Bridging Pre- and Post-silicon Debugging with BiPeD

Andrew DeOrio, Jialin Li and Valeria Bertacco
University of Michigan
{awdeorio, lijl, valeria}@umich.edu

ABSTRACT

The growing complexity of modern chips has caused an increasing share of the verification effort to shift towards post-silicon validation. This phase is challenged by poor observability, limited off-chip bandwidth, and complex, concurrent communication interfaces. Furthermore, pre-silicon verification and post-silicon validation methodologies are very different and share little information between them. As as result, the diagnosis and debugging of post-silicon failures is very much an ad-hoc and time-consuming task that is largely unable to leverage the vast body of design knowledge available in pre-silicon.

We propose BiPeD, a novel methodology to identify the exact time and location of post-silicon bugs. During pre-silicon verification, BiPeD learns the correct behavior of a design's communication patterns. In post-silicon, this knowledge is used to detect errors by means of a reconfigurable hardware unit. When an error is detected, bug reproduction is not necessary: a diagnosis software algorithm analyzes information stored in the hardware unit to provide a wide range of debugging information. We show that our system provides accurate bug localization for a range of failures on the industrial-size OpenSPARC T2 design.

1. INTRODUCTION

As the industry moves deeper into the multi-core era, the number of components on a single die increases. At the system level, this creates many more interactions among components, often structured by communication protocols to facilitate inter-block operation. While hardware blocks are often verified thoroughly in standalone setups, the communication interfaces among blocks are often sources of latent bugs. Current approaches to mitigating bugs typically leverage two disjoint phases: pre-silicon verification of software models, and post-silicon validation of hardware prototypes. Due to the fast growing pace of digital designs, these two effort directions have evolved in an ad-hoc fashion, giving rise to vastly different pre- and post-silicon methodologies.

During **pre-silicon verification**, testing is performed on an abstract model of the chip. In this simulation-driven phase, every hardware unit and signal in the design is observable, and failures can be reliably reproduced by deterministic software models. Debugging is usually carried out with waveform viewers, where an engineer must sift through millions of cycles to locate a bug. Furthermore, the coverage of pre-silicon simulation is severely limited by slow simulation speeds. Moreover, complex chip-level interactions, such as communication protocols, may require system-level simulation: however, often the complexity of the system is beyond the capabilities of logic simulators. The result is an increasing reliance on post-silicon validation to verify these interactions.

Post-silicon validation begins when the first hardware prototypes become available, leveraging an emerging set of methodologies largely disjoint from pre-silicon verification. Tests are executed directly on silicon chips, enabling high-speed, high-coverage validation. It is during post-silicon validation that full-chip interactions are most readily verifiable, since all components are up and running. In contrast to the pre-silicon phase, observability is low on real hardware. The group of signals available for observation is small, and the period of time over which they can be monitored is limited by on-chip storage. Moreover, real silicon may execute non-deterministically, making bug reproduction a challenge. Finally, slow off-chip data transfer rates perturb the execution of the workload under test, further frustrating the debugging process.

Despite vast industry efforts, the challenges of verifying chips with ever-increasing complexity lead to escaped bugs. The protocols that define communication among blocks are particularly challenging, as they may contain bugs that escaped detailed block-level verification. In our **fault model**, we address failures that occur in the behavioral protocols governing the communication among a system's components via its interfaces. The root causes of the errors that we strive to detect and diagnose can be functional bugs, electrical failures and/or manufacturing faults.

1.1 Contributions

The goal of our work is to learn the correct behavior of a system's protocols during high-observability, low-speed pre-silicon verification, and then detect violations of these protocols during high-speed, low-observability post-silicon validation. To this end, we propose BiPeD, named for the two legs of our approach. When an error manifests, BiPeD provides the recent history of the protocol-level activity leading up to the bug. BiPeD eases the debugging process by:

- **Locating the bug** time and location, and tolerating noisy post-silicon environments without requiring failure reproduction.
- **Providing a rich set of debugging information**, including critical signals involved, modules, transactions, and activity history.
- **Incurring zero performance overhead** during regular operation, and transferring data off-chip only for a bug occurrence.
- **Bridging pre- and post-silicon validation** through the abstraction of protocols.

IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2012, November 5-8, 2012, San Jose, California, USA.

