A Case of System-level Hardware/Software Co-design and Co-verification of a Commodity Multi-Processor System with Custom Hardware

Sungpack Hong*
Oracle Labs
sungpack.hong@oracle.com

Tayo Oguntebi
Stanford University
tayo@stanford.edu

Jared Casper
Stanford University
jaredc@stanford.edu

Nathan Bronson*
Facebook, Inc.
nbronson@stanford.edu

Christos Kozyrakis
Stanford University
kozyraki@stanford.edu

Kunle Olukotun
Stanford University
kunle@stanford.edu

ABSTRACT

This paper presents an interesting system-level co-design and co-verification case study for a non-trivial design where multiple high-performing x86 processors and custom hardware were connected through a coherent interconnection fabric. In functional verification of such a system, we used a processor bus functional model (BFM) to combine native software execution with a cycle-accurate interconnect simulator and an HDL simulator. However, we found that significant extensions need to be made to the conventional BFM methodology in order to capture various data-race cases in simulation, which eventually happen in modern multi-processor systems. Especially essential were faithful implementations of the memory consistency model and cache coherence protocol, as well as timing randomization. We demonstrate how such a co-simulation environment can be constructed from existing tools and software. Lessons from our study can similarly be applied to design and verification of other tightly-coupled systems.

Categories and Subject Descriptors

B.4.4 [**Performance Analysis and Design Aids**]: Simulation, Verification

Keywords

Co-Verification, Co-Simulation, Bus Functional Model, FPGA Prototyping, Transactional Memory

1. INTRODUCTION

Modern digital systems are moving increasingly towards heterogeneity. Today, many digital systems feature multiple (heterogeneous) processors, advanced interconnect topologies beyond simple buses, and often specialized hardware to improve the performance-per-watt for specific tasks [5].

Development of such heterogeneous systems, however, requires intensive system-level verification. System-wide co-design and co-verification of hardware and software components [16] is especially essential for systems where multiple heterogeneous components are executing a single task in a tightly-coupled manner, orchestrated by the software. In such systems, for example, data races between multiple computational components should be thoroughly verified since they can induce various unexpected system behaviors.

In this paper, we present our experiences designing and verifying a tightly-coupled heterogeneous system in which multiple x86 processors are accompanied by specialized hardware that accelerates software transactional memory (Section 2.1). We discuss which among the many conventional co-design/verification methods would best serve our purposes and why. Our chosen method was one which combined an un-timed software model with cycle-accurate interconnection and HDL simulation through a processor bus functional model (BFM), since this provided sufficient visibility and simulation speed (Section 2.2). However, we found that *when using a BFM for verification, conventional methodology needs to be significantly extended* in order to capture data-race corner cases induced by modern multi-processor architectures. Especially crucial were correct implementations of memory consistency and cache coherence as well as a need to introduce timing randomization (Section 3.1). This paper also demonstrates how such a co-simulation environment can be constructed on top of existing tools and software (Section 3.2). We show the effectiveness of our methodology and draw out general lessons from it (Section 4), before we conclude in Section 5.

Our contributions can be summarized as follows:

- We present a non-trivial system-level co-design and co-verification experience with x86 processors connected to custom hardware. In this scenario, we found that combining the software model with an interconnection simulator and HDL simulator via a processor BFM is the most effective method for functional verification.
- We also explain, however, that conventional ways of constructing a processor BFM should be revised in accordance with modern multi-core processors and inter-

*This work was done when the authors were at Stanford

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland.

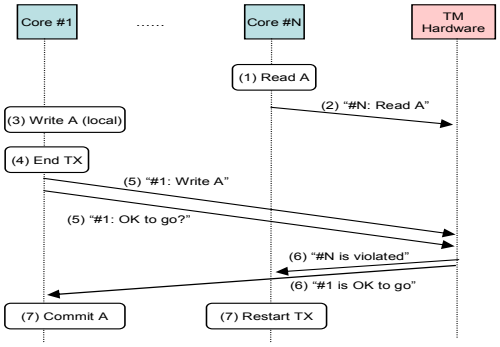

Figure 1: Outline of the STM acceleration protocol.

