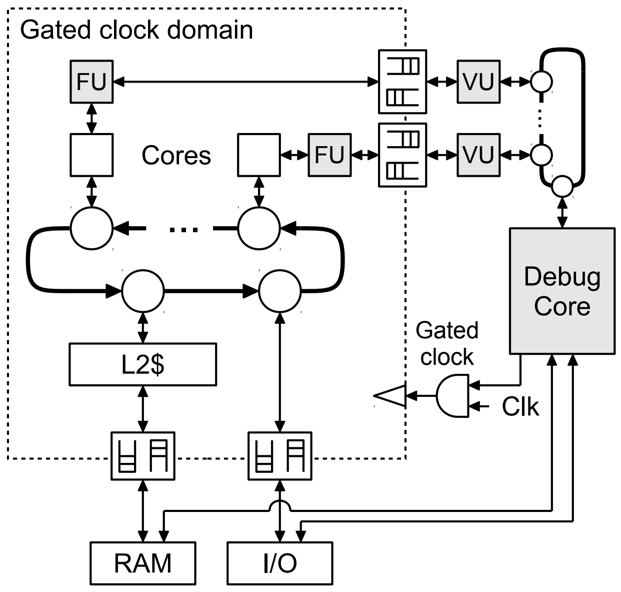


Figure 4: Debugging and verification infrastructure of `fpgasim`, with the Debug Core, Filter Units (FU) and Verification Units (VU).

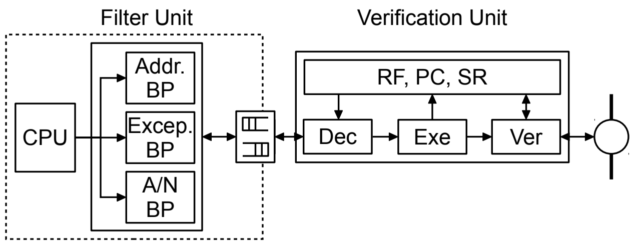


Figure 5: Detail of the Filter Unit (FU) and Verification Unit (VU). In debugging mode, the FU filters instructions that do not match current breakpoint type, and the VU bypasses all the information. In verification mode, the FU is disabled (*filter none*) and the VU decodes, executes and verifies simple instructions in one cycle.

IV. FPGA-based Debugging And Verification On-Chip

The BSV hardware model used in `timesim` can be synthesized into Verilog and mapped to a FPGA. We call this simulation mode `fpgasim`. However, in this mode the software extensions of `timesim` are not available, and any debugging techniques must be implemented as part of the hardware model.

In `fpgasim` the cores generate a packet of data for each instruction, containing basic debugging and verification information, such as the address of the instruction or the arithmetic results if any. **The system can process this information in two modes: debugging and verification.** In the former, the multicore can be cycle-stepped, and breakpoints can be set thanks to the Filter Unit (FU). In the latter, all the instructions executed by the all the cores are verified by the Verification Unit (VU). A Debug Core manages this infrastructure and consists in a simplified Bluebox core, where we disabled features that were not necessary such as the virtual memory. This minimal functional subset of the core is the basic unit of trust in our system, and its correctness was verified with `funcsim`. In Figure 4 we show the architecture of the debugging and verification system.

The whole multicore is isolated in a gated clock domain. If we only paused the CPU, but not the whole system, the locking of the pipeline would interfere with the natural delays of the microarchitecture, which would appear as a zero-delay operation. The interfaces with external units, such as the DDR controller or I/O, are protected by synchronization queues.

The gated clock domain protects the internal consistency of the multicore, but the delays originated outside cannot be preserved. For instance, pausing the system may hide the latency of the DDR memory. **To avoid external delay simulation loss of accuracy, we enriched the cross-domain queues with latency feedback.** This mechanism is aware of the delay of each request and response. If the execution is paused and some delay is missed, it artificially holds the data to emulate the missed cycles, rebalancing the simulation.

A. The Debugging Mode

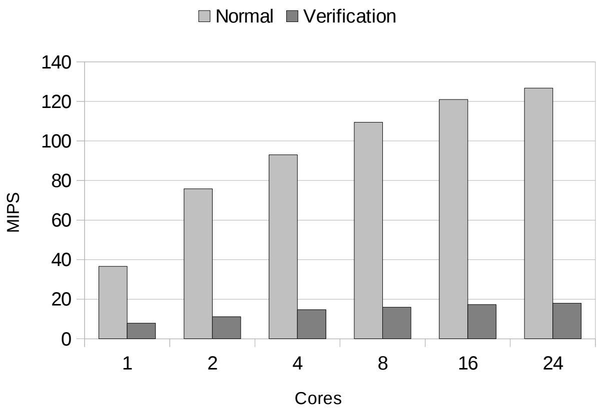
In this mode, the Debug Core runs a debugging software and can send breakpoint commands to the FU located next to each core, inside the gated clock domain. There are three types of breakpoints:

- Address breakpoints: The information generated by the CPUs is dropped, letting the cores run freely until the address matches. The information is sent to the Debug Core, which pauses the simulation and notifies the user.
- Exception breakpoints: The same behavior as address breakpoints, but the condition is a given exception code (e.g., interrupt, TLB, etc.).
- All/None breakpoints: Either all the information is dropped, letting the core run freely, or all the information is submitted to the Debug Core, which allows cycle-stepping.

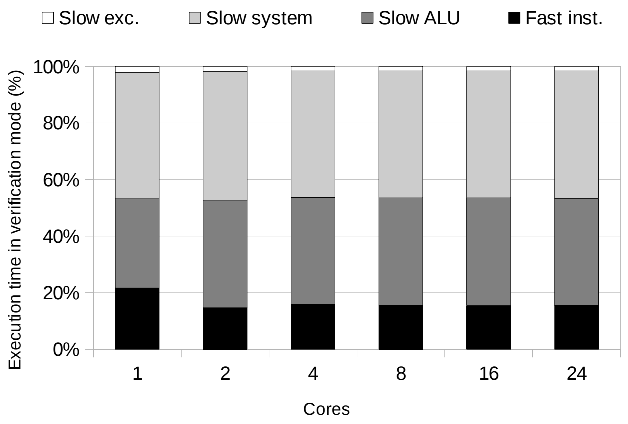
In Figure 5 we show the details of the FU. In debugging mode, the VU just bypasses the data not filtered by the FU.

B. The Verification Mode

In verification mode, the Debug Core runs the `funcsim` software model and the FUs are in *filter none* mode, bypassing all the information. Outside the gated clock domain we placed the hardware VUs, which will verify in hardware simple instructions, like arithmetic operations or branches.



(a) Performance of FPGA-based simulation in normal mode and in verification mode.



(b) Breakdown of time spend on each type of verified instructions. *Fast* instructions are verified in hardware, *slow* in software.

Figure 6: Performance of `fpgasim` running the Dhrystone benchmark and breakdown of verified instructions.

These *fast* instructions represent 99.5% of the execution and are checked in one cycle, while the *slow* instructions, mainly related to exceptions and virtual memory, are verified by software in the Debug Core.

Each VU has an instruction decoder and execution unit. A minimal subset of the state, consisting in the register file, the program counter and the state register is present in the units. The verification proceeds as follows:

1) If a fast instruction is received, it is verified in the VU. If correct, the local state is updated and the information is dropped.
2) If a slow instruction is received or the verification fails, the VU sends the information to the Debug Core and the simulation is stopped.
3) The software in the Debug Core will exchange messages with the VU to obtain the last valid state from their local subset.
4) The instruction is verified by software, and the updated state is sent back to the originating VU.
5) Finally, a continuation message is sent to the VUs and the simulation is resumed.

We included the Coprocessor 0's Status Register (which controls virtual memory and exceptions) in the VU because it is frequently read. This reduced the slow instructions by 30%. Multiplications and divisions are also performed by software. The designer could extend the verification to memory caches, for instance tracking requests and responses.

The Debug Core, when verifying, must have a copy of the state in case an exception is detected. Making such copies requires many cycles. We implemented a simple Transactional Memory [9] scheme to optimize the execution, based on a small journal FIFO, which can undo up to 16 recent writes to the data cache.

V. Experimental Evaluation

In this section we will evaluate the performance of the FPGA-based simulator, `fpgasim`. We mapped from 1 to 24 cores to a Xilinx VC709 board, running at a maximum speed of 80 MHz. The 24-core version consumes 77.75% of the logic elements, each core requiring a 2.18% (7345 LUTs, including the CPU and the caches). The verification units represent a total area overhead of 9.80%.

We run the synthetic, integer-only Dhrystone benchmark (one copy in each core) to measure the performance of the multicore and the verification system. Running a floating-point benchmark would force Linux to emulate the FPU through exceptions, which would represent an abnormally-high number of system instructions to verify.

In Figure 6a we observe that the multicore reaches a peak performance of 120 MIPS with 24 cores. This data shows that the simulated architecture has performance bottlenecks in the CPU and the memory hierarchy. Being an FPGA-based simulator, it should be easy to implement dedicated analysis hardware, such as performance counters, and investigate more advanced interconnection topologies.

