Quick Detection of Difficult Bugs for Effective Post-Silicon Validation

David Lin[1], Ted Hong[1], Farzan Fallah[1], Nagib Hakim[3], Subhasish Mitra[1, 2]

[1]Department of EE and [2]Department of CS
Stanford University, Stanford, CA, USA

[3]Intel Corporation
Santa Clara, CA, USA

Abstract

We present a new technique for systematically creating post-silicon validation tests that quickly detect bugs in processor cores and uncore components (cache controllers, memory controllers, on-chip networks) of multi-core System on Chips (SoCs). Such quick detection is essential because long error detection latency, the time elapsed between the occurrence of an error due to a bug and its manifestation as an observable failure, severely limits the effectiveness of existing post-silicon validation approaches. In addition, we provide a list of realistic bug scenarios abstracted from "difficult" bugs that occurred in commercial multi-core SoCs. Our results for an OpenSPARC T2-like multi-core SoC demonstrate: 1. Error detection latencies of "typical" post-silicon validation tests can be very long, up to billions of clock cycles, especially for bugs in uncore components. 2. Our new technique shortens error detection latencies by several orders of magnitude to only a few hundred cycles for most bug scenarios. 3. Our new technique enables 2-fold increase in bug coverage. An important feature of our technique is its software-only implementation without any hardware modification. Hence, it is readily applicable to existing designs.

Categories and Subject Descriptors

B.6.2 Reliability and Testing – Error-checking, Test generation, B.6.3 Design Aids – Verification, B.7.2 Design Aids – Verification, B.7.3 Reliability and Testing – Test generation, B.8.1 Reliability, Testing, and Fault-Tolerance.

General Terms

Reliability, Verification.

Keywords

Debug, Post-Silicon Validation, Quick Error Detection, Verification

1. Introduction

The purpose of post-silicon validation is to test manufactured ICs in actual systems to detect and fix design flaws (bugs). Traditional pre-silicon verification alone is no longer adequate because it is too slow and is often incapable of detecting electrical bugs. *Electrical bugs* manifest themselves only under specific operating conditions, such as voltage, frequency, and/or temperature corners [Patra 07]. Existing post-silicon validation approaches are *ad hoc*, and their costs are rising [Keshava 10]. Massive integration of a wide variety of components in complex System-on-Chips (*SoCs*) consisting of processor cores, accelerators, adaptive power / thermal / reliability control, and *uncore components* such as cache / memory / network controllers significantly exacerbate post-silicon validation challenges [Adir 11, Mitra 10, Singerman 11].

Post-silicon validation involves three activities: detecting a problem by applying proper stimuli, localizing the problem to a small region inside the chip, and fixing the problem through software patches, circuit editing, or silicon re-spin. The effort to localize the problem from an observed failure often dominates the cost of post-silicon validation [Josephson 06]. Long *error detection latency*, the time elapsed between the occurrence of an error due to a bug and its manifestation as an observable failure, limits the effectiveness of existing bug localization techniques. Bugs with error detection latencies longer than a few thousand clock cycles are highly challenging because it is extremely difficult to trace too far back in history for bug localization.

This paper makes the following contributions:

1. We present a list of bug scenarios obtained by analyzing "difficult" bugs (from proprietary bug databases) that occurred in latest commercial multi-core SoCs. Researchers can use these bug scenarios to quantify the benefits and drawbacks of existing and new validation techniques. Such quantification is essential in advancing the field of design validation.

2. We demonstrate that bugs in uncore components of SoCs can result in very long error detection latencies of several millions to billions of clock cycles unless special attention is paid to shorten these long error detection latencies. We also demonstrate that "typical" post-silicon validation tests with "end result checks" (that check results upon test completion) or "typical" self-checking tests are inadequate in shortening these long error detection latencies.

3. We present a new Proactive Load and Check (*PLC*) technique for **systematically** creating post-silicon validation tests with very short error detection latencies to quickly detect bugs in both uncore components and processor cores of multi-core SoCs. Simulation results of our PLC tests for a complex OpenSPARC T2-like multi-core SoC [OpenSPARC] demonstrate:

(a) Several orders of magnitude improvement in error detection latencies for bugs inside processor cores and uncore components. The error detection latencies of PLC tests are within a few hundred cycles for most bug scenarios we simulated.

(b) 2-fold improvement in the coverage of bug scenarios.

Our PLC tests can be implemented entirely in software with no hardware modifications. Hence, they can be readily applied to existing designs.

Section 2 presents a comprehensive list of bug scenarios. Section 3 describes our new technique for systematically creating post-silicon validation tests. In Sec. 4, we present simulation results to demonstrate the effectiveness of our technique. Related work is discussed in Sec. 5, followed by conclusion.