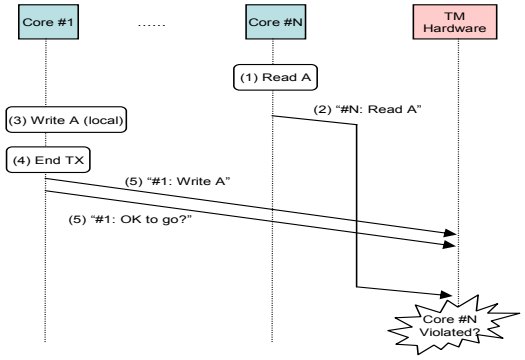

Figure 2: A failure case in our first design.

connection architecture; it is especially important to accurately implement the memory consistency model and cache coherence protocol. Introducing timing randomization is also very important.

2. DESIGN AND VERIFICATION

2.1 Target Design

This section outlines the design of our system for the sake of providing sufficient background context for one to understand our co-design and co-verification issues, while the detailed design of the system is outside scope of this paper.

Transactional Memory (TM) [9] is an abstract programming model that aims to greatly simplify parallel programming. In this model, the programmer declares atomicity boundaries (transactions), while the runtime system ensures a consistent ordering among concurrent reads and writes to the shared memory. However, a typical software transactional memory (STM), an implementation of such a runtime system solely with software, tends to exhibit huge performance overhead.

Our system is composed of specialized hardware which is externally connected to commodity CPUs and accelerates an STM system [2]. The motivation was reduction of the performance overhead generally experienced by STM systems without modifying the core and thus increasing processor design complexity. The TM abstraction is enforced through the following protocol, which is outlined in Figure 1:

(1) Whenever a core reads a shared variable, (2) the core fires a notification message to the TM hardware. (3) Writes to shared variables are kept in a software buffer, and are not visible to other cores until (4) execution reaches the end of a transaction. At this point, the core (5) sends the addresses of all the writes that it wants to commit to the hardware and waits for a response. (6) The TM hardware, based on all the read/write address received from all the cores, now determines which cores(transactions) have conflicting read and writes and sends out the requisite messages to those cores. (7) Depending on these messages, each core proceeds to commit or restart its current transaction from the beginning.

The above protocol is implemented as a tightly-coupled software and hardware system. Software sends messages and manages the local write buffer, while the external hardware accelerator performs fast conflict detection. For our development environment, we used an instance of an FPGA-based

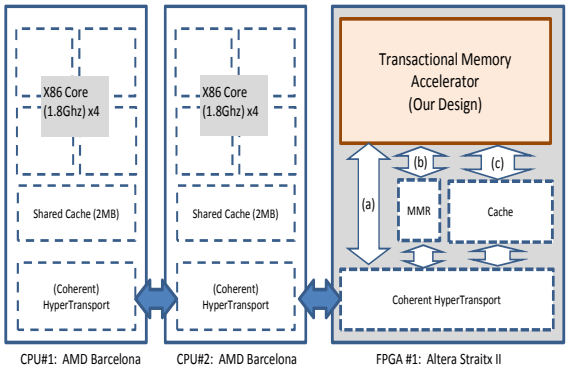

Figure 3: Block Diagram of Implementation Environment: Our design is connected to the rest of the system via (a) non-coherent interface, (b) memory-mapped register interface, and (c) coherent cache interface.

rapid prototyping system [14]. Figure 3 illustrates the block diagram of the prototyping system. The system is composed of two quad-core x86 CPUs and an FPGA that is connected coherently via a chain of point-to-point links. We implemented our custom hardware (TM Accelerator) on the FPGA of the system (Figure 3). Messages from the CPU are sent to our HW via a non-coherent interface, while responses to the CPU go through the coherent cache. Note that the former non-coherent mechanism enables hiding communication latency between the CPU and FPGA while the latter coherent communication avoids interrupts and long-latency polling.