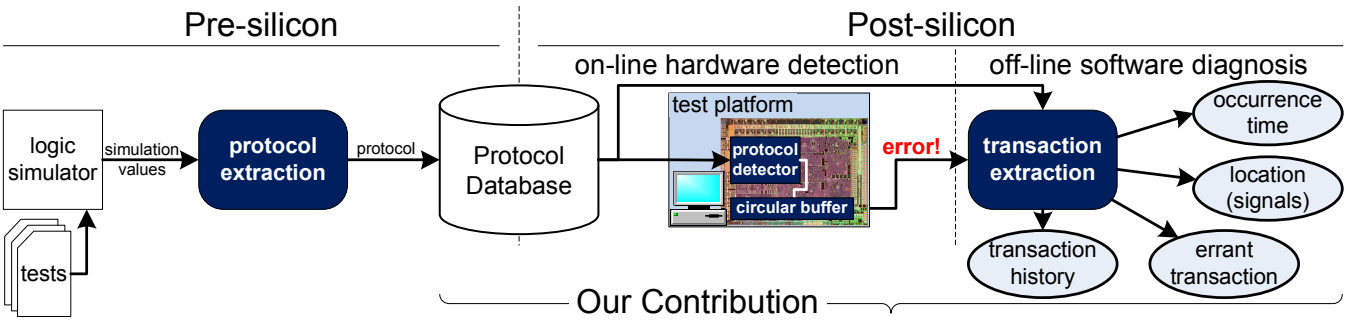

Figure 1: Methodology overview. During fully observable pre-silicon verification, BiPeD learns the protocols that define interfaces between design blocks. These protocols are then programmed into reconfigurable hardware during post-silicon validation, and checked at high-speed during high-coverage tests. When a bug is detected, a recent history of observed activity is transferred off-chip for analysis by a companion software algorithm. The algorithm extracts intuitive transaction diagrams, and presents to the user a rich set of debugging information.

2. RELATED WORK

The verification process of a modern digital design encompasses pre-silicon verification and post-silicon validation. While the pre-silicon phase uses abstract models and the post-silicon phase runs on prototype hardware, both provide an input workload to a system and checking of its output. When a check fails, the debugging process begins, a time-consuming, largely manual process.

Pre-silicon verification focuses on simulation, where any signal in the design can be observed. Signals are typically analyzed manually using a waveform viewer, and may require sifting through millions of simulation cycles. Data mining [10] can be used to create a high-level specification of the design, later used for verification. Sharing similar goals, the authors of Inferno [5] propose a higher level of abstraction used to understand RTL, using transactions, displayed as graphs to present the activity observed in simulation. Like Inferno, we leverage the abstraction of transactions to identify potentially buggy transactions as those representing abnormal behavior. While Inferno extracts transactions from complete traces during pre-silicon verification, BiPeD operates with partial information from post-silicon failures.

Formal tools, such as Murφ [6], can be leveraged to verify abstract models of protocols, but do not scale to full, chip-level implementations. Other work aims to learn properties for the purpose of formal verification [9, 10]. In contrast, our solution leverages the pre-silicon environment to learn correct activity, particularly inter-block interfaces, and then enforces it in post-silicon validation.

Post-silicon validation operates at full speed on early silicon prototypes, but is plagued by the difficulty of observing internal signals. During this phase, on-chip logic analyzers and flexible logging systems enable the observation of a few signals over a few thousand cycles, after which execution is suspended and the data slowly transferred off-chip. Thus, the pre-silicon techniques discussed above are not well-suited to post-silicon validation. There is a need for techniques that operate at a high abstraction level, such as [12], to support more effective post-silicon debugging. For example, the authors of [14] propose hard-coding high-level checkers on chip. Our flexible technique monitors targeted internal interfaces, requiring off-chip data transfer only when an error occurs.

An on-chip debugging infrastructure was proposed by the company DAFCA [1], instrumenting a design with observability hardware. In contrast, BiPeD learns protocols during pre-silicon verification, and checks them during post-silicon validation, providing trace information only when a protocol is violated. Post-processing connects the trace back to the high-level design. Complementary DAFCA-like instrumentation can augment the BiPeD methodology by providing additional debug information.