The verification units scale at the same rate as the multicore, reaching a maximum performance of 17 MIPS for 24 cores. The slowdown compared to the non-verified version ranges from 4x to 7x. In Figure 6b we show a breakdown of the simulation time in verification mode. The verification of ALU instructions represents a 38% of the simulation time. These instructions are mainly multiply and divide operations, which Dhrystone uses intensively. We did not considerer it useful to verify hardware divisions with a similar hardware divider. The system instructions, mainly virtual memory operations, represent the 0.27% of the instructions and 44% of the verification time.

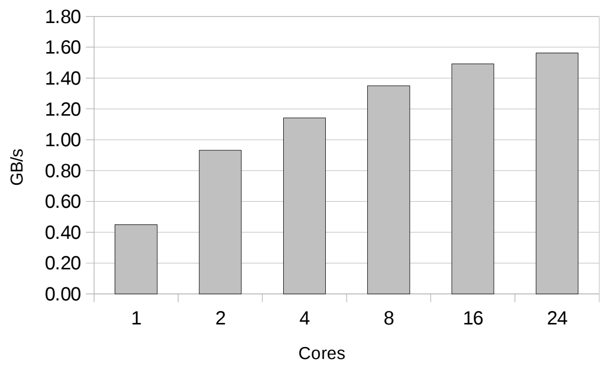


Figure 7: Verification data processed by `fpgasim` (GB/s).

A. *The Suitability Of On-Chip Verification*

The instructions that cannot be verified by hardware represent 0.5% of the instructions and require 78% of the execution time. Each slow instruction can take between 600 and 1500 cycles of multicore time. Not verifying the system instructions and the exception events would reduce the verification overhead by a 56%, speeding up the verification performance by 2.3x. But we considered important to develop a full-system simulator, capable of verifying operating system events.

Choosing between a self-contained, versatile solution like the on-chip Debug Core or other alternatives is an interesting research problem. For example, if an I/O interface is not required by the simulated system, the software verification can be offloaded to a host. In Figure 7 we can observe that the amount of verification data generated by the CPUs, which reaches a maximum peak of 1.56 GB/s. If the VC709 PCI Express interface was used, and with a measured, round-trip transfer delay of 25 μs, sending an instruction to a host for verification is equivalent to 2000 Debug Core cycles at 80 MHz (without including the software verification time on the host).

Thus, off-chip, host-assisted verification is not worthwhile if advanced techniques are not implemented. For instance, verification instructions should be aggregated to compensate the latency, and the verification should be performed in blocks. The only problem with this approach is that slow instructions are not always consecutive, which makes difficult packing them. On the other hand, on-chip solutions have very high bandwidth and low latency.

Using embedded hard cores would not require such complex modifications. For instance, the Xilinx Zynq 7000 FPGA has a measured latency from CPU to BlockRAM between 40 and 74 ns [10], and its embedded ARM dual core runs at 720 MHz. In this platform between 3 and 6 Bluebox cycles are needed to obtain a verification request. However, FPGAs with embedded cores have less logic resources, and the simulated multicore should be smaller.

B. *Use Cases*

We want to illustrate the usefulness of the methods previously described with our own experience during the development of Bluebox. In particular, we detected some timing errors in a functionally correct hardware. One of those errors was related to the MIPS Load-Linked (LL) and Store Conditional (SC) instructions, which allow atomic writes to the memory and are essential for multicores. Under certain conditions, the LL address could be mapped to the same L2 cache line as the SC instruction, which caused an always-failing atomic update. We fixed that problem with independent cache sets in the L2.

Another problem was related to the CPU's pipeline. If an instruction fetch entered the pipeline shortly before updating a TLB entry, it would miss the new virtual address and potentially generate a false-positive TLB error. Such problem required extra control logic to avoid instruction memory requests during TLB updates. Those two cases exemplify the necessity of a cycle-accurate verification system that does not interfere with the simulation behavior.

VI. Related Work

A. *Verification On-Chip*

DIVA [11] introduced dynamic verification, augmenting an out-of-order CPU with a small in-order verification CPU. Results were checked at commit time. This technique was targeted to silicon prototype verification. Argus [12] has microarchitectural verification units. Both are targeted to functional verification, and cycle-accuracy is not strictly preserved as in Bluebox.

B. *Software Simulators*

We will discuss four of the most popular and recent proposals of software multicore simulators. gem5 [1] and MARSS [2] are full-system multicore simulators that use statistical models to speedup the execution, reaching speeds of up to 0.8 MIPS for 8 cores. Sniper [3] and ZSim [4] perform interval simulation but they do not model the full system and running multiple applications can be complicated. Sniper can simulate 16 cores at 2 MIPS, with the average errors of simulation within 25%. ZSim can simulate up to 314 MIPS when simulating a 1000-core system. None of them is cycle-accurate.