2.2 Our Initial Failure and Issues with Co-Verification

Since our system was composed of tightly coupled hardware and software, we had to deal with a classic chicken-and-egg co-verification problem: How can we verify the correctness of the hardware without the correctly-working software and vice versa? Our initial approach was to decouple hardware and software verification; we verified the software with an instruction set simulator (ISS) and virtual hardware model while targeting the hardware using unit-level HDL simulation and direct debugging on the FPGA.

It was only after the simulation in our ISS environment

Method	Strength	Weakness
(A) Prototyping	Fastest execution on real HW	Limited visibility regarding hardware status
(B) Full HW simulation	Full visibility and control	Extremely low simulation speed
(C) ISS + HW simulation [15, 12]	Faster simulation than (B)	Waste of simulation cycles for unrelated instructions
(D) ISS + Virtual HW model [7, 18]	Faster simulation than (C); Can predate HDL development	Same issues as (C); Fidelity of virtual HW model; Requires a separate HW verification step
(E) SW model + HW sim. [20, 17, 21] (possibly through BFM)	Verification of target HDL with real SW execution	Overhead of SW re-writing; Lack of timing information for SW execution
(F) Emulation [13, 1]	Fast execution of target HDL	Cost of the systems; Limited support of core types
(G) Binary Translation [10]	Fast SW execution on host HW	Lack of deterministic replay of concurrent execution.

Table 1: Comparisons of HW-SW co-verification methods.

and unit tests in FPGA environment had all passed successfully, but the whole system was tested altogether for the first time, when we realized that there was a flaw in our original design. The specific error scenario is shown in Figure 2. The figure depicts the same read-write sequence as in Figure 1 except that in this case, the delivery of the read message from core N is delayed until after delivery of the commit message from core 1. As there was no consideration of such case in our original design, the software execution was simply crashed after a failure of consistency enforcement. Note that since the ISS assumed in-order instruction execution and the HW model assumed in-order packet delivery, our environment was not able to generate the scenario described in Figure 2. As it happened, this scenario occurred quite frequently in real HW.[1]

That being said, we had spent a significant amount of engineering efforts to confirm that Figure 2 is really the reason of the observed failure. Single-threaded SW executions never failed, while multi-threaded ones crashed occasionally but *non-deterministically*. A logic analyzer, tapped onto our FPGA, was not very helpful because a typical execution involves billions of memory accesses, each access generating tens of memory packets – it was extremely hard to identify which of those packets were relevant to the error and how those are related to SW execution. For example, a crash observed after one last memory access from one core (e.g. de-referencing a dangling pointer), could be a result of a write, falsely allowed to commit, from another core millions of cycles before. Simple step-by-step execution of a single processor was not helpful either due to our parallel execution requirement. As a matter of fact, we identified this issue through a deep speculation on our initial design, which was confirmed later only after adapting another co-design/co-verification methodology (Section 3).

Fixing up this issue, we properly augmented the protocol in Figure 1 and re-designed our hardware and software accordingly. This time, however, in order to save ourselves from repeating the same mistake, we considered applying other methodologies and tools proposed for HW/SW co-design and co-verification [16, 1, 18, 12, 20, 15, 7, 17, 13] Our requirements could be summarized as follows:

- The new STM software needed to be intensely validated (with the new hardware), especially under the assumption of parallel execution, variable latency and out-of-order message delivery as shown in Figure 2. Such an intensive SW validation naturally demanded fast execution.

- We were more interested in functional verification of the new hardware rather than architecture exploration. Furthermore, as our new hardware was becoming available soon (after small modification from the initial version), we wanted direct verification of target RTL as well.

- We wanted a mechanism for error analysis better than manual inspection of GBs of logs/waveforms from simulation or logic analyzer. At a minimum, we wanted to associate such logs with software execution context.