Specialized approaches targeting specific families of components have also been proposed, including speed paths [11]. IFRA/BLoG [13] covers a processor core, as long as errors are detected within approximately 1,000 cycles. Once a bug is detected, the responsible architectural block is identified. Blocks comprise about 10,000 gates, and the engineer must trace the bug's root cause among them. Our goal is to provide high-level, intuitive debugging information that can be quickly understood and used to locate bugs.

Software verification engineers face similar problems to those arising in hardware: generating checkers and validating correct behavior. For example, the Daikon tool [8] detects invariants and then checks a set of test executions against the invariants. Another approach [2] generates a specification for an API (application program interface) by means of software analysis. Program correctness is important to security concerns, and software verification can use abstract models to discover vulnerabilities [3].

3. OVERVIEW

BiPeD bridges the gap between pre-silicon verification and post-silicon validation: first, during pre-silicon verification, it learns the semantics of a design's protocols with protocol extraction software. Next, these semantics are checked at-speed by flexible (programmable) protocol detection hardware during post-silicon validation. When a check fails, the history of the activity observed on the failed interface is transferred off-chip for analysis by an off-line software algorithm. The resulting debugging information includes a trace of intuitive, high-level descriptions of the behavior leading up to the failure. Figure 1 shows an overview of this process.

Our approach leverages the high-level transaction representation of [5], which we generate using a slightly modified version of their open source distribution [4]. Based on [5], protocols describe the operation of a block or communication interface, and are represented by a graph with a vertex for each unique combination of signal values observed on the interface. There is an edge from vertex A to vertex B if there is at least one occurrence of signal combination A immediately followed by B in the simulation trace. Transaction diagrams represent the high-level semantic behavior of the design and are obtained by partitioning the simulation trace into multiple intervals, each corresponding to a transaction. Each distinct transaction typically repeats many times in a trace.

The protocols for a design's interfaces are generated during pre-silicon verification. First, interfaces are identified by designers, and each is defined by a set of the design's signals, usually control signals. Passing testcases are then run on the system, generating traces

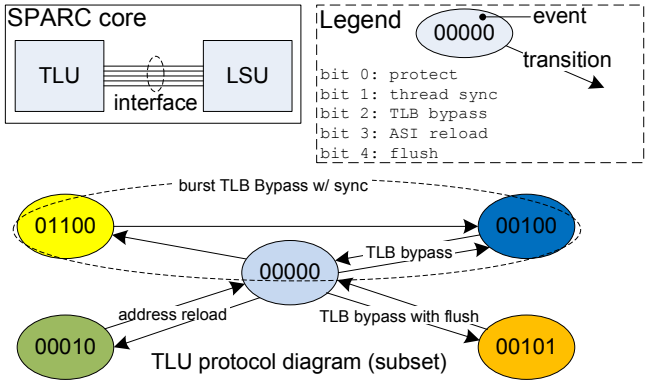

Figure 2: Protocol subset of the OpenSPARC T2 TLU/LSU interface, which defines its valid behavior. Each vertex represents an event, and each edge a transition. The bits in a vertex indicate signal values. Transactions are a subgraphs of the protocol, and are labeled.

for protocol extraction. The end result is a protocol representing the expected behavior of the interface, saved to a "protocol database" for use during post-silicon validation. This process is repeated for each interface selected, and it is illustrated with an example in Figure 2. As an alternative to automatic protocol extraction, protocols may also be specified by hand, for instance, using the designer's protocol description.

The example interface in Figure 2 is a subset of the OpenSPARC trap logic unit (TLU) interface, which regulates the communication between the TLU and Load/Store unit. Each unique set of values observed on the interface's signals constitute an event, that is a vertex; edges connect subsequent events, as shown in the protocol diagram at the bottom of the Figure.

3.1 Failure Detection

Our in-hardware solution monitors a number of interfaces simultaneously, confirming that the observed events and transitions conform to the protocol. During post-silicon validation, a number of protocols are programmed into "detector" hardware blocks, one block for each monitored interface. At runtime, the detector hardware units monitor the interfaces' activity to check that it conforms to the known protocol.