2. Bug Scenarios

A comprehensive list of bug scenarios is critical for understanding the limitations of existing validation techniques and for evaluating new approaches. Toward this end, we compiled a list of realistic bug scenarios by analyzing reports of "difficult" bugs (primarily "logic" bugs) detected during validation of actual multi-core SoCs. These bug scenarios are considered "difficult" because of very long debug times as indicated in bug reports. (A more precise definition of "difficulty" is desirable).

We performed extensive analysis of various bug reports, and worked closely with validation teams to abstract these bugs into bug scenarios using higher-level descriptions while removing product-specific details. As a result, several actual bugs are abstracted into a single bug scenario. Each bug scenario is decomposed into a bug activation criterion (Table 1a) and a bug effect (Table 1b).

DAC'12, June 3–7, 2012, San Francisco, CA, USA.

Bug activation criterion is the condition that must be satisfied to activate a bug. In Table 1a, criteria 1-4 correspond to cache controller bugs, criterion 5 corresponds to bugs inside cache / memory controllers and on-chip networks, and criteria 6-8 correspond to processor core bugs. Since we abstracted the activation criteria from "informal" bug reports, it is possible that all activation conditions might not have been completely captured.

Bug effect is defined as the incorrect behavior resulting from bug activation. In Table 1b, effects A-E correspond to cache controller bugs, effect F corresponds to memory controller bugs, effect G corresponds to interconnection network bugs, and effects H-J correspond to bugs inside processor cores.

Tables 1a and 1b allow us to implement each bug scenario using various micro-architectural and RTL simulators (details in Sec. 4). One can create families of bug scenarios by adjusting integer parameters *X* and *Y* in Tables 1a and 1b. For example, pairing bug activation criterion 2, for *X*=10, with bug effect A produces the following bug scenario:

> Two stores in 10 cycles to the same cache line cause cache coherence message for that line to be dropped.

Table 1a. Bug activation criteria.

Uncore components	1. Two stores in *X* clock cycles to different cache lines.
	2. Two stores in *X* clock cycles to the same cache line.
	3. Two stores in *X* clock cycles to adjacent cache lines.
	4. Two cache misses in *X* clock cycles.
	5. A sequence of load and/or store operations in *X* clock cycles.
Processor core	6. Data forwarding between pipeline stages.
	7. Two branch instructions in *X* clock cycles.
Other	8. Any clock cycle chosen at random.

Table 1b. Bug effects.

Uncore components	A. Next received cache* coherence message dropped.
	B. Next received cache* coherence message delayed.
	C. Next store operation not allocated a cache* line.
	D. Next store update to cache* delayed by *Y* clock cycles.
	E. Next data accessed from cache* corrupted.
	F. Next data coming from main memory to cache* / core* corrupted.
	G. Processor core's* load value corrupted.
Processor core	H. Core* jumps to incorrect (random) address in the next cycle.
	I. Error in decoding next instruction's operand inside core*.
	J. Processor core* incorrectly decodes next instruction to a NOP instruction.

* where activation criterion is satisfied.

While there is on-going work on understanding electrical bug behaviors [Gao 11, Hong 10, McLaughlin 09], there exists little consensus on what constitutes accurate logic bug models [ITRS 09]. Earlier researchers analyzed logic bugs using research chips, class projects, and published errata pages [Constantinides 08, DeOrio 08, 09, Ho 95, Van Campenhout 00, Velev 03]. Bug scenarios resulting from Tables 1a and 1b subsume bugs in [DeOrio 08, 09, Ho 95, Velev 03]. It is difficult to compare bugs in [Constantinides 08, Van Campenhout 00] vs. our bug scenarios. This is because [Constantinides 08] provides RTL-specific bug examples, and [Van Campenhout 00] focuses on implementation-dependent root-cause analysis, e.g., missing inputs, incorrect signal sources.

3. Post-Silicon Validation Tests Targeting Uncore Components

Traditional post-silicon validation tests are inadequate for shortening error detection latencies, especially for bugs in uncore components of complex SoCs. Consider the example in Fig. 1a. When Core 1 stores 1 to memory location *A* (*mem*[*A*]), a bug in an uncore component, e.g., cache or memory controller, can result in data corruption. The value stored in *mem*[*A*] itself may be corrupted, or a local (cached) copy of *mem*[*A*] for Core 2 may not be properly updated. After a very long time, Core 2 uses the corrupt value from *mem*[*A*] and produces incorrect outputs. Tests with "end result checks" that check for expected output values upon test completion result in very long error detection latencies (Fig. 1a).

Following the above example, the corrupt value read by Core 2 can result in a livelock / deadlock (Fig. 1b). Techniques for detecting livelocks / deadlocks [Bayazit 05, Chandy 83] are inadequate because it is already too late when the livelock / deadlock happened (long after *mem*[*A*] got corrupted).