Table 1 summarizes the advantages and disadvantages of conventional approaches that we considered. Our initial approach used prototyping (Method A in Table 1) and ISS combined with virtual models (Method D), but as described previously failed to serve our purpose. Full simulation (Method B) was never an option since we didn't have access to the HDL source of the CPUs and it would have surely failed to meet the fast execution requirement. We have already described how ISS (Method C) initially failed to find the erroneous case in the first place. An alternative was to use a detailed architecture simulator (e.g. [22]) but its simulation speed was insufficient. Also, there were no emulators (Method F) available to us which supported our core (AMD x86) and interconnection type (HyperTransport). Finally, we were not able to use fast software simulation techniques based on binary instrumentation (Method G), because it is not trivial with this method, to replay every load/store instruction of every processors in exact same order. Note that such a feature is crucial for verifying multi-processor systems like ours by nature.

The only remaining option was to construct a model that faithfully reflected the behavior of the software while interfacing with the HDL simulation (Method E). However, this method brought its own challenges which we discuss in detail in the following section.

3. APPLYING SW MODEL-BASED CO-SIMULATION

3.1 General Issues and Solutions

In this section, we discuss general issues that arise when using a software model for HW/SW co-verification and how we overcame those issues. For each issue, we either introduce lessons learned from previous research or discuss how they were not applicable in our case.

[1]There are two major reasons for this. (1) The underlying network enforces no delivery ordering between different cores. (2) We used a non-temporal x86 store instruction to implement asynchronous message transmission; however the semantics of the instruction allowed the message to stay in the store-buffer for an indefinite amount of time.

(Issue #1) Software has to be re-written into a form that can interface with HDL simulation.

The first issue can be minimized by using the BFM of the processor [17] (also known as processor abstraction with transaction-level (TL) interface) which works as a bridge between software execution and hardware simulation. Specifically, instead of creating a separate software model, the whole software is executed natively; however every *read* and *write* instruction that may potentially affect the target hardware is replaced with a `simul_read()` and `simul_write()` function call. These function calls invoke data transfers in simulation, suspending the software context until the underlying hardware simulation finishes processing the requested data transfer (in a cycle-accurate way). In the worst case, every load and store should be replaced; in most cases, it is enough to change only the Hardware Abstraction Layer (HAL), a small well-encapsulated portion of software [21, 6].

Hardware simulation can consist of two different components: a cycle-accurate interconnection network simulation and HDL simulation. The interconnection simulator can easily interact with HDL through simple function-to-signal translators [3] as long as the simulation is packet-wise accurate. This decoupled approach provides more flexibility and better simulation speed than doing whole HDL simulation [19].

In our case, only a small portion of code inside the STM was identified as HAL – the user application was entirely built upon the STM interface. For the interconnect, we used a cycle-accurate HyperTransport simulator, developed by AMD, which can interact with the HDL simulator (e.g. ModelSim) via PLI.

(Issue #2) The multi-threaded software should be executed concurrently, but also be re-playable in a determistic manner.

The second issue arises because we are combining execution of multi-threaded software with a simulator which is executed sequentially by nature. Note that we cannot let each thread in the original application freely interact with the simulator for the purpose of deterministic re-execution, i.e. simulators based on binary instrumentation such as Pin [10] cannot be used. Instead, we rely on the single-threaded simulator to interleave multiple software execution contexts.

SystemC [8] is a standard simulation engine that can handle multiple execution contexts for such use cases [21, 17]; SystemC allows for trivial implementation of blocking calls. Unfortunately, there are still plenty of (in-house) simulators which do not adhere to the SystemC standard, including popular CPU simulators [22] and interconnection simulators [4]. Our interconnection simulator[2] was not compliant with SystemC, either. We therefore implemented our own blocking-call mechanism using co-routines, or fibers.

(Issue #3) Software models lose timing information.

Being natively executed, software models lose timing information – they simply continuously inject read/write requests to the BFM simulator. The conventional solution is to insert `delay` calls explicitly before each call-site of `simul_read` and `simul_write` in the user application, which compensates for the CPU cycles between the previous blocking call and the current one [21, 6].