Figure 3 shows a diagram of a detector block. The detector checks all activity of its corresponding interface, by sampling all its signals at each clock cycle. These signals must be available by means of an on-chip debug infrastructure; frequently MUX selection trees are available that connect to many signals in the design. The interface signals are first routed to a content addressable memory (CAM) that matches their sampled values against known protocol events. If a matching event is found, a priority encoder converts it to an index value. Transitions are checked by consulting the transition CAM with a pair of indexes: the one from the current event and the one from the previous event – which is stored in a local register. The encodings of transitions are known by their indexes in the event CAM when the transition CAM is programmed. If either CAM fails to find a match, an error detection is flagged.

When a mismatch is detected by the hardware, execution is suspended and the event histories stored in the detection hardware's circular buffer are transferred off-chip for software analysis. The offline software analysis considers the event histories as well as the protocols as input. Some debugging information is immediately available: the time at which execution was suspended, and the interface(s) that flagged the error. The time provided corresponds to bug manifestation time. In addition, we can deduce the modules and signals involved in the bug by using the protocol database, which contains the names of the signals included in each interface and their connected modules.

The algorithm then processes the contents of the circular buffer for each interface that flagged an error: events are first decoded and then reconstituted into the interface signal values that they represent. The result is a partial trace of activity observed on the flagged interface(s). At this point, BiPeD applies a transaction extraction algorithm inspired by [5].

A number of observations make extraction on partial traces possible in our work. First, we noted that interfaces tend to return to a "stand-by" state between transactions. An example of this is the `00000` boundary in the TLU interface of Figure 2. We found that even with a partial trace, the stand-by event is typically the first repeated event. Thus, by using this event as stand-by we have been successful in extracting transactions in our experimental evaluation. We also explored other methods of identifying the best boundary events to separate transactions in partial traces. We considered storing the boundary events found in pre-silicon analysis along with their protocols in the protocol database. However, we found that different testcases would occasionally highlight different boundary events, and when using the union of all these events our transactions would be too fragmented. This latter approach may be useful during pre-silicon verification, when the design is changing frequently while the same testcases are re-executed. However, during post-silicon validation, many varied testcases lead to the extreme fragmentation mentioned above.

In developing BiPeD, we tried other methods of identifying the best boundary events to separate transactions. One possibility was for the protocol database to store not only events and transitions, but a set of boundary events as well. Transaction extraction would then use this pre-determined set. The problem that arose with this approach was that different testcases would sometimes yield different boundary events. When the set of all boundary events was used to extract transactions, they tended to be small and fragmented, due to large number of boundary events. While this approach may be useful during pre-silicon verification, when the design is changing, but the same testcases may be reused, it is not effective for post-silicon validation. Post-silicon validation enables the execution of many tests not run during the pre-silicon stage. Thus, we

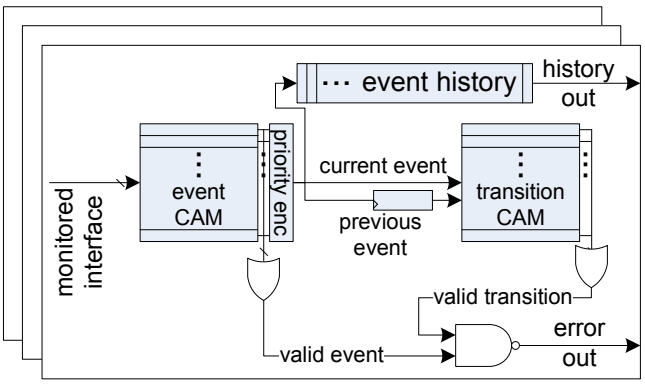

Figure 3: Protocol detector hardware units validate protocols during post-silicon validation. Each unit leverages two CAMs: one for events and one for transitions. Both pre-programmed with data collected during pre-silicon validation. The circular buffer maintains a history of events, transferred off-chip upon bug detection.