There are three negative consequences when using software simulators that are not cycle-accurate:

1) Accuracy is traded off for performance, introducing subtle differences in the simulation.
2) This performance gain is based on replacing cycle-level simulation with architectural-level statistical models, which exhibit similar architectural events, but have a different cycle-level behavior.
3) The correctness of the statistical models is matched against real hardware. However, there is no reason

for those models to continue being valid when new features are added to the system [13], which is the purpose of computer architecture research.

We believe that 1) and 3) question the suitability of fast, non-cycle-accurate software tools for new architectural research proposals. Such models are accurate only when simulating existing hardware. In addition, when verifying the correctness of the new microarchitectural modifications, 2) may hide software and hardware errors that only appear under certain timing conditions, as we show in section V-B.

Bluebox can perform full-system, cycle-accurate simulations, and accuracy is preserved with new changes.

Regarding functional verification of multicore simulators, Tomić et al. [14] followed the same methodology as we did to verify `timesim` with `funcsim`. In their case, they implemented fast, functional modules to verify architectural elements in software simulators, such as hash maps to test data caches.

C. FPGA-Based Multicores And Simulators

Existing FPGA soft-cores did not match our requirements. OpenRISC [15] is a popular open-source microarchitecture, but it is written in Verilog and supports a custom, non-standard ISA. The commercial soft-cores Microblaze (Xilinx) and Nios (Altera) provide high performance and support the Linux Kernel, but they are closed-source and the designer cannot experiment with its microarchitectural characteristics.

Tan et al. [16] provide an excellent review of FPGA Architecture Model Execution (FAME) versus Software Architecture Model Execution (SAME). From their work we conclude that many FAME tools follow the same strategy as the SAME tools, allowing non-cycle-accurate timing models for the sake of performance (which can be several orders of magnitude higher than software tools). This causes several problems, as we previously discussed. That is the case of RAMP Gold [17] and FAST [18]. In contrast, HaSim [19] and Arete [13] use *A-Ports* or *Latency-Insensitive Bounded Data-flow Networks*, which decouple FPGA cycles from model cycles while guaranteeing cycle-accuracy. HaSim uses time multiplexing, which does not scale. Arete can model up to 8 PowerPC cores with 4 FPGAs at 55 MIPS. While these large cores do not allow large multicore simulations, their mehtodology is a good compromise between performance and accuracy.

VII. Conclusions

In this paper we presented debugging and verification techniques for host-based and FPGA-based multicore simulation. We showed how to implement unobtrusive full-system debugging and verification on the FPGA. As a proof of concept, we implemented those techniques in a reference architecture, Bluebox. Bluebox simulates up to 24 cores with cycle-accuracy at 120 MIPS, and perform full-system verification at 17 MIPS.

References

[1] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[2] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *DAC*, 2011.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*. ACM, 2011.

[4] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," in *ISCA*, 2013.

[5] (2015, Jan.) Bluespec systemverilog. [Online]. Available: http://bluespec.com

[6] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[7] A. Canis *et al.*, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *FPGA*. ACM, 2011.

[8] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *WoTUG-18*, vol. 44, 1995.

[9] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," in *ISCA*, 1993.

[10] (2015, Jan.) CPU latency to access an AXI Slave using Master AXI GP. Xilinx Inc. [Online]. Available: http://www.xilinx.com/support/answers/47266.html

[11] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *MICRO-32*. IEEE, 1999.

[12] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *MICRO-40*. IEEE, 2007.

[13] A. Khan *et al.*, "Fast and cycle-accurate modeling of a multicore processor," in *ISPASS*, 2012.

[14] S. Tomić, A. Cristal, O. Unsal, and M. Valero, "Using Dynamic Runtime Testing for Rapid Development of Architectural Simulators," *IJPP*, 2012.

[15] (2015, Jan.) Openrisc open-source architecture. [Online]. Available: http://opencores.org/or1k

[16] Z. Tan *et al.*, "A case for fame: Fpga architecture model execution," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[17] Tan, Zhangxi and others, "RAMP gold: an FPGA-based architecture simulator for multiprocessors," in *DAC*, 2010.

[18] D. Chiou *et al.*, "FPGA-Accelerated Simulation Technologies (FAST): Fast, full-system, cycle-accurate simulators," in *MICRO-40*, 2007.

[19] M. Pellauer *et al.*, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *HPCA*, 2011.