[2]The simulator is a proprietary implementation by AMD and requires an NDA to access.

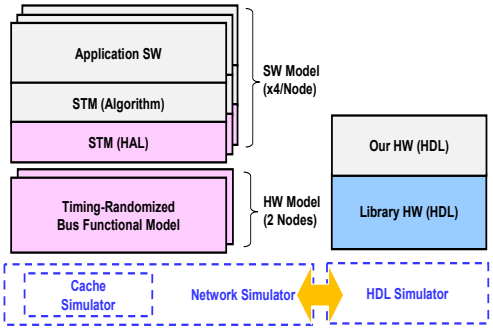

Figure 4: Block diagram of our co-simulation environment.

Our approach differs from the conventional solution in two ways: (1) we insert the delay inside the `simul_read` method rather than at the call-sites in the user-application code, and (2) we (pseudo-)randomize the delay values. We justify this for the following reasons: First, our method requires no further modification of the user application code. Second, user-provided delay information at the call-sites is already inaccurate. For instance, CPU cycles between two call-sites cannot be accurately compensated if there are (exponentially) many execution paths between them. Finally, for the purpose of functional verification, the exact number of execution cycles for non-relevant SW sections is of little interest. Rather, for functional verification, we want to interleave packet injection from multiple cores in varying orders as much as possible.

(Issue #4) The memory consistency model must be carefully considered.

This is one issue that has *not* been discussed extensively in the literature. In previous studies [17, 21, 6], the BFM simply injected network packets to the interconnection network in the order requested by the software as is the behavior in classic embedded processors. However, this does not closely approximate the memory packet generation pattern of our modern x86 processor; it fails to account for the aggressive reordering of memory requests.

Instead, we implement a more realistic memory consistency model in our processor BFM, namely Total Store Ordering (TSO) [11]. This is the model on which many modern processors (e.g. x86 and SPARC) are based. To ensure TSO, we keep a per-core store buffer inside our BFM. The write request goes to the buffer without injecting a packet into the network as long as there is an available slot in the buffer. On a read request, we first search for the target address in the store buffer. If not found, a new (read-request) packet is injected into the network. Otherwise, the contents in the store buffer, up to the entry that has been matched, are flushed before the new packet is injected. The store buffer is always flushed in FIFO order.

In addition, cache coherency should be implemented correctly as well, simply for the sake of correct parallel execution. However, we were able to leverage the cache simulator already embedded in our network simulator.

(Issue #5) There should be an easy way of error analysis.

In previous section, we explained how painful and unsuc-

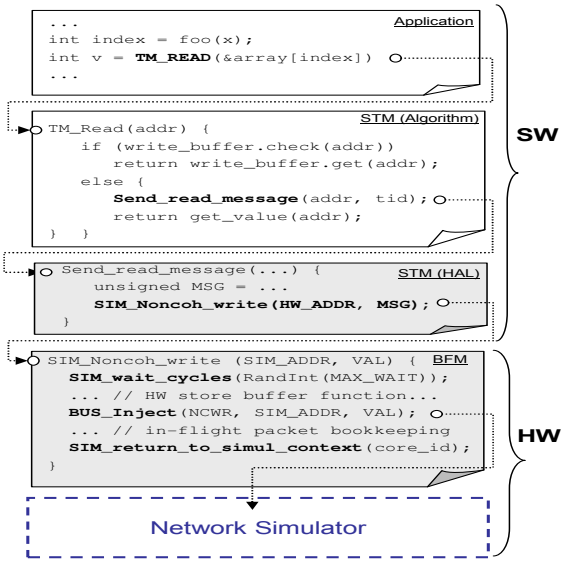

Figure 5: Execution sequence: From SW-context to simulator context.