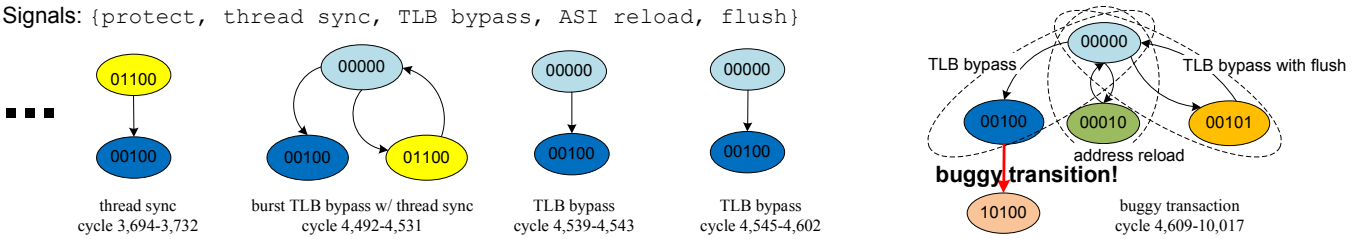

Figure 4: Transaction history example with a bug in the OpenSPARC T2 TLU interface. The buggy transaction is shown at the right, preceded by the four transactions that led to it. Bit vectors in each state represent signal values. The buggy transaction contains a transition edge not included in the approved protocol diagram. Dotted ovals indicate behaviors that appear within the buggy transaction.

found that dynamically detecting boundary events was more effective than saving and restoring them from the protocol database.

The result of a transaction extraction on a partial trace is a sequence of high-level, intuitive transaction diagrams representing the behavior of the interface preceding the failure. The last transaction in the sequence is the erroneous transaction which caused the protocol detector to suspend execution. BiPeD marks this transaction and indicates the exact event or transition that caused the mismatch. Thus, BiPeD is able to identify the erroneous transaction, event and transition.

In order to identify candidate signals for further debugging, BiPeD uses the errant event or transition. If it was a transition that caused the mismatch, the exact signals within the interface are identified by comparing the pair of events that the transition connects. On the other hand, if an unknown event is flagged, BiPeD compares this event to the other events in the interface (from the protocol database), identifying those events which are most similar to the errant event. Furthermore, these signals are used to identify the hardware block responsible for the error.

3.2 Hardware Reuse

The hardware protocol detector has the advantage of flexibility: not only can it be used for identifying and debugging failures during post-silicon validation, but can be applied to a variety of verification, runtime and performance tasks. Other verification tasks include tracking coverage: the detector hardware can be configured to monitor an area of the design, measuring not only the occurrence of a particular event, but a sequence of them. By augmenting the detection hardware with a small set of counters, these complex interactions could be measured. In the runtime verification sphere, solutions such as Semantic Guardian [16] observe control signals to check for invalid instantaneous combinations. This detection logic might be replaced by a protocol detector, enabling more complex interactions involving sequences of events over time to be checked.

In the context of a final product, a flexible protocol detector can be used as the basis for a hardware watchpoint system. For example Memtracker [15] uses bits of state added to each memory address along with a programmable state transition table to perfom memory access monitoring. BiPeD's programmable protocol detector could support this functionality with little modification.

3.3 Limitations

While BiPeD is effective in locating many types of hardware bugs, it has a number of limitations. First, the signals selected during pre-silicon verification for post-silicon observation have a critical impact on bug-finding capabilities. While we focus on control signals because of their central role in the activity and sequencing of on-chip protocols, incorrect computation affecting data signals may be missed. The tests used to learn protocols also affect BiPeD's ability to find bugs. High-coverage tests provide the best training and reduce false positives. However, in the event of a false positive, the protocol database can be amended to include the missed event or transition, and the hardware detectors can be updated.

4. CASE STUDY

We illustrate BiPeD's capabilities on the OpenSPARC T2 microprocessor. We identified 10 interfaces within the design, creating a protocol for each. In this case study, we focus on the TLU interface (Figure 2), which monitors signals that connect the trap logic unit (TLU) and the load store unit (LSU). The interface contains 5 signals: `protect`, indicating protected memory; `thread sync` occurs with high latency LSU operations, e.g., a D-cache miss. `TLB bypass` indicates that the instruction in the bypass stage bypassed the TLB. `ASI reload` indicates the ASI (address space identifier) reload is enabled, and `flush` monitors whether the instruction in the bypass stage is being flushed.

We first built the protocol database containing all 10 interfaces by running all our test cases, each with 100 different random seeds. More details on the protocols can be found in Section 5, Table 2. We then ran a buggy version of the design, with hardware protocol detectors programmed to monitor each interface. The buggy version contained an error in the LSU's `protect` signal, injected after 10,000 execution cycles.