Self-checking tests [Aharon 95, Raina 98, Wagner 08], including QED tests that shorten error detection latencies for bugs inside processor cores, are not sufficient either. As shown in Fig. 1c, it is already too late when checking occurs in Core 2. Assertions for post-silicon validation also have similar challenges (more details in Sec. 5). A self-checking test variant, referred to as *store readback* test (Fig. 1d), performs a load operation on the same core that performed the store operation (Core 1 in this case) to check if the loaded value matches the stored value. However, a bug which does not correctly update a local (cached) copy of *mem*[*A*] in Core 2 may not be detected by this check.

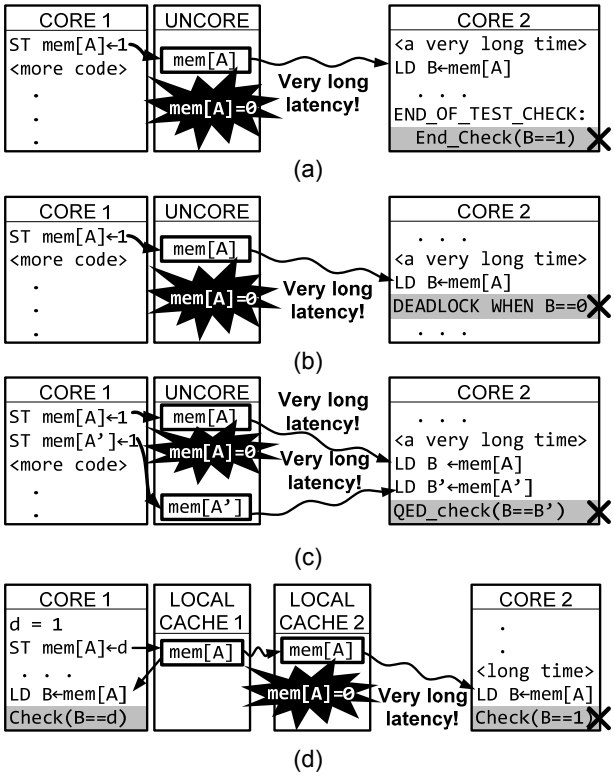

Figure 1. (a) End result check. (b) Deadlock detection. (c) QED test for processor cores. (d) Store readback test.

Checkpoint-based tests, that periodically checkpoint a system and compare checkpointed values with expected values (obtained from simulations), are also insufficient. System-level checkpointing [Silas 03] of processor registers and memory states is complicated for complex many-core SoCs. Hence, frequent system-level checkpointing may not be feasible. To reduce complexity, only processor registers may be locally checkpointed. In that case, our bug example in Fig. 1 will have a long error detection latency (detected only after Core 2 uses *mem*[A]).

3.1. Proactive Load and Check (PLC) Transformation

We overcome the above error detection latency challenges by statically transforming existing validation tests into new tests (during compile time) using a special **Proactive Load and Check (*PLC*) transformation**. Unlike QED tests targeting processor cores [Hong 10], the PLC transformation does not solely rely on (identical or diverse) re-execution of instructions in the original test. Instead, it inserts special PLC operations at very fine granularities across memory (and I/O) address spaces using targeted instructions. The idea is to "strategically" perform loads on all threads executing on all processor cores from selected variables and to insert self-consistency checks on those variables. The PLC transformation is compatible with QED. Hence, bugs inside processor cores can also be detected with short error detection latencies.

In this paper, PLC is implemented as a software-only technique. Hence, it does not require any hardware modification and can readily fit into existing post-silicon validation flows. However, PLC can be further enhanced with hardware support, a complete description of which is beyond the scope of this paper.

Next, we provide a detailed overview of the steps required to implement PLC for an OpenSPARC T2-like SoC.

Step 1: Initialization

We first transform an existing validation test into a QED test, e.g., using the EDDI-V transformation [Hong 10] (Fig. 2). EDDI-V bounds error detection latencies for bugs inside processor cores by strategically inserting duplicated instructions and checks in the original test. EDDI-V uses different (but identically initialized) registers, memory, and variables for the duplicated instructions. The results of the duplicated instructions are compared with the original ones, and an error is indicated upon mismatch.

A global array variable called *PLC_List* is created. Each *PLC_List* entry consists of the tuple *<original variable pointer, EDDI-V variable pointer>*. The *original variable pointer* points to a variable selected for PLC. Variable selection strategies for PLC are discussed later. The *EDDI-V variable pointer* points to the corresponding duplicated variable created by EDDI-V (e.g., *<&a, &a'>* and *<&b, &b'>* in Fig. 2). For dynamically allocated variables, the pointer values are determined during runtime by the memory allocator. For statically allocated variables, the pointers are known *a priori* via source code labels.