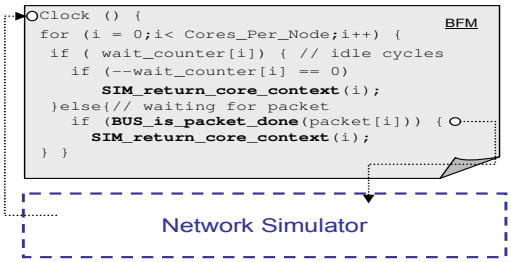

Figure 6: Execution sequence: From simulator context to SW-context.

cessful it was manually inspecting massive amounts of logs that are blindly generated by logic analyzer (or simulator), when identifying breach of consistency protocol.

To the contrary, our BFM-based approach enabled a better scheme for an off-line analysis; we exploit the facts that (1) the multi-processor simulation is actually being executed single-threaded on a workstation, that (2) the simulated address space is separated from simulator's address space, and that (3) simulated execution context and native execution context are also clearly divided. In specific, we further instrumented the STM (i.e. our HAL) such that we add a log entry in a global shadow data structure whenever there is a relevant activity from the current core, such as transactional memory access or commit request. Whenever a new entry is appended to the shadow data structure, a global check on consistency protocol is performed as well – for instance, if the log indicates that the current transaction has a conflicting memory access with another transaction but both are allowed to commit, the simulation immediately reports an error with accurate conflict information. Note that the global shadow data structure is kept inside simulator's context, and therefore such a error check is thread-safe and does not consume any simulation cycle.

3.2 Our Co-simulation Environment: Implementation

This subsection details the implementation of our co-simulation environment where all the issues discussed in previous subsections are resolved. Figure 4 depicts the block diagram of our co-simulation environment. Our HDL design is simulated alongside the library HDL of our FPGA framework, whose interconnect pin-outs are connected to the network simulator through the PLI mechanism. Our BFM implementation is treated as a traffic generator by the interconnection network simulator, which is dynamically linked at runtime. The BFM module, network simulator and HDL simulator represent the hardware part of the system in this simulation environment. The software part is the whole application and STM software that are unmodified except HAL, which is now built upon BFM API (Table 2). Each application thread is implemented as a fiber (co-routines) whose context switching is managed by BFM. Specifically, we used the POSIX `makecontext(3)` and `swapcontext(3)` mechanisms for fiber implementation. We instantiate two BFM modules (CPU nodes) on the simulator with each BFM module executing four SW threads (CPU cores) at a time, which faithfully models our system configuration (see Figure 3).

Table 2 summarizes the API of our processor BFM, which is called by the software model. Noticeably, the API provides separate methods for normal (cached), non-coherent, and uncached accesses as well as flush and atomic operations. Separation of these methods is required for accurate implementation of TSO memory consistency as explained in the previous subsection.

Figure 5 shows how execution flows from a SW context (i.e. a fiber executing the SW model) to the simulator context (i.e. the main fiber for simulation). As in normal execution, whenever the user application reads a shared variable, the code jumps to and executes the `TM_READ` function in the STM library. Note that the application has executed natively up to this point and has not yet consumed a single simulation cycle. However, instead of actually sending a transactional read message to the FPGA, the HAL part of the STM calls into the BFM API (`SIM_Noncoh_write`) which eventually injects a packet into the simulator (`BUS_Inject`) and switches context to simulator execution (`SIM_return_simulator`). However, before injecting each packet, the BFM adds random idle cycles (`SIM_wait_cycles`).

Figure 6 shows execution flow from the simulator context to SW context. The network simulator, which performs simple cycle-based simulation, calls `clock()` function for each BFM at each simulation cycle. Since the software context is either idle-waiting or waiting for packet transfer, the BFM checks those conditions and resumes any software model that is ready by context-switching back to the software model (`SIM_return_core_context`).