The protocol detector monitoring the TLU interface quickly identified an erroneous transition at cycle 10,016. A history of events from the circular buffer in the protocol detector then underwent transaction extraction, and 70 transactions were identified. Figure 4 shows a subset of these transactions: four correct, and one buggy transaction. The transactions shown in the Figure begin with a thread sync, where thread synchronization occurs after the TLB is bypassed. Next, a burst TLB bypass with sync, was observed, a sequence of two TLB bypasses: the first bypass triggers synchronization and the second one without synchronization. The last two correct transactions were TLB bypasses, single bypasses with no synchronization.

The rightmost transaction in Figure 4 contains the errant transition, detected by a protocol detector mismatch. This transaction has some similarity to previously observed correct transactions: first, it contains a single TLB bypass (shown with dashed circle). Additional components in this complex transactions are a burst address reload, a burst TLB bypass with flush, as well as a TLB bypass. The transition from the single TLB bypass to data access protection is not included in the approved protocol diagram, thus an error is flagged. After detecting the bug, BiPeD provides accurate and intuitive debugging information: the relevant interface (TLU), modules (load-store unit and trap logic unit), the exact buggy transaction, the buggy signal (`protect`) and the detection cycle (10,016).

test case	description	length (cycles)
blimp_rand	hypervisor test	251,480
fp_addsub	floating point add/subt	913,093
fp_muldiv	floating point mult/div	238,343
isa2_basic	constrained-random	452,009
isa3_asr_pr	constrained-random	1,178,151
isa3_window	constrained-random	1,282,348
mpgen_smc	constrained-random	135,251
ldst_sync	thread sync. instrs.	64,570
n2_lsu_asi	load/store unit test	62,523
tlu_rand	trap logic unit test	591,434

Table 1: Workloads used for evaluation. The testcases are subset of those that ship with OpenSPARC T2.

interface	description	signals	bits	transitions	events
CPX	cache to processor	5	33	188	18
branch	EX branch logic	5	5	222	16
CCX	cache Xbar	6	20	215	23
memory	memory control unit	12	12	135	21
execute	execute unit	5	7	107	16
FPU	floating point unit	10	10	622	62
fetch	fetch unit	6	6	101	16
perf	performance monitor	3	5	23	6
TLU	trap logic unit	5	5	161	16
PCX	processor to cache	4	4	12	6

Table 2: Monitored interfaces. We instrumented the design to monitor 10 interfaces during program execution.

5. EXPERIMENTAL RESULTS

We employed BiPeD to locate failures in two hardware designs: a simple 5-stage pipeline, and the OpenSPARC T2 design. We simulated the OpenSPARC T2 design in its `cmp1` configuration, which included a SPARC core, cache, memory and crossbar. The 5-stage in-order pipeline implemented a subset of the Alpha ISA, and comprised approximately 5,000 lines of Verilog HDL code. Each design was simulated in behavioral Verilog, and equipped to monitor the protocols of 10 simultaneous interfaces. We observed similar trends in both designs, and thus we report only OpenSPARC's results in this section.

First, we ran the design free of bugs, training the protocol detector on 10 testcases (Table 1), ranging from 60,000 cycles to almost 12 million cycles in length. 100 variations of each testcase were run, using different random seeds to introduce execution variations with variable and random communication latencies. The number of events and transitions observed is shown in Table 2, as well as the number of signals and bits.

5.1 Protocol Detection

After building the protocol database with bug-free testcases, we employed the protocol detection hardware to detect a set of 10 bugs injected into the design, described in Table 3. Each buggy execution was simulated with different random variations (random seeds)

bug	description
branch	failure in branch to fetch communication
EX valid instr.	execution unit error
cache-proc req	erroneous cache-to-processor request
MEM read ack	erroneous memory load acknowledgment
FPU exception	floating point exception error
fetch thread id	LD/ST to fetch communication error
LSU data access	incorrect LSU access
table walk req	page table walk request
PCX stall	processor-to-cache communication stall
CCX/PCX req	processor/cache communication error

Table 3: Bugs injected, one at a time, after 10,000 cycles.