All variables listed in the *PLC_List* must be protected against race conditions between stores to these variables by one thread and PLC operations (details later) by another thread. Such race conditions can occur due to unexpected interleaving of the four operations: store to an original variable in the *PLC_List*, store to the corresponding EDDI-V variable, load from an original variable in the *PLC_List*, and load from the corresponding EDDI-V variable. We achieved this by locking the original variable and the corresponding EDDI-V variable pair during store and load operations to the variable pair (Fig. 2). More efficient techniques to protect against race conditions exist using architectural support (e.g., store double word) or software-only techniques [Larsson 04]. The details of such techniques are beyond the scope of this paper.

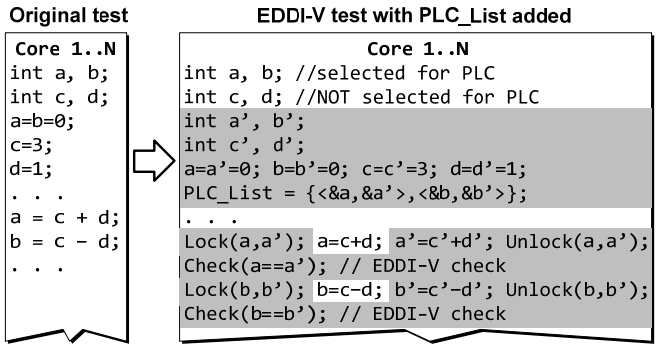

Figure 2. PLC transformation Step 1: initialization.

Step 2: PLC Operation Insertion

After initialization, the PLC transformation inserts proactive load and check operations in each thread in each processor core. Each *Proactive Load and Check (PLC) operation* (Fig. 3) performs the following function. It iterates once over all tuples in *PLC_List*. For each tuple in the list, it locks the tuple and then loads the values from both the *original variable pointer* and the *EDDI-V variable pointer*. After the values are loaded, it unlocks the tuple. Next, the two loaded values are compared with each other. Under bug-free situations, the two values should **exactly** match. An error is indicated upon mismatch. Upon error detection, the system can be halted for bug localization and debug using a variety of existing techniques. Bugs affecting the *PLC_List* itself can be quickly detected because it is highly unlikely that the corrupted addresses corresponding to the *original variable pointer* and the *EDDI-V variable pointer* fields will contain the same data values.

PLC operations are inserted at periodic intervals in **each thread in each processor core**. This is necessary because bugs can affect various pathways between processor cores and uncore components. Furthermore, PLC operations check all variables in the *PLC_List* to cover situations in which some arbitrary variable (not necessarily a recently modified or used variable) is affected by a bug. For example, a store / load operation to one variable can trigger an electrical bug that creates an error in another variable.

Intuitively, highly frequent PLC operations are preferred for quick error detection. However, excessive PLC operations may result in excessive *intrusiveness*, i.e., the deviation in the execution behavior of the transformed test compared to the original test, which can adversely impact coverage of the transformed test. Our results in Sec. 4 do not show any such coverage degradation.

To minimize possible intrusiveness due to PLC operations, we introduce a *PLC_inst_min* parameter, defined as the minimum number of instructions in the same thread that must execute before a PLC operation is inserted. Similar to [Hong 10], code transformations, such as loop unrolling and loop splitting, may be required to satisfy the *PLC_inst_min* constraint.

Variable Selection Strategies for PLC

There can be several strategies to select variables to be included in the *PLC_List*:

1. One can include all variables in a given test. However, the resulting error detection latencies can be long, especially when PLC is implemented in software. This is because PLC operations must iterate through all variables, which can take a long time.

2. One can create a **family** of tests from a given test, where each test in the family selects a subset of variables (in the given test) for its *PLC_List*. The resulting tests are referred to as *PLC family tests*. There is a possibility that some bugs may not be detected quickly with short error detection latencies when PLC family tests are used. This is because a bug may or may not be activated by each

individual test in the family. Our results in Sec. 4 demonstrate that PLC family tests are highly effective in significantly shortening error detection latencies for the bug scenarios in Sec. 2.

3. If the inputs to a test are known *a priori* (which is often the case for validation tests [Bentley 01]), one can perform profiling to determine variables with store-to-load or load-to-load latencies longer than the desired error detection latency bounds, and include only those variables in the *PLC_List*.

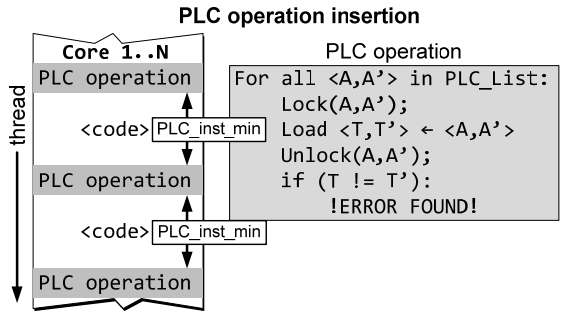

Figure 3. PLC transformation Step 2: PLC operation insertion.