4. RESULTS AND DISCUSSION

Our new co-simulation environment (Section 3.2), was extremely useful for verifying the functional correctness of our system. With this environment, we first simulated our old design and confirmed that the old system fails at cases like Figure 2. Note that the new environment is able to generate such cases, while our previous ISS-based simulation wasn't; also it is easy to track down errors in this environment, which

Method Name	Description
SIM_write/read	Normal cacheable memory accesses.
SIM_noncoh_write/read	Memory accesses bypass cache but stay in store-buffer.[a]
SIM_uncached_write/read	Memory accesses bypass cache and store-buffer[b]
SIM_memory_fence	Flush pending memory accesses
SIM_atomic_cas	Atomic compare and swap
SIM_wait_cycles	Insert idle cycles

[a] We keep two separate store-buffers; one for cacheable accesses and the other for non-coherent ones.
[b] An uncached access flushes all pending memory accesses.

Table 2: API of our processor BFM.

was nearly impossible in our previous FPGA execution. Afterwards, we used this co-simulation environment intensively to validate functional correctness of the new design.

The merits of this BFM-based software model approach can be summarized as follows. First, it enabled us to execute the application software compiled with the entire STM library in conjunction with the target hardware simulation. We remind the readers that our STM library itself is a very complicated piece of software which orchestrates the correct execution of our tightly-coupled HW-SW system. In other words, there was *no* way of writing *simple test-bench* for our system level simulation but executing the whole software.

Second, the co-simulation environment could generate many variations of overlapped transactions. On one hand, randomized timing helped to explore corner cases in data-race conditions. Note that this randomized timing does not accurately reflect actual software execution timing at all. However, for the purpose of functional verification, this is better than an un-timed or even ISS-based approach that does not provide for such variation. This is analogous to the usefulness of a random-based test bench in unit-level hardware verification.

On the other hand, accurate implementation of the memory consistency model was another key factor in successful test-pattern generation. For example, our `stm_read` instruction is composed of an actual memory read followed by a message passed to the FPGA – the race between the read request packet and the message created many interesting corner cases in our design. Note that in-order packet generation, as used in previous studies [21, 6, 17], is not able to capture such cases. Overall, the consistency model creates interleaving of data accesses from a single core; timing randomization among multiple cores.

Third, we were able to execute many different simulations in a relatively short time. The absolute simulation speed was still dominated by the HDL simulator and its PLI interface. However, since most of the software model was executed natively, there was no waste of valuable simulation cycles to execute instructions that were not necessary for functional verification. Table 3 compares the performance of all of our co-verification environments. We remind the users that SW-model based approach only consumes simulation cycles for required instructions, i.e. shared memory access; other instructions, such as local memory access, branches, and arithmetic operations are executed natively at the speed of underlying workstation.

Fourth, the co-simulation environment provided a very helpful error detection mechanism, which was impossible in native execution on FPGA. Figure 7 displays an example of an error-identified log generated by our simulation environment. Note that the last row points out which address

Environment	Speed (cycles/secs)	Time[b]
Prototype	1.8 (GHz)	$<$ 1 ms
(SW-model[a]+ HDL sim)	$\sim$ 400	$<$ 15 mins
(SW-model[a]+virtual model)	$\sim$ 3M	$<$ 100 ms
ISS[a]+virtual model	$\sim$ 50K	$<$ 1.5 hrs

(a) Simulation time was measured on a 2.3Ghz Nehalem machine.
(b) Time to execute our small application.

Table 3: Simulation Speed of our Verification Environments

cycle	T_1	T_2	...
...		...	
1048976	– TX Begin –		
1048990		W 1000786h	
1059102	R 1000786h		
1070428		– TX commit –	
1078824		C 1000786h	
1081034		– TX end –	
1081106	– TX commit –		
1081300	Error: T_1 got Commit Okay.		
1081300	It should be violated by 100786h		

Figure 7: Example log and error detection from our simulation.

is violating serialize-ability From this log, we were able to relate SW context and HW status, since the simulation cycle is shared by both SW simulation and HDL simulation. The ability of deterministic re-play of problematic execution was also crucial for debugging. As a result, we found bugs not only in HW, but also in SW or in protocol with this environment.