	OpenSPARC Bug									
Interface	branch	EX valid instr.	cache-proc req	MEM read ack	FPU exception	fetch thread id	LSU data access	table walk req	PCX stall	CCX/PCX req
CPX	1,719		16					f.n.		
branch	242									
CCX	16k	39	16							742
memory				223						
execute		16								
FPU		f.p.	22k	48k	739	48k			22k	
fetch						47				
perf.										
TLU							16			
PCX									767	764

Table 4: Bug detection latency (cycles) from bug injection to detection. Each bug was first detected after 22 cycles, on average, in the 5-stage design, and after 281 cycles in OpenSPARC. Additionally, most bugs were detected by one interface earlier than the others, demonstrating precise bug localization.

compared to protocol extraction: thus, no buggy execution matched any training execution. Each contained one bug, which was injected after 10,000 cycles. First, the detectors were programmed with the protocols described in Table 2. Each bug/testcase combination was then run with protocol detection hardware monitoring the 10 protocols, and the latency from bug injection to bug detection was recorded for each protocol.

Table 4 reports the latency (cycles) from bug injection to bug detection for each bug/testcase combination. We note that BiPeD reports the exact cycle of a protocol mismatch, while the table measures the time from bug injection to detection. For each bug in OpenSPARC, the first interface to detect the error flagged it within 281 cycles, on average. The first interface to identify the bug is marked in green (light shading). While most OpenSPARC bugs were detected by one interface many cycles before all others, the `cache-proc req` bug was detected by two interfaces simultaneously. In this case, two closely related interfaces caught the bug: the cache-to-processor crossbar (`CPX`) and the cache crossbar (`CCX`). We observed one false positive (red/dark shading, marked "f.p.") with the `EX valid instr.` bug, due to noise introduced by the new random variations, which had not been observed during training. One bug evaded detection (`table walk req`, which resulted in false negatives ("f.n.") marked with orange/medium shading. In this case, the bug signal was not part of any interface did not cause wider system effects detectable by other interfaces.

5.2 Protocol Extraction

We also evaluated the effectiveness of pre-silicon protocol detection, using the 10 testcases each with 100 random variations, for a total of 1,000 training tests. With each test execution, new events and transitions were added to the protocol database. Figure 5 shows the number of events and transitions in each interface, on average. We observed that, as the volume of training data increased, the number of events and transitions increased too, quickly with the first few tests, and then leveling off.

Training data impacts the number of false positives encountered by the detection phase. We applied leave-one-out-cross-validation to determine the effect of test data that differs from training data in both the random variations (random seed) as well as the workload. Here, 10 different protocol databases were used: each trained on 9 of the 10 available testcases. We found that leaving out a testcase had the effect of increasing the number of false positives, as shown

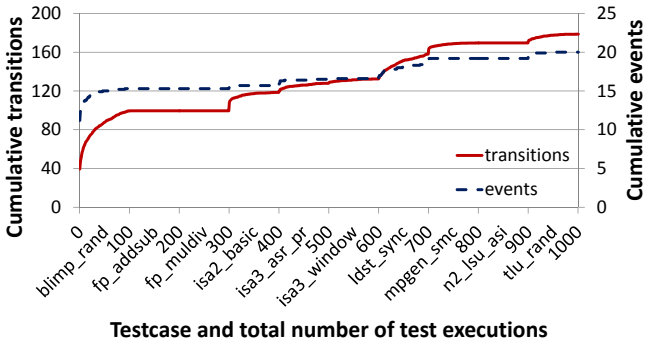

Figure 5: Protocol extraction. The plot reports the number of events and transitions in a protocol, on average. 10 testcases were used with 100 random seeds each for a total of 1,000 training tests per design, reflected on the X-axis. As the number of training tests accumulates, the size of the protocol approaches a consistent value. Steps in the graph represent the transition from one testcase to another.

in Figure 6. We found that excluding the `blimp_rand` testcase resulted in a 24% false positive rate among all bug/testcase combinations, underscoring its importance as a training test.

5.3 Transaction Extraction

We then explored the effect of circular buffer size on the number of transactions extracted. Figure 7 reports the total number of transactions, as well as unique transactions, for different buffer sizes. We observed that the total transactions scaled with the size of the buffer (on average), while unique transactions leveled off. As more unique transactions are discovered, the observance of repeated transactions increases, indicating that high-quality transaction are being extracted.

5.4 Area Overhead

We evaluated the area overhead of an implementation of the protocol detection hardware in Verilog HDL, synthesized with a 65nm TSMC target library. We found that the storage dominates the area.