4. Results

We evaluated the effectiveness of our PLC transformation technique by inserting the bug scenarios in Sec. 2 into a micro-architectural simulator. We used Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) [Martin 05] to simulate an OpenSPARC T2-like SoC [OpenSPARC], a 500 million-transistor design with 8 processor cores, 64 threads, private split L1 data and instruction caches, crossbar-based interconnects, 8-way banked L2 cache using directory-based cache coherence protocol, and 4 memory controllers (Fig. 4).

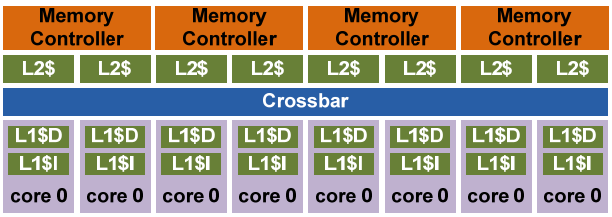

Figure 4. OpenSPARC T2-like SoC [OpenSPARC].

We implemented the bug scenarios in the simulator in the following way: for a selected bug scenario, we modified the source code of the simulated system to include two additional routines: one that constantly monitors the simulated system for the activation criterion, and another that inserts the bug effect in the system (initially disabled). When the activation criterion is satisfied, the routine that inserts the bug effect in the system is enabled to insert a single instance of the bug effect. Since the activation criterion can be satisfied multiple times during a simulation run, the bug effect can be inserted multiple times as well (i.e., once for each time the activation criterion is satisfied), which is consistent with the behavior of (logic) bugs in actual systems.

We ran a separate simulation experiment for each of the 80 bug scenarios (cross product of 8 bug activation criteria and 10 bug effects in Tables 1a and 1b). For each experiment, only one bug scenario is inserted. For bug activation criteria and bug effect parameters, we set X=10 clock cycles and Y=100 clock cycles because these values gave us "close" representations (on our simulation platform) of the "difficult" bugs found in the bug databases. Our PLC transformation is independent of X and Y, and is not engineered based on the bug scenarios. For each experiment, the selected bug scenario affects any one of the cores, L2 caches, and memory controllers (i.e., no specific component was pre-selected).

The results are summarized in Fig. 5 for the following tests: FFT and LU programs from the SPLASH-2 benchmark suite [Woo 95] in Figs. 5a and 5b, respectively, and a proprietary industrial post-silicon validation test targeting memory bugs (details omitted for confidentiality) in Fig. 5c. The results correspond to 8-threaded (single thread per core) versions of the tests. 64-threaded versions of the tests were not used because of slow simulation speed.

In Figs. 5a-c, the *original* tests are instrumented with "end result checks" only. Since the inputs to these tests are known *a priori*, the expected end results can be calculated.

For *PLC family tests* in Figs. 5a and 5b, we split all the heap variables in the original test into 1,024 groups. This is done by aggregating a list of all heap variables (known *a priori* since the inputs to the tests are known) and dividing the list into 1,024 groups. This results in 1,024 individual tests in the PLC family, each performing PLC operations for only the variables in its corresponding group. Each test in the family is run one after another. For each bug scenario, the error detection latency reported in Fig. 5a (5b) corresponds to the error detection latency when the bug scenario was detected for the first time by the PLC family tests.

For the industrial test, there is a single PLC family test consisting of all variables allocated on the memory. Register variables are not included for PLC because errors in these variables can be quickly detected by EDDI-V alone. Since this test targets memory bugs only, Fig. 5c considers memory bug scenarios (i.e., activation criteria 1-5 in Table 1a and effects A-G in Table 1b).

We performed PLC transformations at the C source code level with *PLC_inst_min* to be approximately 10 lines of code. Figures 5a-c also report results for *QED* tests obtained by transforming the original tests using the QED EDDI-V transformation only.

Table 2 shows the relative runtimes of various tests (normalized to the corresponding original test). The runtimes of PLC family tests are significantly longer than the original tests. That raises the question: Are the significant benefits of PLC tests due to their PLC operations or due to their longer runtimes? To answer this question, we performed a controlled set of experiments. For an original test, we created a version of the test whose runtime is approximately the same as that of the PLC family tests. This is done by keeping most of the PLC family tests intact but removing the error reporting code when an EDDI-V or PLC check detects an error. These tests are referred to as *OERT* tests (Original Equivalent RunTime tests). An OERT was not created for the industrial test because the runtime of the original test is already very long.

The following observations can be made from the results.

Observation 1: Post-silicon validation tests created using our PLC transformation technique shorten error detection latencies by several orders of magnitude for all bug scenarios in Sec 2. Error detection latencies of original tests can be extremely long: several millions or billions of cycles. QED tests targeting bugs inside processor cores have long error detection latencies for uncore bugs. In contrast, PLC tests have error detection latencies of only a few hundred cycles for most bug scenarios. These benefits come from the PLC operations and not from the longer runtimes of PLC tests.