The lessons from our case study can be summarized as follows: (a) It is beneficial to combine a SW model with an interconnection simulator and an HDL simulator for functional verification, as it enables fast simulation. This combination can be easily realized through a BFM processor model. (b) In such an environment, it is essential to accurately implement the memory consistency model and cache coherency in order to capture corner cases induced by data races. Adding timing randomization is also important. (c) Keeping a shadow log in the (non-simulated) memory can help track down data race-induced errors by exploiting the fact that the simulation is being sequentially executed.

In a sense, the above lessons are a minimal simulation abstraction for fast functional co-verification of tightly coupled HW-SW systems, where data race between multiple computation units becomes an issue. We believe these lessons are also valuable to other designers since the trend of modern digital system is to introduce more and more parallelism and heterogeneity in it; our prototype was an exemplar instance of such systems.

5. CONCLUSION

In this paper, we presented our HW/SW co-verification experience on a commodity multi-processor system with custom hardware. For the sake of functional verification of such a system, it was most effective to combine native SW execution with cycle-based interconnect simulation and HDL simulation by means of a processor BFM. However, our experiences showed that such BFMs should faithfully reflect the memory consistency models of their target processors and would benefit greatly from randomized packet injection timing in their network simulations. These requirements enable the co-simulation to generate a wide variety of data access interleavings, which is essential for co-verification of modern multi-processor systems.

6. ACKNOWLEDGEMENTS

This paper was supported by DOE contract, Sandia order 942017; Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order US1226344; NSF grant CCF-0546060; DARPA Contract, SEEC: Specialized Extremely Efficient Computing, Contract # HR0011-11-C-0007; NSF grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, and NVIDIA. Authors also acknowledge additional support from Oracle.

7. REFERENCES

[1] Cadence, Inc. Palladium series. `http://www.cadence.com/products/sd/palladium_series/pages/default.aspx`.

[2] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware acceleration of transactional memory on commodity systems. ASPLOS '11. ACM, 2011.

[3] J. Cornet, F. Maraninchi, and L. Maillet-Contoz. A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip. In *DATE*, 2008.

[4] W. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[5] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[6] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. RTOS-centric hardware/software cosimulator for embedded system design. In *CODES+ISSS*. ACM, 2004.

[7] S. Hong, S. Yoo, S. Lee, S. Lee, H. Nam, B. Yoo, J. Hwang, D. Song, J. Kim, J. Kim, et al. Creation and utilization of a virtual platform for embedded software optimization:: an industrial case study. In *CODES+ISSS*, 2006.

[8] O. S. Initiative. Systemc. `http://www.systemc.org`.

[9] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2006.

[10] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[11] P. McKenney. Memory ordering in modern microprocessors, part I. *Linux Journal*, 2005(136):2, 2005.

[12] Mentor Graphics, Inc. seamless. `http://www.mentor.com/products/fv/seamless/`.

[13] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In *DAC*, 2004.

[14] T. Oguntebi, S. Hong, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. FARM: A prototyping environment for tightly-coupled, heterogeneous architectures. In *FCCM*, 2010.

[15] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC*, 2003.

[16] J. Rowson. Hardware/software co-simulation. In *DAC*, 1994.

[17] L. Séméria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *ASP-DAC*, 2000.

[18] Sysnopsys, Inc. Virtual prototyping solutions. `http://http://www.synopsys.com/Systems/VirtualPrototyping/Pages/default.aspx`.

[19] J. Um, W. Kwon, S. Hong, Y. Kim, K. Choi, J. Kong, S. Eo, and T. Kim. A systematic IP and bus subsystem modeling for platform-based system design. In *DATE*, 2006.

[20] Xilinx, Inc. Bus functional model (bfm) simulation of processor intellectual property. `http://www.xilinx.com/support/documentation/application_notes/xapp516.pdf`.

[21] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. Jerraya. Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer. In *DATE*, 2003.

[22] M. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *ISPASS*. IEEE, 2007.