Despite the complexity of the OpenSPARC T2 design, its protocols were limited in size. A detector sized to handle the largest OpenSPARC T2 interface can handle 62 events, each 33 bits wide, and 622 transitions. With this configuration, the resulting protocol detector required 15.3KB of storage and comprised 0.251 mm^2 in

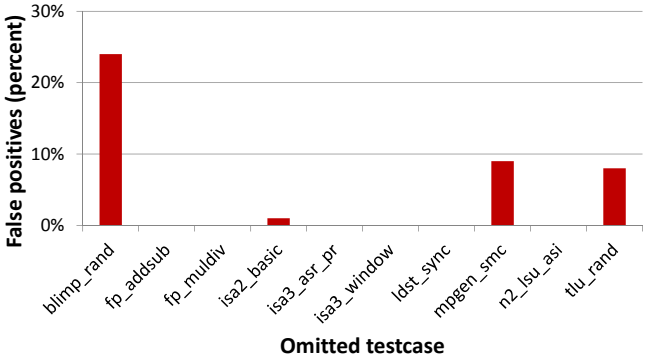

Figure 6: Effect of leave-one-out-cross-validation. The percentage of false positives with 9 training testcases and 1 testing in the OpenSPARC T2 design.

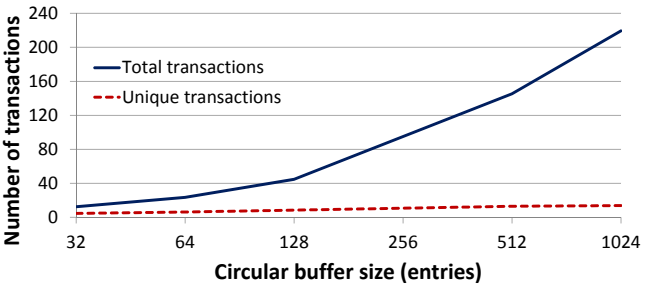

Figure 7: Transactions extracted from the circular buffer as the number of buffer entries changes. The plot shows both total, as well as the number of unique transactions. While the total increases with buffer size, unique transactions approach a constant.

our 65nm library. When compared to the total area of the OpenSPARC T2 chip (342 mm^2 [7]), the area overhead of 10 detectors (one for each monitored interfaces), is 0.7%.

6. CONCLUSIONS

We have presented a verification methodology that bridges pre-silicon verification with post-silicon validation. By leveraging high-level, compact, intuitive transactions and protocols, BiPeD is able to learn the behavior of a design's interfaces during pre-silicon verification and enforce it during post-silicon validation. Our system is capable of detecting bugs in the industrial-size OpenSPARC T2 design and able to accelerate post-silicon validation with intuitive debugging information,

7. REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, 2006.

[2] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. POPL*, 2002.

[3] C. Y. Cho, D. Babi, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proc. USENIX*, 2011.

[4] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen. INFERNO: An automatic semantic information extractor. http://www.eecs.umich.edu/inferno.

[5] A. DeOrio, A. B. Bauserman, V. Bertacco, and B. C. Isaksen. Inferno: streamlining verification with inferred semantics. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28, 2009.

[6] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proc. ICCD*, 1992.

[7] X. Dong and Y. Xie. System-level cost analysis and design exploration for three-dimensional integrated circuits (3D ICs). In *Proc. ASPDAC*, 2009.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programing*, 69(1-3), 2007.

[9] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proc. DAC*, 2005.

[10] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Proc. DAC*, 2010.

[11] R. McLaughlin, S. Venkataraman, and C. Lim. Automated debug of speed path failures using functional tests. In *Proc. VTS*, 2009.

[12] N. Nicolici and H. F. Ko. Design-for-debug for post-silicon validation: Can high-level descriptions help? In *Proc. HLDVT*, 2009.

[13] S.-B. Park, A. Bracy, H. Wang, and S. Mitra. BLoG: Post-silicon bug localization in processors using bug localization graphs. In *Proc. DAC*, 2010.

[14] E. Singerman, Y. Abarbanel, and S. Baartmans. Transaction based pre-to-post silicon validation. In *Proc. DAC*, 2011.

[15] G. Venkataramani and B. Roemer. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proc. HPCA*, 2007.

[16] I. Wagner and V. Bertacco. Engineering trust with semantic guardians. In *Proc. DATE*, 2007.