Observation 2: Post-silicon validation tests created using our PLC transformation technique continue to detect bug scenarios detected by the original tests. There is not a single bug scenario that the original tests (or the QED tests or the OERT tests) detected but the PLC tests didn't. Note that, the activation criteria in Table 1a are fairly rare with only tens of activations per ten million cycles.

Observation 3: Post-silicon validation tests created using our PLC transformation technique detect up to 2-fold more bug scenarios that would otherwise remain undetected by the original tests, the QED tests, or the OERT tests. Our simulation experiments confirmed that each bug scenario was activated at least once by the original tests, the QED tests, and the OERT tests.

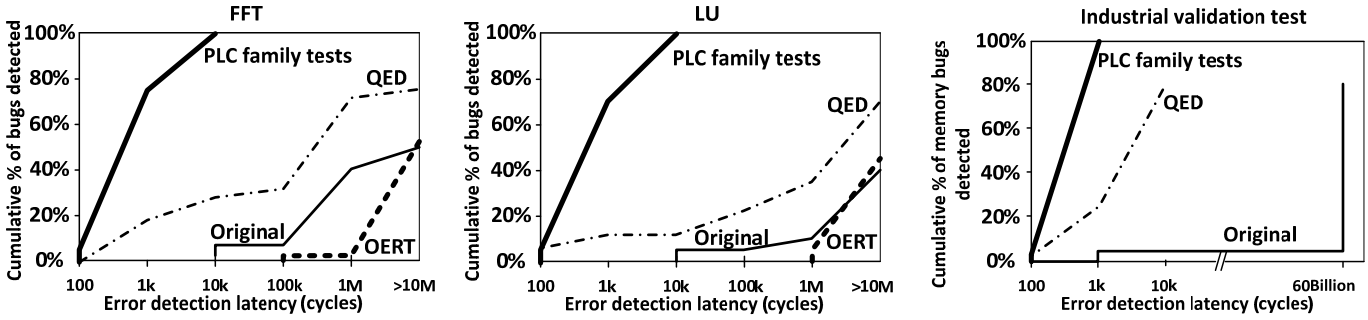

Figure 5. Error detection latencies and coverage of post-silicon validation tests. (a) FFT. (b) LU. (c) Industrial validation test.

Table 2. Runtimes normalized to corresponding original tests.

	FFT	LU	Industrial validation test
Original	1	1	1
QED	≈ 2	≈ 3	≈ 3
PLC family tests	≈ 1,024 x 28	≈ 1,024 x 32	≈ 35
OERT	≈ 1,024 x 28	≈ 1,024 x 32	NA

Since the PLC family tests have long runtimes, we experimented with an alternative variable selection technique for PLC. This variable selection technique first performs test code profiling to determine variables with long store-to-load latencies (since test inputs are known *a priori*). Each PLC operation only checks the most recently stored variables with long store-to-load latencies. Note that, unlike the store readback test in Fig. 1d, all threads executing on all cores (not just the thread which performed the store) perform these PLC operations. Figure 6 compares error detection latency and coverage of this PLC variable selection technique vs. store readback for the FFT test. As expected, the PLC test achieves better error detection latency and higher coverage compared to the store readback test. The runtime of this PLC test is 25 times longer than the original test (which is significantly shorter than the PLC family tests in Fig. 5).

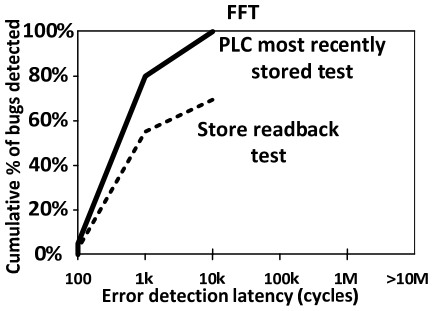

Figure 6. PLC test using most recently stored variables with long stored-to-load latencies vs. store readback.

5. Related Work

Existing work related to this paper can be classified into: post-silicon validation stimuli generation, memory scrubbing for fault-tolerant computing, and assertions for post-silicon validation.

Automatic Post-Silicon Validation Stimuli Generation

We demonstrated that test techniques that target bugs inside processor cores (e.g., [Aharon 95, Hong 10, Wagner 08]) alone are not sufficient for bugs in uncore components. Such tests can have very long error detection latencies for uncore bugs. [Raina 98] presented a technique for automatically generating random self-tests for microprocessor caches only, without bounds on error detection latencies. Our new technique ensures bounded error detection latencies for bugs in a variety of uncore components as well as processor cores. Furthermore, our technique is compatible with existing test generation techniques such as those using constraint satisfaction [Katz 12].

Memory Scrubbing for Fault-Tolerant Computing

Memory scrubbing [Abraham 83, Shirvani 00] is used in fault-tolerant computing to detect and correct errors inside memory arrays. The PLC transformation technique is inspired by the concept of memory scrubbing. However, there are significant differences:

1. Scrubbing generally uses error-correcting codes to target errors inside memory arrays, but may not detect errors due to bugs outside memory arrays such as cache or memory controllers.
2. Memory scrubbing generally occurs infrequently compared to PLC operations in post-silicon validation.
3. In post-silicon validation, reducing error detection latency is very important because debug time, rather than test execution time, often dominates overall post-silicon validation costs [Josephson 06]. Therefore, some test execution time penalties can be tolerated if error detection latencies can be significantly improved.
4. For post-silicon validation tests, test program inputs may be known *a priori* [Bentley 01]. This allows special transformations, e.g., through profiling (e.g., Fig. 6), to improve both error detection latencies and coverage while reducing test execution time penalties.
5. Failure containment and recovery are not primary concerns during validation.

Assertions for Post-Silicon Validation

The generation and use of assertions for post-silicon validation [Boule 07] are non-trivial. As noted in [Bentley 01], assertions have to be carefully crafted, and it is difficult to keep them up-to-date and to validate their correctness. While there are methods for automated assertion generation [Ernst 07, Hangal 05], these techniques are not widely applicable to all SoC components. In contrast, our approach allows a **structured** way of performing extensive checks, and can be automatically generated and validated.

6. Conclusion

Our new PLC technique for **systematically** creating post-silicon validation tests is highly effective in quickly detecting difficult bug scenarios inside uncore components as well as processor cores in multi-core SoCs. Our results demonstrate several orders of magnitude improvement in error detection latencies and 2-fold improvement in coverage simultaneously. Such short error detection latencies can significantly improve the productivity of post-silicon validation. Moreover, our technique is readily applicable to existing post-silicon validation flows because it

doesn't require any hardware changes. The list of bug scenarios, derived from difficult bugs that occurred in commercial multi-core SoCs, can act as excellent benchmarks to advance validation research.

Several opportunities exist to further enhance validation methodologies using our approach. Examples include: 1. Bug localization by analyzing the checks in our PLC tests that detected (or did not detect) errors. 2. Hardware support for PLC tests to reduce their runtimes. 3. Continued collection and analysis of difficult bugs in actual SoCs for better bug benchmarks. 4. Extending our approach to system verification using emulators, hardware accelerators, and prototyping and virtual platforms.

7. Acknowledgment

This research was supported in part by FCRP, GSRC, NSF, SRC, and Stanford Graduate Fellowship. The authors thank Eswaran S. and Sharad Kumar of Freescale, Jagannath Keshava and Sandip Ray of Intel, and Christine Cheng and Subhasis Das of Stanford University for their valuable inputs and support.

8. References

[Abraham 83] Abraham, J. A., E. S. Davidson, and J. H. Patel, "Memory System Design for Tolerating Single Event Upsets," *IEEE Trans. Nuclear Science*, Vol. 30, Issue 6, pp. 4339-4344, December, 1983.

[Adir 11] Adir, A., *et al*., "A Unified Methodology for Pre-Silicon Verification and Post-silicon Validation," *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 1-6, 2011.

[Aharon 95] Aharon, A., *et al*., "Test Program Generation for Functional Verification of PowerPC Processors in IBM*," Proc. IEEE/ACM Design Automation Conf.*, pp 279-285,1995.

[Bayazit 05] Bayazit, A. A., and S. Malik, "Complementary Use of Runtime Validation ad Model Checking," *Proc. IEEE/ACM Intl. Conf. Computer-aided Design*, pp. 1049-1056, 2005.

[Bentley 01] Bentley, B., and R. Gray, "Validating the Intel Pentium 4 Processor." *Intel Technology Journal*, Vol. 5 Issue 1, pp. 1-8, February, 2001.

[Boule 07] Boule, M., J.-S. Chenard, and Z. Zilic, "Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis," *Proc. IEEE Intl. Symp. Quality Electronic Design*, pp. 613-620, 2007.

[Chandy 83] Chandy, K. M., J. Misra, and L. M. Haas, "Distributed Deadlock Detection," *ACM Trans. Computer Systems,* Vol. 1, Issue 2, pp 144-156, May 1983.

[Constantinides 08] Constantinides, K., O. Mutlu, and T. Austin, "Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation," *Proc. IEEE/ACM Intl. Symp. Microarchitecture*, pp. 282-293, 2008.

[DeOrio 08] DeOrio, A., A. Bauserman, and V. Bertacco, "Post-Silicon Verification for Cache Coherence," *Proc. IEEE Intl. Conf. Computer Design*, pp.348-355, 2008.

[DeOrio 09] DeOrio, A., I. Wagner, and V. Bertacco, "DACOTA: Post-silicon Validation of the Memory Subsystem in Multi-Core Designs," *Proc. IEEE Intl. Symp. High-Performance Computer Architecture*, pp. 405-416. 2009.

[Ernst 07] Ernst, M. D., *et al*., "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, Vol. 69, Issues 1-3, pp. 35-45, December, 2007.

[Gao 11] Gao, M., P. Lisherness, and K.-T. Cheng, "Post-silicon Bug Detection for Variation Induced Electrical Bugs" *Proc. IEEE/ACM Asia and South Pacific Design Automation Conf.*, pp 273-273, 2011.

[Hangal 05] Hangal S., *et al*., "IODINE: A Tool to Automatically Infer Dynamic Invariants," *Proc. IEEE/ACM Design Automation Conf*., pp. 775-778, 2005.

[Ho 95] Ho, R. C., *et al*. "Architecture Validation for Processors," *Proc. ACM/IEEE Intl. Symp. Computer Architecture*, pp. 404-413, 1995.

[Hong 10] Hong, T. *et al*., "QED: Quick Error Detection Tests for Effective Post-Silicon Validation," *Proc. IEEE Intl. Test Conf.*, pp. 1-10, 2010.

[ITRS 09] http://www.itrs.net/Links/2009ITRS/Home2009.htm.

[Josephson 06] Josephson, D., "The Good, the Bad, and the Ugly of Silicon Debug," *Proc. IEEE/ACM Design Automation Conf*., pp. 3-6, 2006.

[Katz 12] Katz, Y., M. Rimon, and A. Ziv, "Generating Instruction Streams Using Abstract CSP," *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 15-20, 2012.

[Keshava 10] Keshava, J., N. Hakim, and C. Prudvi, "Post-silicon Validation Challenges: How EDA and Academia Can Help," *Proc. IEEE/ACM Design Automation Conf*., pp. 3-7, 2010.

[Larsson 04] Larsson, A., *et al.*, "Multi-Word Atomic Read/Write Registers on Multiprocessor System," *Lectures Notes in Computer Science*, Vol. 3221, pp. 736-748, 2004.

[Martin 05] Martin, M., *et al.*, "Multifacet's General Execution-Drive Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, Vol. 33, Issue 4, pp. 92-99, November, 2005.

[McLaughlin 09] McLaughlin, R., S. Venkataraman, and C. Lim, "Automated Debug of Speed Path Failures Using Functional Tests," *Proc. IEEE VLSI Test Symp*., pp. 91-96, 2009.

[Mitra 10] Mitra, S., *et al.*, "Post-Silicon Validation Opportunities, Challenges and Recent Advances," *Proc. IEEE/ACM Design Automation Conf*., pp. 12-17, 2010.

[OpenSPARC] "OpenSPARC: World's First Free 64-bit Microprocessor," http://www.opensparc.net.

[Patra 07] Patra, P., "On the Cusp of a Validation Wall," *IEEE Design & Test of Computers*, Vol. 24, Issue 2, pp. 193-196, March, 2007.

[Raina 98] Raina, R., and R. Molyneaux, "Random Self-Test Method Applications on PowerPC™ microprocessor cache," *Proc. ACM/IEEE Great Lakes Symp. VLSI*, pp 222-229, 1998.

[Shirvani 00] Shirvani, P. P., N. R. Saxena, and E. J. McCluskey, "Software-Implemented EDAC Protection Against SEUs," *IEEE Trans. on Reliability*, Vol. 49, Issue 3, pp 273-284. September, 2000.

[Silas 03] Silas, I., *et al*., "System-Level Validation of the Intel® Pentium® M Processor," *Intel Technology Journal*, Vol. 7 Issue 2, pp. 37-43, May 2003.

[Singerman 11] Singerman, E., Y. Abarbanel, and S. Baartmans, "Transaction Based Pre-To-Post Silicon Validation," *Proc. IEEE/ACM Design Automation Conf.*, pp. 564-568, 2011.

[Van Campenhout 00] Van Campenhout, D., *et al.*, "Collection and Analysis of Microprocessor Design Errors," *IEEE Design & Test of Computers,* Vol. 17, Issue 4, pp. 51-60, Oct-Dec, 2000.

[Velev 03] Velev, M. N., "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs", *Proc. IEEE Intl. Test Conf*., pp. 138-147, September, 2003.

[Wagner 08] Wagner, I., and V. Bertaco, "Reversi: Post-Silicon Validation System for Modern Microprocessors," *Proc. IEEE Intl. Conf. Computer Design*, pp. 307-314, October, 2008.

[Woo 95] Woo, S. C., *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. ACM/IEEE Intl. Symp. Computer Architecture*, pp. 24-36, 1995.